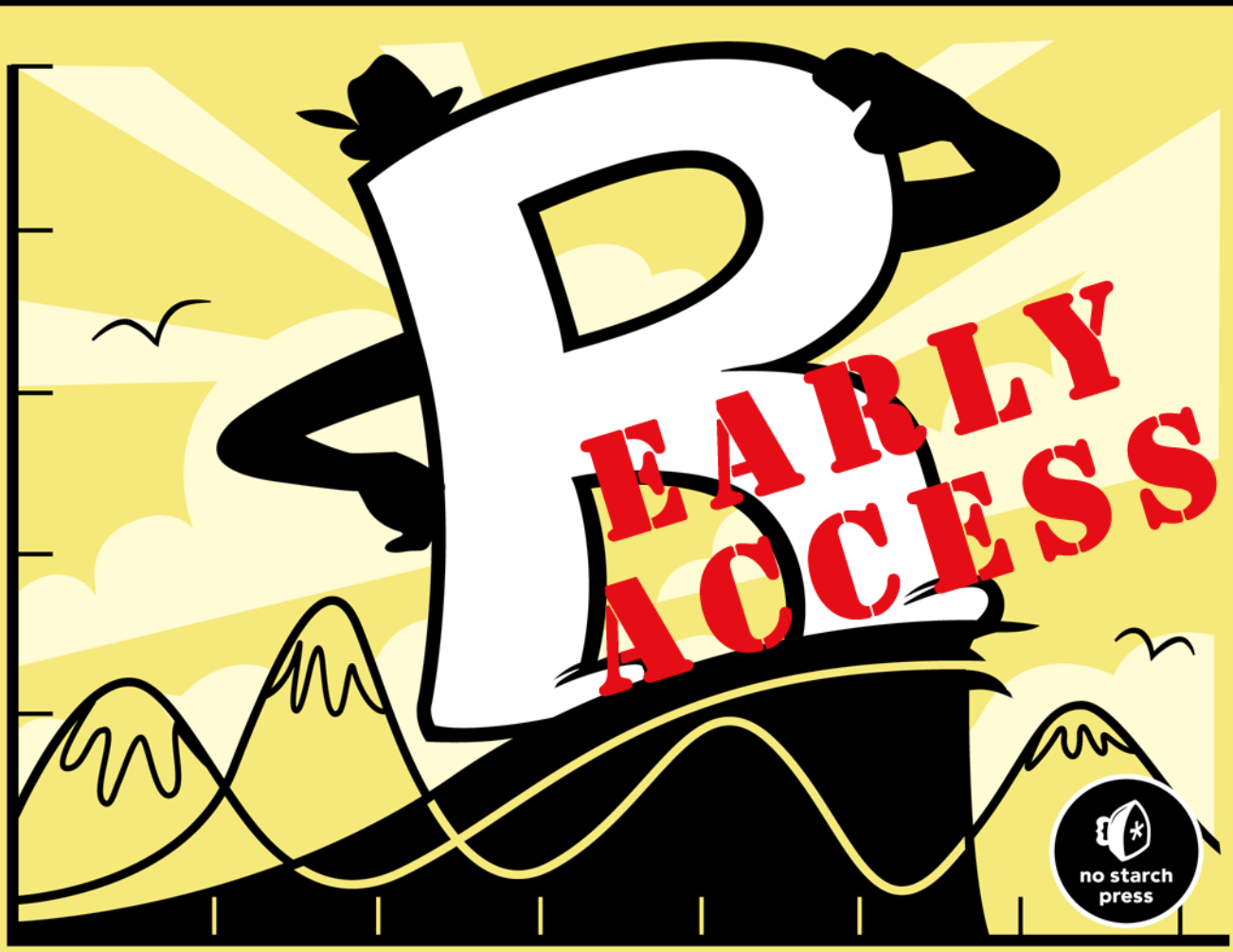
THE BOOK OF R

A FIRST COURSE IN PROGRAMMING AND STATISTICS

TILMAN M. DAVIES



NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Book of R* by Tilman M. Davies! As a prepublication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

THE BOOK OF R

TILMAN M. DAVIES

Early Access edition, 12/16/15

ISBN-10: 1-59327-651-6
ISBN-13: 978-1-59327-651-5

Publisher: William Pollock
Production Editor: Riley Hoffman
Cover Illustration: Josh Ellingson
Developmental Editors: Seph Kramer and Liz Chadwick
Technical Reviewer: Debbie Leader
Copyeditor: Kim Wimpsett
Compositor: Riley Hoffman
Proofreader: Paula L. Fleming

BRIEF CONTENTS

The chapters in **red** are included in this Early Access PDF.

PART V: ADVANCED GRAPHICS

APPENDICES

CONTENTS IN DETAIL

2

NUMERICS, ARITHMETIC, ASSIGNMENT, AND THE VECTOR

In its simplest role, R can function as a mere desktop calculator. In this chapter, I'll discuss how to use the software for arithmetic. I'll also show how to store results so you can use them later in other calculations. Then, you'll learn about vectors, which let you handle multiple values at once. Vectors are an essential tool in R, and much of R's functionality was designed with vector operations in mind. You'll examine some common and useful ways to manipulate vectors and take advantage of vector-oriented behavior.

2.1 R for Basic Math

All common arithmetic operations and mathematical functionality are ready to use at the console prompt. You can perform addition, subtraction, multiplication, and division with the symbols `+`, `-`, `*`, and `/`, respectively. You can create exponents (also referred to as *powers* or *indices*) using `^`, and you control the order of the calculations in a single command using parentheses, `()`.

2.1.1 Arithmetic

In R, standard mathematical rules apply throughout, and follow the usual left-to-right order of operations: parentheses, exponents, multiplication, division, addition, subtraction (PEMDAS). Here's an example in the console:

```
R> 2+3
[1] 5
R> 14/6
[1] 2.333333
R> 14/6+5
[1] 7.333333
R> 14/(6+5)
[1] 1.272727
R> 3^2
[1] 9
R> 2^3
[1] 8
```

You can find the square root of any non-negative number with the `sqrt` function. You simply provide the desired number to `x` as shown here:

```
R> sqrt(x=9)
[1] 3
R> sqrt(x=5.311)
[1] 2.304561
```

When using R, you'll often find that you need to translate a complicated arithmetic formula into code for evaluation (for example, when replicating a calculation from a textbook or research paper). The next examples provide a mathematically expressed calculation, followed by its execution in R:

$$10^2 + \frac{3 \times 60}{8} - 3$$

```
R> 10^2+3*60/8-3
[1] 119.5
```

$$\frac{5^3 \times (6-2)}{61-3+4}$$

```
R> 5^3*(6-2)/(61-3+4)
[1] 8.064516
```

$$2^{2+1} - 4 + 64^{-2^{2.25-\frac{1}{4}}}$$

```
R> 2^(2+1)-4+64^((-2)^(2.25-1/4))
[1] 16777220
```

$$\left(\frac{0.44 \times (1-0.44)}{34}\right)^{\frac{1}{2}}$$

```
R> (0.44*(1-0.44)/34)^(1/2)
[1] 0.08512966
```

Note that some R expressions require extra parentheses that aren't present in the mathematical expressions. Missing or misplaced parentheses are common causes of arithmetic errors in R, especially when dealing with exponents. If the exponent is itself an arithmetic calculation, it must always appear in parentheses. For example, in the third expression, you need parentheses around `2.25-1/4`. You also need to use parentheses if the number being raised to some power is a calculation, such as the expression 2^{2+1} in the third example. Note that R considers a negative number a calculation because it interprets, for example, `-2` as `-1*2`. This is why you also need the parentheses around `-2` in that same expression. It's important to highlight these issues early because they can easily be overlooked in large chunks of code.

2.1.2 Logarithms and Exponentials

You'll often see or read about researchers performing a *log transformation* on certain data. This refers to rescaling numbers according to the *logarithm*. When supplied a given number x and a value referred to as a *base*, the logarithm calculates the power to which you must raise the base to get to x. For example, the logarithm of $x = 243$ to base 3 (written mathematically as $\log_3 243$) is 5, because $3^5 = 243$. In R, the log transformation is achieved with the `log` function. You supply `log` with the number to transform, assigned to the value `x`, and the base, assigned to `base`, as follows:

```
R> log(x=243,base=3)
[1] 5
```

Here are some things to consider:

- Both x and the base must be positive.
- The log of any number x when the base is equal to x is 1.
- The log of $x = 1$ is always 0, regardless of the base.

There's a particular kind of log transformation often used in mathematics called the *natural log*, which fixes the base at a special mathematical number—*Euler's number*. This is conventionally written as e and is approximately equal to 2.718.

Euler's number gives rise to the *exponential function*, defined as e raised to the power of x, where x can be any number (negative, zero, or positive). The exponential function, $f(x) = e^x$, is often written as $\exp(x)$ and represents the *inverse* of the natural log such that $\exp(\log_e x) = \log_e \exp(x) = x$. The R command for the exponential function is `exp`:

```
R> exp(x=3)
[1] 20.08554
```

The default behavior of `log` is to assume the natural log:

```
R> log(x=20.08554)
[1] 3
```

You must provide the value of `base` yourself if you want to use a value other than e. The logarithm and exponential functions are mentioned here because they become important later on in the book—many statistical methods use them because of their various helpful mathematical properties.

2.1.3 E-Notation

When R prints large or small numbers beyond a certain threshold of significant figures, set at 7 by default, the numbers are displayed using the classic scientific e-notation. The e-notation is typical to most programming languages—and even many desktop calculators—to allow easier interpretation of extreme values. In e-notation, any number x can be expressed as xey, which represents exactly $x \times 10^y$. Consider the number $2,342,151,012,900$. It could, for example, be represented as follows:

- 2.3421510129e12, which is equivalent to writing $2.3421510129 \times 10^{12}$
- 234.21510129e10, which is equivalent to writing $234.21510129 \times 10^{10}$

You could use any value for the power of y, but standard e-notation uses the power that places a decimal just after the first significant digit. Put simply, for a *positive* power $+y$, the e-notation can be interpreted as "move the decimal point y positions to the *right*." For a *negative* power $-y$, the interpretation is "move the decimal point y positions to the *left*." This is exactly how R presents e-notation:

```
R> 2342151012900
[1] 2.342151e+12
R> 0.0000002533
[1] 2.533e-07
```

In the first example, R shows only the first seven significant digits and hides the rest. Note that no information is lost in any calculations even if R hides digits; the e-notation is purely for ease of readability by the user, and the extra digits are still stored by R, even though they aren't shown.

Finally, note that R must impose constraints on how extreme a number can be before it is treated as either infinity (for large numbers) or zero (for small numbers). These constraints depend on your individual system, and I'll discuss the technical details a bit more in Section 6.1.1. However, any modern desktop system can be trusted to be precise enough by default for most computational and statistical endeavors in R.

Exercise 2.1

a. Using R, verify that

$$\frac{6a + 42}{3^{4.2-3.62}} = 29.50556$$

when $a = 2.3$.

b. Which of the following squares negative 4 and adds 2 to the result?
   i. `(-4)^2+2`
   ii. `-4^2+2`
   iii. `(-4)^(2+2)`
   iv. `-4^(2+2)`

c. Using R, how would you calculate the square root of half of the average of the numbers 25.2, 15, 16.44, 15.3, and 18.6?

d. Find $\log_e 0.3$.

e. Compute the exponential transform of your answer to (d).

f. Identify R's representation of −0.00000000423546322 when printing this number to the console.

2.2 Assigning Variables and Objects

So far, R has simply displayed the results of the example calculations by printing them to the console. If you want to save the results and perform further operations, you need to be able to *assign* the results of a given computation to an *object* in the current workspace. Put simply, this amounts to storing some item or result under a given name so it can be accessed later, without having to write out that calculation again. In this book, I will use the terms *assign* and *store* interchangeably. Note that some programming books refer to a stored object as a *variable* because of the ability to easily overwrite that object and change it to something different, meaning that what it represents can vary throughout a session. However, I'll use the term *object* throughout this book because we'll discuss variables in Part III as a distinctly different statistical concept.

You can specify an assignment in R in two ways: using arrow notation (`<-`) and using a single equal sign (`=`). Both methods are shown here:

```
R> x <- -5
R> x
[1] -5
R> x = x + 1  # this overwrites the previous value of x
R> x
[1] -4
```

```
R> mynumber = 45.2

R> y <- mynumber*x
R> y
[1] -180.8

R> ls()
[1] "mynumber" "x"        "y"
```

As you can see from these examples, R will display the value assigned to an object when you type the name of the object into the console. When you use the object in subsequent operations, R will substitute the value you assigned to it. Finally, if you use the `ls` command (which you saw in Section 1.3.1) to examine the contents of the current workspace, it will reveal the names of the objects in alphabetical order (along with any other previously created items).

Although `=` and `<-` do the same thing, it is wise (for the neatness of code if nothing else) to be consistent. Many users choose to stick with the `<-`, however, because of the potential for confusion in using the `=` (for example, I clearly didn't mean that x is *mathematically* equal to $x + 1$ earlier). In this book, I'll do the same and reserve `=` for setting function arguments, which begins in Section 2.3.2. So far you've used only numeric values, but note that the procedure for assignment is universal for all types and classes of objects, which you'll examine in the coming chapters.

Objects can be named almost anything as long as the name begins with a letter (in other words, not a number), avoids symbols (though underscores and periods are fine), and avoids the handful of "reserved" words such as those used for defining special values (see Section 6.1) or for controlling code flow (see Chapter 10). You can find a useful summary of these naming rules in Section 9.1.3.

Exercise 2.2

a. Create an object that stores the value $3^2 \times 4^{1/8}$.

b. Overwrite your object in (a) by itself divided by 2.33. Print the result to the console.

c. Create a new object with the value -8.2×10^{-13}.

d. Print directly to the console the result of multiplying (b) by (c).

2.3 Vectors

Often you'll want to perform the same calculations or comparisons upon multiple entities, for example if you're rescaling measurements in a data set. You could do this type of operation one entry at a time, though this is clearly not ideal, especially if you have a large number of items. R provides a far more efficient solution to this problem with *vectors*.

For the moment, to keep things simple, you'll continue to work with numeric entries only, though many of the utility functions discussed here may also be applied to structures containing non-numeric values. You'll start looking at these other kinds of data in Chapter 4.

2.3.1 Creating a Vector

The vector is the essential building block for handling multiple items in R. In a numerical sense, you can think of a vector as a collection of observations or measurements concerning a single variable, for example, the heights of 50 people or the number of coffees you drink daily. More complicated data structures may consist of several vectors. The function for creating a vector is the single letter `c`, with the desired entries in parentheses separated by commas.

```
R> myvec <- c(1,3,1,42)
R> myvec
[1]  1  3  1 42
```

Vector entries can be calculations or previously stored items (including vectors themselves).

```
R> foo <- 32.1
R> myvec2 <- c(3,-3,2,3.45,1e+03,64^0.5,2+(3-1.1)/9.44,foo)
R> myvec2
[1]    3.000000   -3.000000    2.000000    3.450000 1000.000000    8.000000
[7]    2.201271   32.100000
```

This code created a new vector assigned to the object `myvec2`. Some of the entries are defined as arithmetic expressions, and it's the result of the expression that's stored in the vector. The last element, `foo`, is an existing numeric object defined as `32.1`.

Let's look at another example.

```
R> myvec3 <- c(myvec,myvec2)
R> myvec3
 [1]    1.000000    3.000000    1.000000   42.000000    3.000000   -3.000000
 [7]    2.000000    3.450000 1000.000000    8.000000    2.201271   32.100000
```

This code creates and stores yet another vector, `myvec3`, which contains the entries of `myvec` and `myvec2` appended together in that order.

2.3.2 Sequences, Repetition, Sorting, and Lengths

Here I'll discuss some common and useful functions associated with R vectors: `seq`, `rep`, `sort`, and `length`.

Let's create an equally spaced sequence of increasing or decreasing numeric values. This is something you'll need often, for example when programming loops (see Chapter 10) or when plotting data points (see Chapter 7). The easiest way to create such a sequence, with numeric values separated by intervals of 1, is to use the colon operator.

```
R> 3:27
 [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

The example `3:27` should be read as "*from* 3 *to* 27 (*by* 1)." The result is a numeric vector just as if you had listed each number manually in parentheses with `c`. As always, you can also provide either a previously stored value or a (strictly parenthesized) calculation when using the colon operator:

```
R> foo <- 5.3
R> bar <- foo:(-47+1.5)
R> bar
 [1]   5.3   4.3   3.3   2.3   1.3   0.3  -0.7  -1.7  -2.7  -3.7  -4.7
[12]  -5.7  -6.7  -7.7  -8.7  -9.7 -10.7 -11.7 -12.7 -13.7 -14.7 -15.7
[23] -16.7 -17.7 -18.7 -19.7 -20.7 -21.7 -22.7 -23.7 -24.7 -25.7 -26.7
[34] -27.7 -28.7 -29.7 -30.7 -31.7 -32.7 -33.7 -34.7 -35.7 -36.7 -37.7
[45] -38.7 -39.7 -40.7 -41.7 -42.7 -43.7 -44.7
```

Sequences with seq

You can also use the `seq` command, which allows for more flexible creations of sequences. This ready-to-use function takes in a `from` value, a `to` value, and a `by` value, and it returns the corresponding sequence as a numeric vector.

```
R> seq(from=3,to=27,by=3)
[1]  3  6  9 12 15 18 21 24 27
```

This gives you a sequence with intervals of 3 rather than 1. Note that these kinds of sequences will always start at the `from` number but will not always include the `to` number, depending on what you are asking R to increase (or decrease) them `by`. For example, if you are increasing (or decreasing) by even numbers and your sequence ends in an odd number, the final number won't be included. Instead of providing a `by` value, however, you can specify a `length.out` value to produce a vector with that many numbers, evenly spaced between the `from` and `to` values.

```
R> seq(from=3,to=27,length.out=40)
 [1]  3.000000  3.615385  4.230769  4.846154  5.461538  6.076923  6.692308
 [8]  7.307692  7.923077  8.538462  9.153846  9.769231 10.384615 11.000000
[15] 11.615385 12.230769 12.846154 13.461538 14.076923 14.692308 15.307692
```

```
[22] 15.923077 16.538462 17.153846 17.769231 18.384615 19.000000 19.615385
[29] 20.230769 20.846154 21.461538 22.076923 22.692308 23.307692 23.923077
[36] 24.538462 25.153846 25.769231 26.384615 27.000000
```

By setting `length.out` to `40`, you make the program print exactly 40 evenly spaced numbers from 3 to 27.

For decreasing sequences, the use of `by` must be negative. Here's an example:

```
R> foo <- 5.3
R> myseq <- seq(from=foo,to=(-47+1.5),by=-2.4)
R> myseq
 [1]   5.3   2.9   0.5  -1.9  -4.3  -6.7  -9.1 -11.5 -13.9 -16.3 -18.7 -21.1
[13] -23.5 -25.9 -28.3 -30.7 -33.1 -35.5 -37.9 -40.3 -42.7 -45.1
```

This code uses the previously stored object `foo` as the value for `from` and uses the parenthesized calculation `(-47+1.5)` as the `to` value. Given those values (that is, with `foo` being greater than `(-47+1.5)`), the sequence can progress only in negative steps; directly above, we set `by` to be `-2.4`. The use of `length.out` to create decreasing sequences, however, remains the same (it would make no sense to specify a "negative length"). For the same `from` and `to` values, you can create a decreasing sequence of length 5 easily, as shown here:

```
R> myseq2 <- seq(from=foo,to=(-47+1.5),length.out=5)
R> myseq2
[1]   5.3  -7.4 -20.1 -32.8 -45.5
```

There are shorthand ways of calling these functions, which you'll learn about in Chapter 9, but in these early stages I'll stick with the explicit usage.

Repetition with rep

Sequences are extremely useful, but sometimes you may want simply to repeat a certain value. You do this using `rep`.

```
R> rep(x=1,times=4)
[1] 1 1 1 1
R> rep(x=c(3,62,8.3),times=3)
[1]  3.0 62.0  8.3  3.0 62.0  8.3  3.0 62.0  8.3
R> rep(x=c(3,62,8.3),each=2)
[1]  3.0  3.0 62.0 62.0  8.3  8.3
R> rep(x=c(3,62,8.3),times=3,each=2)
 [1]  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0
[16] 62.0  8.3  8.3
```

The `rep` function is given a single value or a vector of values as its argument `x`, as well as a value for the arguments `times` and `each`. The value for `times` provides the number of times to repeat `x`, and `each` provides the

number of times to repeat each element of x. In the first line directly above, you simply repeat a single value four times. The other examples first use rep and times on a vector to repeat the entire vector, then use each to repeat each member of the vector, and finally use both times and each to do both at once.

If neither times nor each is specified, R's default is to treat the values of times and each as 1 so that a call of rep(x=c(3,62,8.3)) will just return the originally supplied x with no changes.

As with seq, you can include the result of rep in a vector of the same data type, as shown in the following example:

```
R> foo <- 4
R> c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
[1]  3.00  8.30 32.00 32.00 32.00 32.00 -2.00 -1.25 -0.50  0.25  1.00
```

Here, I've constructed a vector where the third to sixth entries (inclusive) are governed by the evaluation of a rep command—the single value 32 repeated foo times (where foo is stored as 4). The last five entries are the result of an evaluation of seq, namely a sequence from −2 to 1 of length foo+1 (5).

Sorting with sort

Sorting a vector in increasing or decreasing order of its elements is another simple operation that crops up in everyday tasks. The conveniently named sort function does just that.

```
R> sort(x=c(2.5,-1,-10,3.44),decreasing=FALSE)
[1] -10.00  -1.00   2.50   3.44

R> sort(x=c(2.5,-1,-10,3.44),decreasing=TRUE)
[1]   3.44   2.50  -1.00 -10.00

R> foo <- seq(from=4.3,to=5.5,length.out=8)
R> foo
[1] 4.300000 4.471429 4.642857 4.814286 4.985714 5.157143 5.328571 5.500000
R> bar <- sort(x=foo,decreasing=TRUE)
R> bar
[1] 5.500000 5.328571 5.157143 4.985714 4.814286 4.642857 4.471429 4.300000

R> sort(x=c(foo,bar),decreasing=FALSE)
 [1] 4.300000 4.300000 4.471429 4.471429 4.642857 4.642857 4.814286 4.814286
 [9] 4.985714 4.985714 5.157143 5.157143 5.328571 5.328571 5.500000 5.500000
```

The sort function is pretty straightforward. You supply a vector to the function as the argument x, and a second argument, decreasing, indicates the order in which you want to sort. This argument takes a type of value you have not yet met: one of the all-important *logical* values. A logical value can be only one of two specific, case-sensitive values: TRUE or FALSE. Generally

speaking, logicals are used to indicate the satisfaction or failure of a certain *condition*, and they form an integral part of all programming languages. You'll investigate logical values in R in greater detail in Section 4.1. For now, in regards to `sort`, you set `decreasing=FALSE` to sort from smallest to largest, and `decreasing=TRUE` sorts from largest to smallest.

Finding a Vector Length with length

I'll round off this section with the `length` function, which determines how many entries exist in a vector given as the argument `x`.

```
R> length(x=c(3,2,8,1))
[1] 4

R> length(x=5:13)
[1] 9

R> foo <- 4
R> bar <- c(3,8.3,rep(x=32,times=foo),seq(from=-2,to=1,length.out=foo+1))
R> length(x=bar)
[1] 11
```

Note that if you include entries that depend on the evaluation of other functions (in this case, calls to `rep` and `seq`), `length` tells you the number of entries *after* those inner functions have been executed.

Exercise 2.3

a. Create and store a sequence of values from 5 to −11 that progresses in steps of 0.3.

b. Overwrite the object from (a) using the same sequence with the order reversed.

c. Repeat the vector `c(-1,3,-5,7,-9)` twice, with each element repeated 10 times, and store the result. Display the result sorted from largest to smallest.

d. Create and store a vector that contains, in any configuration, the following:
   i. A sequence of integers from 6 to 12 (inclusive)
   ii. A threefold repetition of the value 5.3
   iii. The number −3
   iv. A sequence of nine values starting at 102 and ending at the number that is the total length of the vector created in (c)

e. Confirm that the length of the vector created in (d) is of length 20.

2.3.3 Subsetting and Element Extraction

In all the results you have seen printed to the console screen so far, you may have noticed a curious feature. Immediately to the left of the output there is a square-bracketed `[1]`. When the output is a long vector that spans the width of the console and wraps onto the following line, another square-bracketed number appears to the left of the new line. These numbers represent the *index* of the entry directly to the right. Quite simply, the index corresponds to the *position* of a value within a vector, and that's precisely why the first value always has a `[1]` next to it (even if it's the only value and not part of a larger vector).

These indexes allow you to retrieve specific elements from a vector, which is known as *subsetting*. Suppose you have a vector called `myvec` in your workspace. Then there will be exactly `length(x=myvec)` entries in `myvec`, with each entry having a specific *position*: `1` or `2` or `3`, all the way up to `length(x=myvec)`. You can access individual elements by asking R to return the values of `myvec` at specific locations, done by entering the name of the vector followed by the position in square brackets.

```
R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
R> length(x=myvec)
[1] 10
R> myvec[1]
[1] 5

R> foo <- myvec[2]
R> foo
[1] -2.3

R> myvec[length(x=myvec)]
[1] -8
```

Because `length(x=myvec)` results in the final index of the vector (in this case, `10`), entering this phrase in the square brackets extracts the final element, `-8`. Similarly, you could extract the second-to-last element by subtracting 1 from the length; let's try that, and also assign the result to a new object:

```
R> myvec.len <- length(x=myvec)
R> bar <- myvec[myvec.len-1]
R> bar
[1] 40221
```

As these examples show, the index may be an arithmetic function of other numbers or previously stored values. You can assign the result to a new object in your workspace in the usual way with the `<-` notation. Using your knowledge of sequences, you can use the colon notation with the length of the specific vector to obtain all possible indexes for extracting a particular element in the vector.

```
R> 1:myvec.len
[1]  1  2  3  4  5  6  7  8  9 10
```

You can also delete individual elements by using *negative* versions of the indexes supplied in the square brackets. Continuing with the objects `myvec`, `foo`, `bar`, and `myvec.len` as defined earlier, consider the following operations:

```
R> myvec[-1]
[1]    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0    -8.0
```

This line produces the contents of `myvec` without the first element. Similarly, the following code assigns to the object `baz` the contents of `myvec` without its second element:

```
R> baz <- myvec[-2]
R> baz
[1]     5     4     4     4     6     8    10 40221    -8
```

Again, the index in the square brackets can be the result of an appropriate calculation, like so:

```
R> qux <- myvec[-(myvec.len-1)]
R> qux
[1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0 -8.0
```

Using the square-bracket operator to extract or delete values from a vector does not change the original vector you are subsetting *unless* you explicitly overwrite the vector with the subsetted version. For example, in the previous line of code, `qux` is a new vector defined as `myvec` without its second-to-last entry, but in your workspace, `myvec` itself *remains unchanged*. In other words, subsetting vectors in this way simply returns the requested elements, which can be assigned to a new object if you want, but doesn't alter the original object in the workspace.

Now, suppose you want to piece `myvec` back together from `qux` and `bar`. You can call something like this:

```
R> c(qux[-length(x=qux)],bar,qux[length(x=qux)])
 [1]     5.0    -2.3     4.0     4.0     4.0     6.0     8.0    10.0 40221.0
[10]    -8.0
```

As you can see, this line uses `c` to reconstruct the vector in three parts: `qux[-length(x=qux)]`, the object `bar` defined earlier, and `qux[length(x=qux)]`. For clarity, let's examine each part in turn.

- `qux[-length(x=qux)]`

  This piece of code returns the values of `qux` except for its last element.

```
R> length(x=qux)
[1] 9
R> qux[-length(x=qux)]
[1]  5.0 -2.3  4.0  4.0  4.0  6.0  8.0 10.0
```

  Now you have a vector that's the same as the first eight entries of `myvec`.

- `bar`

  Earlier, you had stored `bar` as the following:

```
R> bar <- myvec[myvec.len-1]
R> bar
[1] 40221
```

  This is precisely the second-to-last element of `myvec` that `qux` is missing. So, you'll slot this value in after `qux[-length(x=qux)]`.

- `qux[length(x=qux)]`

  Finally, you just need the last element of `qux` that matches the last element of `myvec`. This is extracted from `qux` (not deleted as earlier) using `length`.

```
R> qux[length(x=qux)]
[1] -8
```

Now it should be clear how calling these three parts of code together, in this order, is one way to reconstruct `myvec`.

As with most operations in R, you are not restricted to doing things one by one. You can also subset objects using *vectors of indexes*, rather than individual indexes. Using `myvec` again from earlier, you get the following:

```
R> myvec[c(1,3,5)]
[1] 5 4 4
```

This returns the first, third, and fifth elements of `myvec` in one go. Another common and convenient subsetting tool is the colon operator (discussed in Section 2.3.2), which creates a sequence of indexes. Here's an example:

```
R> 1:4
[1] 1 2 3 4
R> foo <- myvec[1:4]
R> foo
[1]  5.0 -2.3  4.0  4.0
```

This provides the first four elements of `myvec` (recall that the colon operator returns a numeric vector, so there is no need to explicitly wrap this using `c`).

The order of the returned elements depends entirely upon the index vector supplied in the square brackets. For example, using `foo` from the previous example, consider the order of the indexes and the resulting extractions, shown here:

```
R> length(x=foo):2
[1] 4 3 2
R> foo[length(foo):2]
[1]  4.0  4.0 -2.3
```

Here you extracted elements starting at the end of the vector, working backward. You can also use `rep` to repeat an index, as shown here:

```
R> indexes <- c(4,rep(x=2,times=3),1,1,2,3:1)
R> indexes
 [1] 4 2 2 2 1 1 2 3 2 1
R> foo[indexes]
 [1]  4.0 -2.3 -2.3 -2.3  5.0  5.0 -2.3  4.0 -2.3  5.0
```

This is now something a little more general than strictly "subsetting"—by using an index vector, you can create an entirely new vector of any length consisting of some or all of the elements in the original vector. As shown earlier, this index vector can contain the desired element positions in any order and can repeat indexes.

You can also return the elements of a vector after deleting more than one element. For example, to create a vector after removing the first and third elements of `foo`, you can execute the following:

```
R> foo[-c(1,3)]
[1] -2.3  4.0
```

Note that it is not possible to mix positive and negative indexes in a single index vector.

Sometimes you'll need to overwrite certain elements in an existing vector with a new value. In this situation, you first specify the elements you want to overwrite using square brackets and then use the assignment operator to assign the new values. Here's an example:

```
R> bar <- c(3,2,4,4,1,2,4,1,0,0,5)
R> bar
 [1] 3 2 4 4 1 2 4 1 0 0 5
R> bar[1] <- 6
R> bar
 [1] 6 2 4 4 1 2 4 1 0 0 5
```

This overwrites the first element of `bar`, which was originally `3`, with a new value, `6`. When selecting multiple elements, you can specify a single value to replace them all or enter a vector of values that's equal in length to the number of elements selected to replace them one for one. Let's try this with the same `bar` vector from earlier.

```
R> bar[c(2,4,6)] <- c(-2,-0.5,-1)
R> bar
 [1]  6.0 -2.0  4.0 -0.5  1.0 -1.0  4.0  1.0  0.0  0.0  5.0
```

Here you overwrite the second, fourth, and sixth elements with `-2`, `-0.5`, and `-1`, respectively; all else remains the same. By contrast, the following code overwrites elements `7` to `10` (inclusive), replacing them all with `100`:

```
R> bar[7:10] <- 100
R> bar
 [1]   6.0  -2.0   4.0  -0.5   1.0  -1.0 100.0 100.0 100.0 100.0   5.0
```

Finally, it's important to mention that this section has focused on just one of the two main methods, or "flavors," of vector element extraction in R. You'll look at the alternative method, using logical flags, in Section 4.1.5.

Exercise 2.4

a. Create and store a vector that contains the following, in this order:
   - A sequence of length 5 from 3 to 6 (inclusive)
   - A twofold repetition of the vector `c(2,-5.1,-3)`
   - The value $\frac{7}{42} + 2$

b. Extract the first and last elements of your vector from (a), storing them as a new object.

c. Store as a third object the values returned by omitting the first and last values of your vector from (a).

d. Use only (b) and (c) to reconstruct (a).

e. Overwrite (a) with the same values sorted from smallest to largest.

f. Use the colon operator as an index vector to reverse the order of (e), and confirm this is identical to using `sort` on (e) with `decreasing=TRUE`.

g. Create a vector from (c) that repeats the third element of (c) three times, the sixth element four times, and the last element once.

h. Create a new vector as a copy of (e) by assigning (e) as is to a newly named object. Using this new copy of (e), overwrite the first, the fifth to the seventh (inclusive), and the last element with the values 99 to 95 (inclusive), respectively.

2.3.4 Vector-Oriented Behavior

The reason vectors are so useful in R is the speed and efficiency with which the software can handle operations carried out on multiple elements at once. This *vector-oriented*, *vectorized*, or *element-wise* behavior is a key feature of the language, one that you will briefly examine here through some examples of rescaling measurements.

Let's start with this simple example:

```
R> foo <- 5.5:0.5
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo-c(2,4,6,8,10,12)
[1]   3.5   0.5  -2.5  -5.5  -8.5 -11.5
```

This code creates a sequence of six values between 5.5 and 0.5, separated by 1. From this vector, you subtract another vector containing 2, 4, 6, 8, 10, and 12. What does this do? Well, quite simply, R matches up the elements according to their respective positions and performs the operation on each corresponding pair of elements. In the previous example, the resulting vector is obtained by subtracting the first element of `c(2,4,6,8,10,12)` from the first element of `foo` ($5.5 - 2 = 3.5$), then by subtracting the second element of `c(2,4,6,8,10,12)` from the second element of `foo` ($4.5 - 4 = 0.5$), and so on. Thus, rather than inelegantly cycling through each element in turn (as you could do by hand or by explicitly using a loop), R permits a fast and efficient alternative using vector-oriented behavior. Figure 2-1 illustrates how you can understand this type of calculation and highlights the fact that the positions of the elements are crucial in terms of the final result; elements in differing positions have no effect on one another.

The situation is made more complicated when using vectors of different lengths, which can happen in two distinct ways. The first is when the length of the longer vector can be evenly divided by the length of the shorter vector. The second is when the length of the longer vector *cannot* be divided by the length of the shorter vector—this is usually unintentional on the user's part. In both of these situations, R essentially attempts to replicate, or *recycle*, the shorter vector by as many times as needed to match the length of the longer vector, before completing the specified operation. As an example, suppose you wanted to alternate the entries of `foo` shown earlier as negative and positive. You could explicitly multiply `foo` by `c(1,-1,1,-1,1,-1)`, but

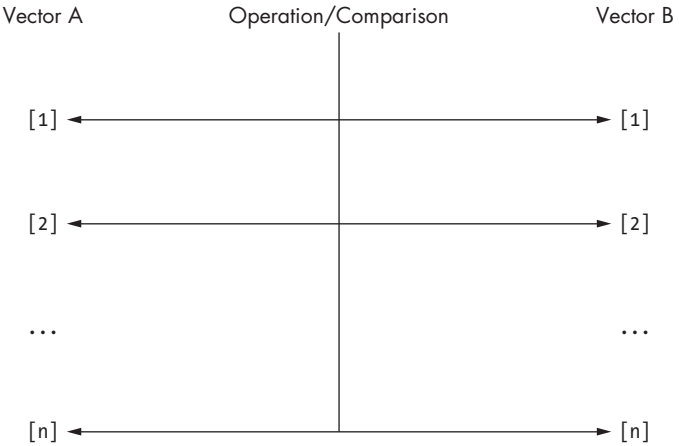


Figure 2-1: A conceptual diagram of the element-wise behavior of a comparison or operation carried out on two vectors of equal length in R. Note that the operation is performed by matching up the element positions.

you don't need to write out the full latter vector. Instead, you can write the following:

```
R> bar <- c(1,-1)
R> foo*bar
[1]  5.5 -4.5  3.5 -2.5  1.5 -0.5
```

Here `bar` has been applied repeatedly throughout the length of `foo` until completion. The left plot of Figure 2-2 illustrates this particular example. Now let's see what happens when the vector lengths are not evenly divisible.

```
R> baz <- c(1,-1,0.5,-0.5)
R> foo*baz
[1]  5.50 -4.50  1.75 -1.25  1.50 -0.50
Warning message:
In foo * baz :
  longer object length is not a multiple of shorter object length
```

Here you see that R has matched the first four elements of `foo` with the entirety of `baz`, but it's not able to fully repeat the vector again. The repetition has been attempted, with the first two elements of `baz` being matched with the last two of the longer `foo`, though not without a protest from R, which notifies the user of the unevenly divisible lengths (you'll look at warnings in more detail in Section 12.1). The plot on the right in Figure 2-2 illustrates this example.

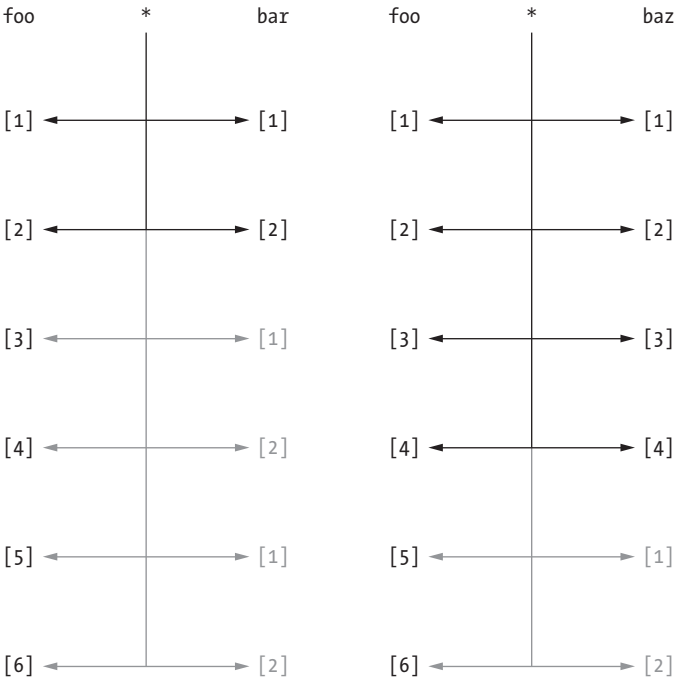


Figure 2-2: An illustration of the examples in the text, where two vectors of differing lengths are used in an element-wise operation. Left: `foo`*multiplied by* `bar`*; lengths are evenly divisible. Right:* `foo` *multiplied by* `baz`*; lengths are not evenly divisible, and a warning is issued.*

As I noted in Section 2.3.3, you can consider single values to be vectors of length 1, so you can use a single value to repeat an operation on all the values of a vector of any length. Here's an example, using the same vector `foo`:

```
R> qux <- 3
R> foo+qux
[1] 8.5 7.5 6.5 5.5 4.5 3.5
```

This is far easier than executing `foo+c(3,3,3,3,3,3)` or the more general `foo+rep(x=3,times=length(x=foo))`. Operating on vectors using a single value in this fashion is quite common, such as if you want to rescale or translate a set of measurements by some constant amount.

Another benefit of vector-oriented behavior is that you can use vectorized functions to complete potentially laborious tasks. For example, if you want to sum or multiply all the entries in a numeric vector, you can just use a built-in function.

Recall `foo`, shown earlier:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
```

You can find the sum of these six elements with

```
R> sum(foo)
[1] 18
```

and their product with

```
R> prod(foo)
[1] 162.4219
```

Far from being just convenient, vectorized functions are faster and more efficient than an explicitly coded iterative approach like a loop. The main takeaway from these examples is that much of R's functionality is designed specifically for certain data structures, ensuring neatness of code as well as optimization of performance.

Lastly, as mentioned earlier, this vector-oriented behavior applies in the same way to overwriting multiple elements. Again using `foo`, examine the following:

```
R> foo
[1] 5.5 4.5 3.5 2.5 1.5 0.5
R> foo[c(1,3,5,6)] <- c(-99,99)
R> foo
[1] -99.0   4.5  99.0   2.5 -99.0  99.0
```

You see four specific elements being overwritten by a vector of length 2, which is recycled in the same fashion you're familiar with. Again, the length of the vector of replacements must evenly divide the number of elements being overwritten, or else a warning similar to the one shown earlier will be issued when R cannot complete a full-length recycle.

Exercise 2.5

a. Convert the vector `c(2,0.5,1,2,0.5,1,2,0.5,1)` to a vector of only `1`s, using a vector of length 3.

b. The conversion from a temperature measurement in degrees Fahrenheit F to Celsius C is performed using the following equation:

$$C = \frac{5}{9}(F - 32)$$

Use vector-oriented behavior in R to convert the temperatures 45, 77, 20, 19, 101, 120, and 212 in degrees Fahrenheit to degrees Celsius.

c. Use the vector `c(2,4,6)` and the vector `c(1,2)` in conjunction with `rep` and `*` to produce the vector `c(2,4,6,4,8,12)`.

d. Overwrite the middle four elements of the resulting vector from (c) with the two recycled values `-0.1` and `-100`, in that order.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`+, *, -, /, ^`	Arithmetic	Section 2.1, p. 15
`sqrt`	Square root	Section 2.1.1, p. 16
`log`	Logarithm	Section 2.1.2, p. 17
`exp`	Exponential	Section 2.1.2, p. 17
`<-, =`	Object assignment	Section 2.2, p. 19
`c`	Vector creation	Section 2.3.1, p. 21
`:, seq`	Sequence creation	Section 2.3.2, p. 22
`rep`	Value/vector repetition	Section 2.3.2, p. 23
`sort`	Vector sorting	Section 2.3.2, p. 24
`length`	Determine vector length	Section 2.3.2, p. 25
`[ ]`	Vector subsetting/extraction	Section 2.3.3, p. 26
`sum`	Sum all vector elements	Section 2.3.4, p. 34
`prod`	Multiply all vector elements	Section 2.3.4, p. 34

3

MATRICES

By now, you have a solid handle on using vectors in R. A *matrix* is simply several vectors of equal length (and data type) stored together. Whereas the size of a vector is exclusively described by its length, the size of a matrix is specified by a number of rows and a number of columns. You can also create higher-dimensional structures that are referred to as *arrays*. In this chapter, we'll begin by scrutinizing matrices before increasing the dimension to form arrays.

3.1 Defining a Matrix

The matrix is an important mathematical construct, and it's essential to many statistical methods, particularly when you're dealing with multivariate observations (see Chapter 13). You will typically describe a matrix A as an $m \times n$ matrix; that is, A will have exactly m rows and n columns. This means

A will have a total of mn entries, with each entry $a_{i,j}$ having a unique position given by its specific row ($i = 1,2,\ldots,m$) and column ($j = 1,2,\ldots,n$). You can therefore express a matrix as follows:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

To create a matrix in R, use the aptly named `matrix` command.

```
R> A <- matrix(data=c(-3,2,893,0.17),nrow=2,ncol=2)
R> A
     [,1]   [,2]
[1,]   -3 893.00
[2,]    2   0.17
```

The entries of the matrix are provided as a vector to the `data` argument, and you must make sure that the length of this vector matches exactly with the number of desired rows (`nrow`) and columns (`ncol`). You can elect not to supply `nrow` and `ncol` when calling `matrix`, in which case R's default behavior is to return a single-column matrix of the entries in `data`. For example, `matrix(data=c(-3,2,893,0.17))` is identical to `matrix(data=c(-3,2,893,0.17),nrow=4,ncol=1)`.

3.1.1 Filling Direction

It's important to be aware of how R fills up the matrix using the entries from `data`. Looking at the previous example, you can see that the 2×2 matrix `A` has been filled in a *column-by-column* fashion when reading the `data` entries from left to right. You can control how R fills in data using the argument `byrow`, as shown in the following examples:

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=FALSE)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Here, I have instructed R to provide a 2×3 matrix containing the digits 1 through 6. By using the optional argument `byrow` set to `FALSE`, you explicitly tell R to fill this 2×3 structure in a column-wise fashion, reading the `data` argument vector from left to right. This is R's default handling of the `matrix` function, so if the `byrow` argument is not supplied, the software will assume `byrow=FALSE`. Figure 3-1 illustrates this behavior.

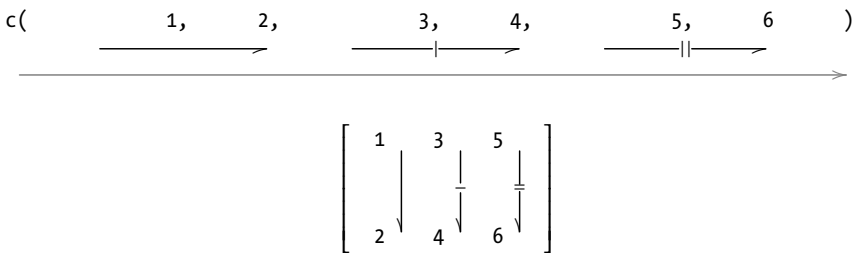


Figure 3-1: Filling a 2 × 3 matrix in a column-wise fashion with `byrow=FALSE` *(R default)*

Now, let's repeat the same line of code but set `byrow=TRUE`.

```
R> matrix(data=c(1,2,3,4,5,6),nrow=2,ncol=3,byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

The resulting 2 × 3 structure has now been filled in a row-wise fashion, as shown in Figure 3-2.

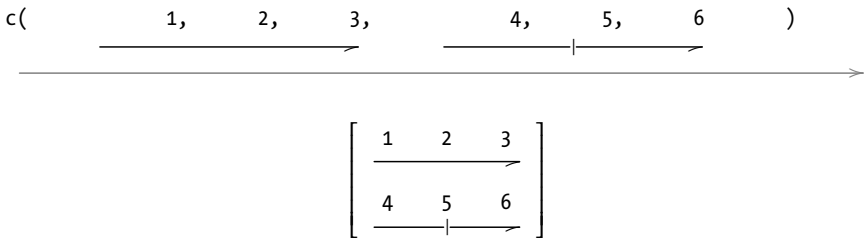


Figure 3-2: Filling a 2 × 3 matrix in a row-wise fashion with `byrow=TRUE`

3.1.2 Row and Column Bindings

If you have multiple vectors of equal length, you can quickly build a matrix by binding together these vectors using the built-in R functions, `rbind` and `cbind`. You can either treat each vector as a row (by using the command `rbind`) or treat each vector as a column (using the command `cbind`). Say you have the two vectors `1:3` and `4:6`. You can reconstruct the 2 × 3 matrix in Figure 3-2 using `rbind` as follows:

```
R> rbind(1:3,4:6)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Here, `rbind` has bound together the vectors as two rows of a matrix, with the top-to-bottom order of the rows matching the order of the vectors supplied to `rbind`. The same matrix could be constructed as follows, using `cbind`:

```
R> cbind(c(1,4),c(2,5),c(3,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Here, you have three vectors each of length 2. You use `cbind` to glue together these three vectors in the order they were supplied, and each vector becomes a column of the resulting matrix.

Another useful function, `dim`, provides the dimensions of a matrix stored in your workspace.

```
R> mymat <- rbind(c(1,3,4),5:3,c(100,20,90),11:13)
R> mymat
     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    5    4    3
[3,]  100   20   90
[4,]   11   12   13

R> dim(mymat)
[1] 4 3
R> nrow(mymat)
[1] 4
R> ncol(mymat)
[1] 3
R> dim(mymat)[2]
[1] 3
```

Having defined a matrix `mymat` using `rbind`, you can confirm its dimensions with `dim`, which returns a vector of length 2; `dim` always supplies the number of rows first, followed by the number of columns. Earlier, you also used two related functions: `nrow` (which provides the number of rows only) and `ncol` (which provides the number of columns only). In the last command shown previously, you use `dim` and your knowledge of vector subsetting to extract the same result that `ncol` would give you.

3.2 Subsetting

Extracting and subsetting elements from matrices in R is much like extracting elements from vectors. The only complication is that you now have an additional dimension. Element extraction still uses the square-bracket operator, but now it must be performed with both a row *and* a column position, given strictly in the order of [*row*,*column*]. Let's start by creating a 3×3 matrix, which I'll use for the examples in this section.

```
R> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

To tell R to "look at the third row of `A` and give me the element from the second column," you execute the following:

```
R> A[3,2]
[1] 105.5
```

As expected, you're given the element at position `[3,2]`.

3.2.1 Row, Column, and Diagonal Extractions

To extract an entire row or column from a matrix, you simply specify the desired row or column number and leave the other value blank. It's important to note that the comma that separates the row and column numbers *must remain in place*—this is how R distinguishes between a request for a row and a request for a column. The following returns the second column of `A`:

```
R> A[,2]
[1]  91.0   0.1 105.5
```

The following examines the first row:

```
R> A[1,]
[1]  0.3 91.0 -4.2
```

Note that whenever an extraction (or deletion, covered in a moment) results in a single value, single row, or single column, R will always return stand-alone vectors comprised of the requested values. You can also perform more complicated extractions, where the result cannot be expressed as a vector but must be returned as a new matrix of the appropriate dimensions. Consider the following subsets:

```
R> A[2:3,]
     [,1]  [,2] [,3]
[1,]  4.5   0.1  8.2
[2,] 55.3 105.5 27.9

R> A[,c(3,1)]
     [,1] [,2]
[1,] -4.2  0.3
[2,]  8.2  4.5
[3,] 27.9 55.3
```

```
R> A[c(3,1),2:3]
      [,1] [,2]
[1,] 105.5 27.9
[2,]  91.0 -4.2
```

The first command returns the second and third columns of `A`, and the second command returns the third and first columns of `A`. The last command accesses the third and first rows of `A`, in that order, and from those rows it returns the second and third column elements.

You can also identify the values along the diagonal of a square matrix (a matrix with an equal number of rows and columns) using the `diag` command.

```
R> diag(x=A)
[1]  0.3  0.1 27.9
```

This returns a vector with the elements along the diagonal of `A`, starting at `A[1,1]`.

3.2.2 Omitting and Overwriting

You can delete or omit elements from a matrix using negative indexes in square brackets. The following provides `A` without its second column:

```
R> A[,-2]
     [,1] [,2]
[1,]  0.3 -4.2
[2,]  4.5  8.2
[3,] 55.3 27.9
```

The following removes the first row from `A` and retrieves the third and second column values, in that order, from the remaining two rows:

```
R> A[-1,3:2]
     [,1]  [,2]
[1,]  8.2   0.1
[2,] 27.9 105.5
```

The following produces `A` without its first row and second column:

```
R> A[-1,-2]
     [,1] [,2]
[1,]  4.5  8.2
[2,] 55.3 27.9
```

Lastly, this deletes the first row and then deletes the second and third columns from the result:

```
R> A[-1,-c(2,3)]
[1]  4.5 55.3
```

Note that this final operation leaves you with the last two elements of the first column of A, which is expressible as a stand-alone vector and is therefore returned as such.

To overwrite particular elements, or entire rows or columns, you can take the same approach as with vectors in Section 2.3.3. You identify the elements to be replaced and then assign the new values. The new elements can be a single value, a vector of the same length as the number of elements to be replaced, or a vector whose length evenly divides the number of elements to be replaced. To illustrate this, let's first create a copy of A and call it B.

```
R> B <- A
R> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

The following overwrites the second row of B with the sequence 1, 2, and 3:

```
R> B[2,] <- 1:3
R> B
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  1.0   2.0  3.0
[3,] 55.3 105.5 27.9
```

The following overwrites the second column elements of the first and third rows with 900:

```
R> B[c(1,3),2] <- 900
R> B
     [,1] [,2] [,3]
[1,]  0.3  900 -4.2
[2,]  1.0    2  3.0
[3,] 55.3  900 27.9
```

Next, you replace the third column of B with the values in the third *row* of B.

```
R> B[,3] <- B[3,]
R> B
```

```
     [,1] [,2]  [,3]
[1,]  0.3  900  55.3
[2,]  1.0    2 900.0
[3,] 55.3  900  27.9
```

To try R's vector recycling, let's now overwrite the first and third column elements of rows 1 and 3 (a total of four elements) with the two values `-7` and `7`.

```
R> B[c(1,3),c(1,3)] <- c(-7,7)
R> B
     [,1] [,2] [,3]
[1,]   -7  900   -7
[2,]    1    2  900
[3,]    7  900    7
```

The vector of length 2 has replaced the four elements *in a column-wise fashion*. The previous command uses the replacement vector `c(-7,7)` to overwrite the elements at positions (1,1) and (3,1), in that order. Then the replacement vector is repeated to overwrite (1,3) and (3,3), in that order. To highlight the role of index order on matrix element replacement, consider the following example:

```
R> B[c(1,3),2:1] <- c(65,-65,88,-88)
R> B
     [,1] [,2] [,3]
[1,]   88   65   -7
[2,]    1    2  900
[3,]  -88  -65    7
```

The four values in the replacement vector have overwritten the four specified elements, again in a column-wise fashion. In this case, because I specified the first and second columns in reverse order, the overwriting proceeded accordingly, filling the second column before moving to the first. Position (1,2) is matched with `65`, followed by (3,2) with `-65`; then (1,1) becomes `88`, and (3,1) becomes `-88`.

It is also possible to avoid explicit indexes and directly overwrite the values on the diagonal of a square matrix using the `diag` command.

```
R> diag(x=B) <- rep(x=0,times=3)
R> B
     [,1] [,2] [,3]
[1,]    0   65   -7
[2,]    1    0  900
[3,]  -88  -65    0
```

Exercise 3.1

a. Construct and store a 4×2 matrix that is filled row-wise with the values 4.3, 3.1, 8.2, 8.2, 3.2, 0.9, 1.6, and 6.5, in that order.

b. Confirm the dimensions of the matrix from (a) are 3×2 if you remove any one row.

c. Overwrite the second column of the matrix from (a) with that same column sorted from smallest to largest.

d. What does R return if you delete the fourth row and the first column from (c)? Use `matrix` to ensure the result is a single-column matrix, rather than a vector.

e. Store the bottom four elements of (c) as a new 2×2 matrix.

f. Overwrite, in this order, the elements of (c) at positions $(4,2)$, $(1,2)$, $(4,1)$, and $(1,1)$ with $-\frac{1}{2}$ of the two values on the diagonal of (e).

3.3 Matrix Operations and Algebra

You can think of matrices in R from two perspectives. First, you can use these structures purely as a computational tool in programming to store and operate on results. Alternatively, you can use matrices for their mathematical properties in relevant calculations. This distinction is important when handling these data structures in the software because the mathematical behavior of matrices does not always match the way they are generically handled by the language. Here I will briefly describe some special matrices, as well as some of the most common mathematical operations involving matrices, and I will demonstrate the corresponding functionality in R. If the mathematical behavior of matrices isn't of interest to you, you can skip this section for now and refer to it later.

3.3.1 Matrix Transpose

For any $m \times n$ matrix A, its *transpose*, $A^\top$, is the $n \times m$ matrix obtained by writing its columns as rows or, equivalently, its rows as columns.

Here's an example:

$$\text{If } A = \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix}, \text{ then } A^\top = \begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix}.$$

In `R`, the transpose of a matrix is found with the function `t`.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> A
     [,1] [,2] [,3]
```

```
[1,]    2    5    2
[2,]    6    1    4
R> t(A)
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
```

Note that taking the "transpose of the transpose" of *A* will recover the original matrix.

```
R> t(t(A))
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
```

3.3.2 Identity Matrix

The *identity matrix* written as I_m is a square $m \times m$ matrix with ones on the diagonal and zeros elsewhere.

Here's an example:

$$\text{The } 3 \times 3 \text{ identity matrix } I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

You can create an identity matrix of any dimension using the standard `matrix` function, but there is a quicker approach using `diag`. Earlier, I used `diag` on an existing matrix to extract or overwrite its diagonal elements. You can also use it as follows:

```
R> A <- diag(x=3)
R> A
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Here you see `diag` can be used to easily produce an identity matrix. To clarify, the behavior of `diag` depends on what you supply to it as its argument `x`. If, as earlier, `x` is a matrix, `diag` will retrieve the diagonal elements of the matrix. If `x` is a single positive integer, as is the case here, then `diag` will produce the identity matrix of the corresponding dimension. You can find more uses of `diag` on its help page.

3.3.3 Scalar Multiple of a Matrix

Multiplication of any matrix A by a scalar (single, univariate) value a results in a matrix in which every individual element is multiplied by a.

Here's an example:

$$-2.5 \times \begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} = \begin{bmatrix} -5 & -12.5 & -5 \\ -15 & -2.5 & -10 \end{bmatrix}$$

The notion of every element in a matrix being multiplied by some constant amount fits right into R's element-wise behavior. Scalar multiplication of a matrix is carried out using the standard arithmetic * operator.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> a <- -2.5
R> a*A
     [,1]  [,2] [,3]
[1,]   -5 -12.5   -5
[2,]  -15  -2.5  -10
```

3.3.4 Matrix Addition and Subtraction

For any two matrices A and B of equal size, addition or subtraction is performed in an element-wise fashion, where corresponding elements are added or subtracted from one another, depending on the operation.

Here's an example:

$$\begin{bmatrix} 2 & 6 \\ 5 & 1 \\ 2 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 8.1 \\ 3 & 8.2 \\ 6 & -9.8 \end{bmatrix} = \begin{bmatrix} 4 & -2.1 \\ 2 & -7.2 \\ -4 & 13.8 \end{bmatrix}$$

Again, the standard operational mode of R is to proceed in an element-wise fashion, which means you can add or subtract any two equally sized matrices with the standard + and - symbols.

```
R> A <- cbind(c(2,5,2),c(6,1,4))
R> A
     [,1] [,2]
[1,]    2    6
[2,]    5    1
[3,]    2    4
R> B <- cbind(c(-2,3,6),c(8.1,8.2,-9.8))
R> B
     [,1] [,2]
[1,]   -2  8.1
[2,]    3  8.2
[3,]    6 -9.8
R> A-B
```

```
     [,1] [,2]
[1,]    4 -2.1
[2,]    2 -7.2
[3,]   -4 13.8
```

3.3.5 Matrix Multiplication

Suppose you have two matrices A and B of size $m \times n$ and $p \times q$, respectively. You can perform *matrix multiplication* to compute their product $A \cdot B$ if and only if $n = p$. The resulting matrix will have the size $m \times q$. The elements of the product are computed in a row-by-column fashion, where the value at position $(AB)_{i,j}$ is computed by element-wise multiplication of the entries in row i of A by the entries in column j of B, summing the result.

Here's an example:

$$\begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & -3 \\ -1 & 1 \\ 1 & 5 \end{bmatrix}$$
$$= \begin{bmatrix} 2_{\times}3 + 5_{\times}(-1) + 2_{\times}1 & 2_{\times}(-3) + 5_{\times}(1) + 2_{\times}5 \\ 6_{\times}3 + 1_{\times}(-1) + 4_{\times}1 & 6_{\times}(-3) + 1_{\times}(1) + 4_{\times}5 \end{bmatrix}$$
$$= \begin{bmatrix} 3 & 9 \\ 21 & 3 \end{bmatrix}$$

Note that, in general, multiplication of appropriately sized matrices (denoted, say, with C and D) is not commutative; that is, $CD \neq DC$.

Unlike addition, subtraction, and scalar multiplication, matrix multiplication is not a simple element-wise calculation, and the standard $*$ operator cannot be used for this operation. Instead, you must use R's matrix product operator, written with percent symbols as `%*%`. Before you try this operator, let's first store the two matrices in the previous example and check to make sure the number of columns in the first matrix matches the number of rows in the second matrix using `dim`.

```
R> A <- rbind(c(2,5,2),c(6,1,4))
R> dim(A)
[1] 2 3
R> B <- cbind(c(3,-1,1),c(-3,1,5))
R> dim(B)
[1] 3 2
```

This confirms the two matrices are compatible to be multiplied by one another, so you can proceed.

```
R> A%*%B
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

You can show that matrix multiplication is noncommutative using the same two matrices. Switching the order of multiplication gives you an entirely different result.

```
R> B%*%A
     [,1] [,2] [,3]
[1,]  -12   12   -6
[2,]    4   -4    2
[3,]   32   10   22
```

3.3.6 Matrix Inversion

Some square matrices possess the property of *invertibility*. The inverse of such a matrix A is denoted A^{-1}. This matrix must satisfy the following equation:

$$AA^{-1} = I_m$$

Various computational approaches exist that are capable of computing the inverse of A, if it exists. Matrices that are not invertible are referred to as *singular*.

Here's an example:

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & -0.5 \\ -2 & 1.5 \end{bmatrix}$$

Inverting a matrix is often necessary when solving equations with matrices and has important practical ramifications. There are several different approaches to matrix inversion, and these calculations can become extremely computationally expensive as you increase the size of a matrix. We won't explore the mathematical intricacies of *spectral decomposition* of matrices here; you can consult authoritative references such as Golub & Van Loan (1989) for formal discussions.

The R function `solve` provides one option for inverting a matrix.

```
R> A <- matrix(data=c(3,4,1,2),nrow=2,ncol=2)
R> A
     [,1] [,2]
[1,]    3    1
[2,]    4    2
R> solve(A)
     [,1] [,2]
[1,]    1 -0.5
[2,]   -2  1.5
```

You can also verify that the product of these two matrices (using matrix multiplication rules) results in the 2×2 identity matrix.

```
R> A%*%solve(A)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Exercise 3.2

a. Calculate the following:

$$\frac{2}{7}\left(\begin{bmatrix}1 & 2\\ 2 & 4\\ 7 & 6\end{bmatrix} - \begin{bmatrix}10 & 20\\ 30 & 40\\ 50 & 60\end{bmatrix}\right)$$

b. Store these two matrices:

$$A = \begin{bmatrix}1\\ 2\\ 7\end{bmatrix} \qquad B = \begin{bmatrix}3\\ 4\\ 8\end{bmatrix}$$

Which of the following multiplications are possible? For those that are, compute the result.

i. $A \cdot B$
ii. $A^\top \cdot B$
iii. $B^\top \cdot (A \cdot A^\top)$
iv. $(A \cdot A^\top) \cdot B^\top$
v. $[(B \cdot B^\top) + (A \cdot A^\top) - 100I_3]^{-1}$

c. For

$$A = \begin{bmatrix}2 & 0 & 0 & 0\\ 0 & 3 & 0 & 0\\ 0 & 0 & 5 & 0\\ 0 & 0 & 0 & -1\end{bmatrix},$$

confirm that $A^{-1} \cdot A - I_4$ provides a 4×4 matrix of zeros.

3.4 Multidimensional Arrays

The progression from a vector (a "line" of elements) to a matrix (a "rectangle" of elements) is a natural result of the unit increase in dimension. You can continue to increase the dimension to obtain more complex data structures. In R, vectors and matrices can essentially be considered special cases of the more general *array*, which is how I'll refer to these types of structures when the dimension increases beyond 2.

So, what's the next step up from a matrix? Well, just as a matrix is considered to be a collection of vectors of equal length, a three-dimensional array can be considered to be a collection of equally dimensioned matrices,

providing you with a rectangular prism of elements. You still have a fixed number of rows and a fixed number of columns, as well as a new third dimension called a *layer*. Figure 3-3 illustrates a three-row, four-column, two-layer ($3 \times 4 \times 2$) array.

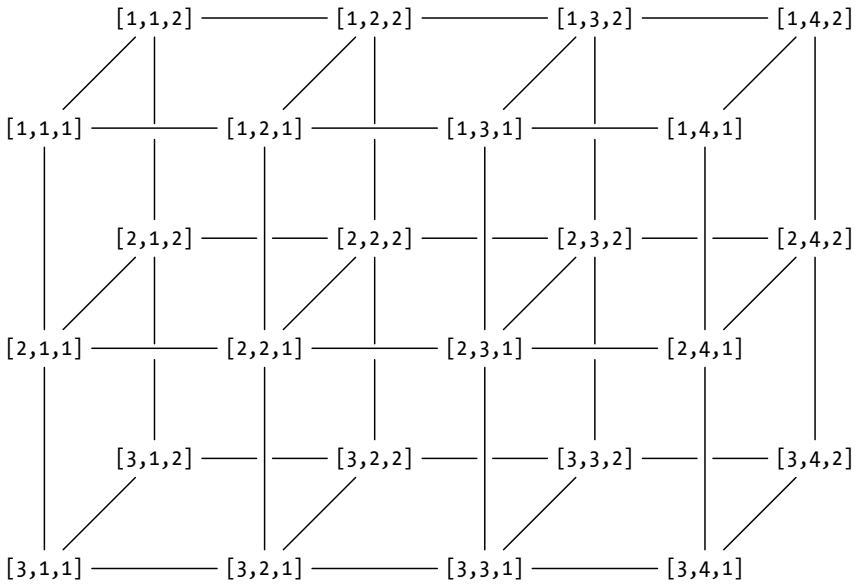


Figure 3-3: A conceptual diagram of a 3 × 4 × 2 array. The index of each element is given at the corresponding position. These indexes are provided in the strict order of `[row,column,layer]`.

3.4.1 Definition

To create these data structures in R, use the `array` function. Individual elements are specified in the `data` argument as a vector. The size is specified in the `dim` argument as another vector with a length corresponding to the number of dimensions. Note that `array` fills the entries of each layer with the elements in `data` in a strict column-wise fashion, starting with the first layer. Consider the following example:

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

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

, , 2
```

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

This gives you an array of the same size as in Figure 3-3—each of the two layers constitutes a 3×4 matrix. In this example, note the order of the dimensions supplied to `dim`: `c(`*`rows`*`,`*`columns`*`,`*`layers`*`)`. Just like a single matrix, the product of the dimension sizes of an array will yield the total number of elements. As you increase the dimension further, the `dim` vector must be extended accordingly. For example, a four-dimensional array is the next step up and can be thought of as *blocks* of three-dimensional arrays. Suppose you had a four-dimensional array comprised of three copies of the three-dimensional array in `AR`. This new array can be stored in R as follows (once again, the array is filled column-wise):

```
R> BR <- array(data=rep(1:24,times=3),dim=c(3,4,2,3))
R> BR
, , 1, 1

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

, , 2, 1

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

, , 1, 2

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

, , 2, 2

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

, , 1, 3
```

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

, , 2, 3

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

Indeed, with `BR` you now have three copies of `AR`. Each of these copies is split into its two layers so R can print the object to the screen. Here, the blocks are indexed by the fourth digit in the dimensional indexer. As before, the layers of each block are indexed by the third digit, the columns are indexed by the second digit, and the rows are indexed by the first digit.

3.4.2 Subsets, Extractions, and Replacements

Even though high-dimensional objects can be difficult to conceptualize, R's indexing remains consistent. This makes extracting elements from these structures straightforward given your knowledge of subsetting matrices—you just have to keep using commas in the square brackets as separators of the dimensions being accessed. This is highlighted in the examples that follow.

Suppose you want the second row of the second layer of the previously created array `AR`. You just enter these exact dimensional locations of `AR` in square brackets.

```
R> AR[2,,2]
[1] 14 17 20 23
```

The desired elements have been extracted as a vector of length 4. If you want specific elements from this vector, say the third and first, in that order, you can call the following:

```
R> AR[2,c(3,1),2]
[1] 20 14
```

Again, this literal method of subsetting makes dealing with even high-dimensional objects in R manageable.

An extraction that results in multiple vectors will be presented as columns in the returned matrix. For example, to extract the first rows of both layers of `AR`, you enter this:

```
R> AR[1,,]
     [,1] [,2]
```

```
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

The returned object has the first rows of each of the two matrix layers. However, it has returned each of these vectors as a *column* of the single returned matrix. As this example shows, when multiple vectors are extracted from an array, they will be returned as columns by default. This means extracted rows will not necessarily be returned as rows.

Turning to the object `BR`, it should be fairly clear that the following gives you the single element located in the second row and first column of the matrix in the first layer of the three-dimensional array located in the third block.

```
R> BR[2,1,1,3]
[1] 2
```

Again, you just need to look at the position of the index in the square brackets to interpret which values you are asking R to return from the array. The following examples highlight this:

```
R> BR[1,,,1]
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

This returns all the values in the first row of the first block. Since we left the column and layer indexes blank in this subset `[1,,,1]`, the previous command has returned values for all four columns and both layers in that block of `BR`.

Next, the following line returns all the values in the second layer of the array `BR`, composed of three matrices:

```
R> BR[,,2,]
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
```

```
[2,]   14   17   20   23
[3,]   15   18   21   24

, , 3

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

This last example highlights a feature noted earlier, where multiple vectors from `AR` were returned as a matrix. Broadly speaking, if you have an extraction that results in multiple d-dimensional arrays, the result will be an array of the next-highest dimension, $d + 1$. In the last example, we extracted multiple (two-dimensional) matrices, and they were returned as a three-dimensional array. This is demonstrated again in the next example:

```
R> BR[3:2,4,,]
, , 1

     [,1] [,2]
[1,]   12   24
[2,]   11   23

, , 2

     [,1] [,2]
[1,]   12   24
[2,]   11   23

, , 3

     [,1] [,2]
[1,]   12   24
[2,]   11   23
```

This extracts the elements at rows 3 and 2 (in that order), column 4, for all layers and for all array blocks. Consider the following final example:

```
R> BR[2,,1,]
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    5    5    5
[3,]    8    8    8
[4,]   11   11   11
```

Here we've asked R to return the entire second rows of the first layers of all the arrays stored in `BR`.

Deleting elements in high-dimensional arrays, as well as overwriting specified elements, follows the same rules as for stand-alone vectors and matrices. You specify the dimension positions the same way, using negative indexes (for deletion) or using the assignment operator for overwriting.

You can use the `array` function to create one-dimensional arrays (vectors) and two-dimensional arrays (matrices) should you want to (by setting the `dim` argument to 1 or 2, respectively). Note, though, that vectors in particular may be treated differently by some functions if created with `array` instead of `c` (see the help file for `array` for technical details). For this reason, as well as ease of readability in large sections of code, it is more conventional in R programming to use the specific vector- and matrix-creation functions `c` and `matrix`.

Exercise 3.3

a. Create and store a three-dimensional array with six layers of a 4×2 matrix, filled with a decreasing sequence of values between 4.8 and 0.1 of the appropriate length.

b. Extract and store as a new object the fourth- and first-row elements, in that order, of the second column only of all layers of (a).

c. Use a fourfold repetition of the second row of the matrix formed in (b) to fill a new array of dimensions $2 \times 2 \times 2 \times 3$.

d. Create a new array comprised of the results of deleting the sixth layer of (a).

e. Overwrite the second and fourth row elements of the second column of layers 1, 3, and 5 of (d) with −99.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`matrix`	Create a matrix	Section 3.1, p. 38
`rbind`	Create a matrix (bind rows)	Section 3.1.2, p. 39
`cbind`	Create a matrix (bind columns)	Section 3.1.2, p. 40
`dim`	Get matrix dimensions	Section 3.1.2, p. 40
`nrow`	Get number of rows	Section 3.1.2, p. 40
`ncol`	Get number of columns	Section 3.1.2, p. 40
`[ , ]`	Matrix/array subsetting	Section 3.2, p. 41
`diag`	Diagonal elements/identity matrix	Section 3.2.1, p. 42
`t`	Matrix transpose	Section 3.3.1, p. 45
`*`	Scalar matrix multiple	Section 3.3.3, p. 47
`+, -`	Matrix addition/subtraction	Section 3.3.4, p. 47
`%*%`	Matrix multiplication	Section 3.3.5, p. 48
`solve`	Matrix inversion	Section 3.3.6, p. 49
`array`	Create an array	Section 3.4.1, p. 51

4

NON-NUMERIC VALUES

So far, you've been working almost exclusively with numeric values. But statistical programming (and programming in general) also requires other, non-numeric values. In this chapter, we'll consider three important non-numeric data types: logicals, characters, and factors. These data types play an important role in effective use of R, especially as we get into more complex R programming in Part II.

4.1 Logical Values

Logical values (also simply called *logicals*) are based on a simple premise: a logical-valued object can only be either `TRUE` or `FALSE`. These can be interpreted as yes/no, one/zero, satisfied/not satisfied, and so on. This is a concept that appears across all programming languages, and logical values (often called *booleans* in other programming languages) have many important uses. Often, they signal whether a condition has been satisfied or whether a parameter should be switched on or off, and you use those values to influence how your code is executed.

You've already encountered logical values briefly, when you used the `sort` function in Section 2.3.2 and the `matrix` function in Section 3.1. Setting `decreasing=TRUE` when using `sort` returns a vector ordered from largest to smallest, and having `decreasing=FALSE` sorts the vector the other way around. Similarly, when constructing a matrix, having `byrow=TRUE` fills the matrix entries row-wise; otherwise, the matrix is filled column-wise. Now, you'll take a more detailed look at ways to use logicals.

4.1.1 TRUE or FALSE?

Logical values in R are written fully as `TRUE` and `FALSE`, but they are frequently abbreviated as `T` or `F`, respectively. The abbreviated version has no effect on the execution of the code (though you must be careful since `T` and `F` are not reserved names; see Section 9.1.4). For example, using `decreasing=T` is equivalent to `decreasing=TRUE` in `sort`.

Assigning logical values to an object is the same as assigning numeric values.

```
R> foo <- TRUE
R> foo
[1] TRUE
R> bar <- F
R> bar
[1] FALSE
```

This gives you one object with the value `TRUE` and one with the value `FALSE`. Similarly, vectors can be filled with logical values.

```
R> baz <- c(T,F,F,F,T,F,T,T,T,F,T,F)
R> baz
 [1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
R> length(x=baz)
[1] 12
```

Matrices (and other higher-dimensional arrays) can be created with these values too, exactly as before. Using `foo` and `baz` from earlier, you could construct something like this:

```
R> qux <- matrix(data=baz,nrow=3,ncol=4,byrow=foo)
R> qux
     [,1]  [,2]  [,3]  [,4]
[1,] TRUE FALSE FALSE FALSE
[2,] TRUE FALSE  TRUE  TRUE
[3,] TRUE FALSE  TRUE FALSE
```

4.1.2 A Logical Outcome: Relational Operators

One common use of logicals is to check whether a certain relationship between values is true. For example, you might want to know whether some number *a* is greater than a predefined threshold *b*. For this, you use the standard *relational operators* shown in Table 4-1, which produce logical values as results.

Table 4-1: Relational Operators

Operator	Interpretation
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Typically, these operators are used on numeric values (though some alternative possibilities will be examined in Section 4.2.1). Here's an example:

```
R> 1==2
[1] FALSE
R> 1>2
[1] FALSE
R> (2-1)<=2
[1] TRUE
R> 1!=(2+3)
[1] TRUE
```

The previous results should be unsurprising: `1` equal to `2` is `FALSE`, `1` greater than `2` is also `FALSE`, and `1` (the result of `2-1`) less than or equal to `2` is `TRUE`; also, it is `TRUE` that `1` is not equal to `5` (`2+3`). These kinds of operations are much more useful when used on numbers that are variable in some way so you can check for satisfaction in a given instance.

You are already familiar with R's element-wise behavior when working with vectors. The same rules apply when using relational operators. To illustrate this, let's first create two vectors and double-check that they're of equal length.

```
R> foo <- c(3,2,1,4,1,2,1,-1,0,3)
R> bar <- c(4,1,2,1,1,0,0,3,0,4)
R> length(x=foo)==length(x=bar)
[1] TRUE
```

Now consider the following four evaluations:

```
R> foo==bar
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
R> foo<bar
 [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
R> foo<=bar
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
R> foo<=(bar+10)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

The first line checks whether the entries in `foo` are equal to the corresponding entries in `bar`, which is true only for the fifth and ninth entries. Note the returned vector is the same length as the vectors being compared, with a logical result to match the position of each pair of elements. The second line compares `foo` and `bar` in the same way, this time checking whether the entries in `foo` are less than the entries in `bar`. Contrast this result with the third comparison, which asks whether entries are less than *or equal to* one another. Finally, the fourth line checks whether `foo`'s members are less than or equal to `bar`, with the elements of `bar` increased by `10`. Naturally, the results are all `TRUE`.

Vector recycling also applies. Let's use `foo` from earlier, along with a shorter vector, `baz`.

```
R> baz <- foo[c(10,3)]
R> baz
[1] 3 1
```

Here you create `baz`, a vector of length 2 comprised of the 10th and 3rd elements of `foo` (in that order). Now consider the following:

```
R> foo>baz
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

Here, the two elements of `baz` are recycled and checked against the 10 elements of `foo`. Elements `1` and `2` of `foo` are checked against `1` and `2` of `baz`, elements `3` and `4` of `foo` are checked against `1` and `2` of `baz`, and so on. You can also check all the values of a vector against a single value. Here's an example:

```
R> foo<3
 [1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

This is a typical operation when handling data sets in R.

Now let's rewrite the contents of `foo` and `bar` as 5×2 column-filled matrices.

```
R> foo.mat <- matrix(foo,nrow=5,ncol=2)
R> foo.mat
     [,1] [,2]
[1,]    3    2
[2,]    2    1
[3,]    1   -1
[4,]    4    0
[5,]    1    3
R> bar.mat <- matrix(bar,nrow=5,ncol=2)
R> bar.mat
     [,1] [,2]
[1,]    4    0
[2,]    1    0
[3,]    2    3
[4,]    1    0
[5,]    1    4
```

The same element-wise behavior from earlier applies here; if you compare the two matrices, you get a logical-valued matrix of the appropriate size.

```
R> foo.mat<=bar.mat
      [,1]  [,2]
[1,]  TRUE FALSE
[2,] FALSE FALSE
[3,]  TRUE  TRUE
[4,] FALSE  TRUE
[5,]  TRUE  TRUE
R> foo.mat<3
      [,1]  [,2]
[1,] FALSE  TRUE
[2,]  TRUE  TRUE
[3,]  TRUE  TRUE
[4,] FALSE  TRUE
[5,]  TRUE FALSE
```

In addition, this same element-wise evaluation applies to arrays of more than two dimensions.

There are two useful functions you can use to quickly inspect a collection of logical values: `any` and `all`. The first returns `TRUE` if, in the vector(s) passed to it, *any* of the logicals contained within are `TRUE`, and it returns `FALSE` otherwise. The second returns a `TRUE` only if *all* of the logicals are `TRUE`, and it returns `FALSE` otherwise. As a quick example, let's work with two of the logical vectors formed by the comparisons of `foo` and `bar` from the beginning of this section.

```
R> qux <- foo==bar
R> qux
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
R> any(qux)
[1] TRUE
R> all(qux)
[1] FALSE
```

Here, the collection of logicals `qux` contains two `TRUE`s, and the rest are `FALSE`—so the result of `any` is of course `TRUE`, but the result of `all` is `FALSE`. Following the same rules, you get this:

```
R> quux <- foo<=(bar+10)
R> quux
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
R> any(quux)
[1] TRUE
R> all(quux)
[1] TRUE
```

Suffice to say, `any` and `all` do the same thing for matrices and arrays of logical values.

Exercise 4.1

a. Store the following vector of 15 values as an object in your workspace: `c(6,9,7,3,6,7,9,6,3,6,6,7,1,9,1)`. Identify the following elements:
   i. Those equal to 6
   ii. Those greater than or equal to 6
   iii. Those less than 6 + 2
   iv. Those not equal to 6

b. Create a new vector from the one used in (a) by deleting its first three elements. With this new vector, fill a $2 \times 2 \times 3$ array. Examine the array for the following entries:
   i. Those less than or equal to 6 divided by 2, plus 4
   ii. Those less than or equal to 6 divided by 2, plus 4, *after* increasing every element in the array by 2

c. Confirm the specific locations of elements equal to 0 in the 10×10 identity matrix I_{10} (see Section 3.3).

d. Check whether *any* of the values of the logical arrays created in (b) are `TRUE`. If they are, check whether they are *all* `TRUE`.

e. By extracting the diagonal elements of the logical matrix created in (c), use `any` to confirm there are no `TRUE` entries.

4.1.3 Multiple Comparisons: Logical Operators

Much of the usefulness of logical values lies in our ability to examine whether multiple conditions are satisfied. Often you will want to perform certain operations only if a number of different conditions have been met.

The relational operators discussed in the previous section compare literal values of stored objects. Now you'll look at *logical operators*, which are used to compare two TRUE or FALSE objects. These operators are based on the statements AND and OR. Table 4-2 summarizes the R syntax and the behavior of logical operators. The AND and OR operators each have a "single" and "element-wise" version—you'll see how they are different in a moment.

Table 4-2: Logical Operators Comparing Two Logical Values

Operator	Interpretation	Results
&	AND (element-wise)	TRUE & TRUE is TRUE TRUE & FALSE is FALSE FALSE & TRUE is FALSE FALSE & FALSE is FALSE
&&	AND (single comparison)	Same as for & above
|	OR (element-wise)	TRUE|TRUE is TRUE TRUE|FALSE is TRUE FALSE|TRUE is TRUE FALSE|FALSE is FALSE
||	OR (single comparison)	Same as for | above
!	NOT	!TRUE is FALSE !FALSE is TRUE

The result of using any logical operator is itself a logical value. An AND comparison is true only if *both* logicals are TRUE. An OR comparison is true if at least one of the logicals is TRUE. The NOT operator (!) simply returns the opposite of the logical value it's used on. You can combine these operators to examine multiple conditions at once.

```
R> FALSE||((T&&TRUE)||FALSE)
[1] TRUE
R> !TRUE&&TRUE
[1] FALSE
R> (T&&(TRUE||F))&&FALSE
[1] FALSE
R> (6<4)||(3!=1)
[1] TRUE
```

As with numeric arithmetic, R imposes an order of importance on logical operators. An AND statement has a higher precedence than an OR statement. It's helpful to place each comparative pair in parentheses to preserve the correct order of evaluation and make the code more readable. You can see this in the first line in the previous code, where the innermost

comparison is the first to be carried out: `T&&TRUE` results in `TRUE`; this is then provided as one of the logical values for the next bracketed comparison where `TRUE||FALSE` results in `TRUE`. The final comparison is then `FALSE||TRUE`, resulting in the final outcome, `TRUE`, which is printed to the console. The second line reads as NOT `TRUE` AND `TRUE`, which of course returns `FALSE`. In the third line, once again the innermost pair is evaluated first: `TRUE||F` is `TRUE`; `T&&TRUE` is `TRUE`; and finally `TRUE&&FALSE` is `FALSE`. The fourth and final example evaluates two distinct conditions in parentheses, which are then compared using a logical operator. Since `6<4` is `FALSE` and `3!=1` is `TRUE`, that gives you a logical comparison of `FALSE||TRUE` and a final result of `TRUE`.

In Table 4-2, there is a short (`&`, `|`) and long (`&&`, `||`) version of the AND and OR operators. The short versions are meant for element-wise comparisons, where you have two logical vectors and you want multiple logicals as a result. The long versions, which you've been using so far, are meant for comparing two individual values and will return a single logical value. This is important when programming conditional checks in R in an `if` statement, which you'll look at in Chapter 10. It is possible to compare a single pair of logicals using the short version—though it is considered better practice to use the longer versions when a single `TRUE/FALSE` result is needed.

Let's look at some examples of element-wise comparisons. Suppose you have two vectors of equal length, `foo` and `bar`:

```
R> foo <- c(T,F,F,F,T,F,T,T,T,F,T,F)
R> foo
 [1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

and

```
R> bar <- c(F,T,F,T,F,F,F,F,T,T,T,T)
R> bar
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

The short versions of the logical operators match each pair of elements by position and return the result of the comparison.

```
R> foo&bar
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
R> foo|bar
 [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Using the long version of the operators, on the other hand, means R carries out the comparison only on the first pair of logicals in the two vectors.

```
R> foo&&bar
[1] FALSE
R> foo||bar
[1] TRUE
```

Notice that the last two results match the first entries of the vectors you got using the short versions of the logical operators.

Exercise 4.2

a. Store the vector `c(7,1,7,10,5,9,10,3,10,8)` as `foo`. Identify the elements greater than 5 *or* equal to 2.

b. Store the vector `c(8,8,4,4,5,1,5,6,6,8)` as `bar`. Identify the elements less than or equal to 6 *and* not equal to 4.

c. Identify the elements that satisfy (a) in `foo` *and* satisfy (b) in `bar`.

d. Store a third vector called `baz` that is equal to the element-wise sum of `foo` and `bar`. Determine the following:
   i. The elements of `baz` greater than or equal to 14 but not equal to 15
   ii. The elements of the vector obtained via an element-wise division of `baz` by `foo` that are greater than 4 *or* less than or equal to 2

e. Confirm that using the long version in all of the preceding exercises performs only the first comparison (that is, the results each match the first entries of the previously obtained vectors).

4.1.4 Logicals Are Numbers!

The binary nature of logical values lends itself to a representation of `TRUE` as 1 and `FALSE` as 0. In fact, in R, if you perform elementary numerical operations on logical values, `TRUE` is treated like `1`, and `FALSE` is treated like `0`.

```
R> TRUE+TRUE
[1] 2
R> FALSE-TRUE
[1] -1
R> T+T+F+T+F+F+T
[1] 4
```

These operations turn out the same as if you had used the digits 1 and 0. Similarly, in certain (but not all) situations where logicals are required, you can substitute the corresponding numerical values.

```
R> 1&&1
[1] TRUE
R> 1||0
[1] TRUE
R> 0&&1
[1] FALSE
```

This dual interpretation allows you to use many arithmetic functions on vectors of logicals, which can facilitate the summary of TRUE/FALSE data. This possibility will be explored a little further in Part II.

4.1.5 Logical Subsetting and Extraction

So far, you've been extracting values from vectors and higher-dimensional objects using index vectors, which specify the positions of the elements you want to extract. You can use logical values to perform the same kinds of operations. Rather than entering explicit indexes in the square brackets, you can supply logical *flag* vectors, where an element is extracted if the corresponding entry in the flag vector is TRUE. As such, logical flag vectors should be the same length as the vector that is being accessed (though recycling does occur for shorter flag vectors, as a later example shows).

At the beginning of Section 2.3.3 you defined a vector of length 10 to be as follows:

```
R> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
```

You could type myvec[c(2,10)] to extract the two negative elements of myvec. You can also do the following, using logical flags:

```
R> myvec[c(F,T,F,F,F,F,F,F,T)]
[1] -2.3 -8.0
```

This particular example may seem far too cumbersome for practical use. However, instead of writing the logical flag vector explicitly, you can extract elements based on whether they satisfy a certain condition (or several conditions). Consider, for example, how easily a relational operator can identify the negative elements in myvec.

```
R> myvec<0
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

This a perfectly valid flag vector that you can use to subset myvec to get the same result as earlier.

```
R> myvec[myvec<0]
[1] -2.3 -8.0
```

As mentioned earlier, R implements the recycling of the flag vector if it is too short. To extract every second element from myvec, starting with the first, you could enter the following:

```
R> myvec[c(T,F)]
[1]     5     4     4     8 40221
```

You can also do more complicated extractions using relational and logical operators.

```
R> myvec[(myvec>0)&(myvec<1000)]
[1]  5  4  4  4  6  8 10
```

This returns the positive elements that are less than 1,000. You can also overwrite specific elements using a logical flag vector, just as with index vectors.

```
R> myvec[myvec<0] <- -200
R> myvec
 [1]     5  -200     4     4     4     6     8    10 40221  -200
```

This replaces all existing negative entries with −200. Note, though, that you cannot directly use negative logical flag vectors to delete specific elements; this can be done only with numeric index vectors.

Logicals can be useful for element extraction. You don't need to know beforehand which specific index positions to return, since the conditional check can find them for you. This is particularly valuable when you are dealing with large data sets and you want to inspect records or recode entries that match certain criteria.

In some cases, you might want to convert a logical flag vector into a numeric index vector. This is helpful when you need the explicit indexes of elements that were flagged `TRUE`. The R function `which` takes in a logical vector as the argument `x` and returns the indexes corresponding to the positions of any and all `TRUE` entries.

```
R> which(x=c(T,F,F,T,T))
[1] 1 4 5
```

You can use this to identify the index positions of `myvec` that meet a certain condition (for example, those containing negative numbers).

```
R> which(x=myvec<0)
[1]  2 10
```

You could do the same for the other `myvec` selections you experimented with. Note that a line of code such as `myvec[which(x=myvec<0)]` is redundant because that extraction can be made using the condition by itself, without using `which`. On the other hand, using `which` lets you delete elements based on logical flag vectors. You can simply use `which` to identify the numeric indexes you want to delete and render them negative. To omit the negative entries of `myvec`, you could execute the following:

```
R> myvec[-which(x=myvec<0)]
[1]     5     4     4     4     6     8    10 40221
```

These ideas carry through to matrices and other arrays. In Section 3.2, you stored a 3×3 matrix as follows:

```
R> A <- matrix(c(0.3,4.5,55.3,91,0.1,105.5,-4.2,8.2,27.9),nrow=3,ncol=3)
R> A
     [,1]  [,2] [,3]
[1,]  0.3  91.0 -4.2
[2,]  4.5   0.1  8.2
[3,] 55.3 105.5 27.9
```

To extract the second and third column elements of the first row of `A` using numeric indexes, you could execute `A[1,2:3]`. To do this with logical flags, you could enter the following:

```
R> A[c(T,F,F),c(F,T,T)]
[1] 91.0 -4.2
```

As before, you usually wouldn't explicitly specify the logical vectors. Suppose for example you want to replace all elements in `A` that are less than 1 with −7. Performing this using numeric indexes is rather fiddly. It's much easier to use the logical flag matrix created with the following:

```
R> A<1
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE  TRUE
[2,] FALSE  TRUE FALSE
[3,] FALSE FALSE FALSE
```

You can supply this logical matrix to the square bracket operators, and the replacement is simple.

```
R> A[A<1] <- -7
R> A
     [,1]  [,2] [,3]
[1,] -7.0  91.0 -7.0
[2,]  4.5  -7.0  8.2
[3,] 55.3 105.5 27.9
```

Note that this is the first time you have subsetted a matrix without using commas to separate out the dimensions in the square brackets. All the required information associated with element positioning is provided by the flag matrix, which has the same number of rows and columns as the target matrix.

If you use `which` to identify numeric indexes based on a logical flag structure, you have to be a little more careful when dealing with two-dimensional objects or higher. Suppose you want the index positions of the elements that are greater than 25. The appropriate logical matrix is as follows:

```
R> A>25
      [,1]  [,2]  [,3]
[1,] FALSE  TRUE FALSE
[2,] FALSE FALSE FALSE
[3,]  TRUE  TRUE  TRUE
```

Now, say you ask R the following:

```
R> which(x=A>25)
[1] 3 4 6 9
```

This indeed returns four indexes (since there are four elements that satisfied the relational check), but they are provided as scalar values. How do these correspond to the row/column positioning of the matrix?

The answer lies in R's default behavior for the `which` function. When you use `which` this way, it essentially treats the multidimensional object as a single vector (laid out column after column) and then returns the vector of corresponding indexes. Say the matrix `A` was arranged as a vector by stacking the columns first through third (using `c(A[,1],A[,2],A[,3])`. Then the indexes returned earlier make more sense.

```
R> which(x=c(A[,1],A[,2],A[,3])>25)
[1] 3 4 6 9
```

This can be difficult to interpret, though, especially when dealing with higher-dimensional arrays. In this kind of situation, you can make `which` return dimension-specific indexes using the optional argument `arr.ind` (array indexes). By default, this argument is set to `FALSE`, which results in vector-converted indexes for matrices and other arrays. Setting `arr.ind` to `TRUE`, however, provides you with the dimension-specific positions of elements in `x` that are `TRUE`.

```
R> which(x=A>25,arr.ind=T)
     row col
[1,]   3   1
[2,]   1   2
[3,]   3   2
[4,]   3   3
```

The returned object is now a matrix, where each row represents an element that satisfied the logical comparison and each column provides the position of the element. Comparing the output here with `A`, you can see these positions do indeed correspond to elements where `A>25`.

Both versions of the output (with `arr.ind=T` or `arr.ind=F`) can be useful—the correct choice depends on the application.

Exercise 4.3

a. Store this vector of 10 values: `foo <- c(7,5,6,1,2,10,8,3,8,2)`. Then, do the following:
   i. Extract the elements greater than or equal to 5, storing the result as `bar`.
   ii. Display the vector containing those elements from `foo` that remain after omitting all elements that are greater than or equal to 5.

b. Use `bar` from (a)(i) to construct a 2×3 matrix called `baz`, filling in a row-wise fashion. Then, do the following:
   i. Replace any elements that are equal to 8 with the *squared* value of the element in row 1, column 2 of `baz` itself.
   ii. Confirm that *all* values in `baz` are now less than or equal to 25 *and* greater than 4.

c. Create a $3 \times 2 \times 3$ array called `qux` using the following vector of 18 values: `c(10,5,1,4,7,4,3,3,1,3,4,3,1,7,8,3,7,3)`. Then, do the following:
   i. Identify the dimension-specific index positions of elements that are either 3 *or* 4.
   ii. Replace with 100 all elements in `qux` that are less than 3 *or* greater than or equal to 7.

d. Return to `foo` from (a). Use the vector `c(F,T)` to extract every second value from `foo`. In Section 4.1.4, you saw that in some situations, you can substitute `0` and `1` for `TRUE` and `FALSE`. Can you perform the same extraction from `foo` using the vector `c(0,1)`? Why or why not? What does R return in this case?

4.2 Characters

Character strings, another data type available in most high-level computing languages, are used to represent text, thereby playing a role in a variety of tasks. In R, they are often used to specify folder locations or software options (as shown briefly in Chapter 1); to select among a fixed set of options when supplying an argument to a function; and to annotate stored objects, provide textual output, or help clarify plots and graphics. In a simple way, they can also be used to define different groups making up a categorical variable, though as you'll see in see Section 4.3, *factors* are better suited for that.

NOTE *There are three different string formats in the R environment. The default string format is called an* extended regular expression*; the other variants are named* Perl *and* literal regular expressions. *The intricacies of these variants are beyond the scope of this book, so any mention of character strings from here on refers to an extended regular expression. For more technical details about other string formats, type* `?regex` *at the prompt.*

4.2.1 Creating a String

Character strings are indicated by double quotation marks, ". To create a string, just enter text between a pair of quotes.

```
R> foo <- "This is a character string!"
R> foo
[1] "This is a character string!"
R> length(x=foo)
[1] 1
```

Note that R treats the string as a single entity. In other words, `foo` is a vector of length 1 (`length` as used here does not count individual words or characters; it counts only the total number of distinct strings). To count the number of individual characters, you can use the `nchar` function. Here's an example using `foo` from the previous code:

```
R> nchar(x=foo)
[1] 27
```

This function can be particularly useful when using a loop, for example to search for a certain character in a string (see Chapter 10).

Almost any combination of characters, including numbers, can be a valid character string.

```
R> bar <- "23.3"
R> bar
[1] "23.3"
```

Note that in this form, the string has no numerical meaning, and it won't be treated like the number 23.3. Attempting to multiply it by 2, for example, results in an error.

```
R> bar*2
Error in bar * 2 : non-numeric argument to binary operator
```

This error occurs because * is expecting to operate on two numeric values (not one number and one string, which makes no sense).

Strings can be compared in several ways, the most common comparison being a check for equality.

```
R> "alpha"=="alpha"
[1] TRUE
R> "alpha"!="beta"
[1] TRUE
R> c("alpha","beta","gamma")=="beta"
[1] FALSE  TRUE FALSE
```

The other relational operators also work in an intuitive fashion. For example, alphabetical ordering is retained in that "later" letters in the alphabet are considered to be greater than "earlier" letters.

```
R> "alpha"<="beta"
[1] TRUE
R> "gamma">"Alpha"
[1] TRUE
```

Furthermore, uppercase letters are considered greater than lowercase letters.

```
R> "Alpha">"alpha"
[1] TRUE
R> "beta">="bEtA"
[1] FALSE
```

Most symbols can also be used in a string. The following string is valid, for example:

```
R> baz <- "&4 _ 3 **%.? $ymbolic non$en$e ,; "
R> baz
[1] "&4 _ 3 **%.? $ymbolic non$en$e ,; "
```

One important exception is the backslash \, also called an *escape*. When a backslash is used within the quotation marks of a string, it initiates some simple control over the printing or display of the string itself. You'll see how this works in a moment in Section 4.2.3. First let's look at two useful functions for concatenating strings.

4.2.2 Concatenation

There are two main functions used to *concatenate* (or glue together) one or more strings: `cat` and `paste`. The fundamental difference between the two is how their contents are returned. The first function, `cat`, sends its output directly to the console screen and does not formally *return* anything. The `paste` function concatenates its contents and then returns the final character string as a usable R object. This is useful when the result of a string concatenation needs to be passed to another function or used in some secondary

way, as opposed to just being displayed. Consider the following vector of character strings:

```
R> qux <- c("awesome","R","is")
R> length(x=qux)
[1] 3
R> qux
[1] "awesome" "R"       "is"
```

As with numbers and logicals, you can also store any number of strings in an appropriately dimensioned matrix or array structure if you want.

When calling `cat` or `paste`, you pass arguments to the function in the order you want them combined. The following lines show the identical usage yet different type of output from the two functions:

```
R> cat(qux[2],qux[3],"totally",qux[1],"!")
R is totally awesome !
R> paste(qux[2],qux[3],"totally",qux[1],"!")
[1] "R is totally awesome !"
```

Here, we've used the three elements of `qux` as well as two additional strings, `"totally"` and `"!"`, to produce the final concatenated string. In the output, note that `cat` has simply concatenated and printed the text to the screen. This means you cannot directly assign the result to a new variable and treat it as a character string. For `paste`, however, the `[1]` to the left of the output and the presence of the " quotes indicate the returned item is a vector containing a character string, and this can be assigned to an object and used in other functions.

NOTE *There is a slight difference between Mac OS X and Windows in the default handling of string concatenation when using the R GUI. After calling* `cat` *in Windows, the new R prompt awaiting your next command appears on the same line as the printed string. On a Mac, the new prompt appears on the next line as usual. This problem can be easily overcome in Windows by using a carriage return* escape sequence, `\n`, *at the end of the last string in each call to* `cat`. *Escape sequences are discussed in Section 4.2.3.*

These two functions have an optional argument, `sep`. This argument takes in a character string (the default is a single space, `sep=" "`) that is used to separate the multiple strings when they are concatenated to produce the final result.

```
R> paste(qux[2],qux[3],"totally",qux[1],"!",sep="---")
[1] "R---is---totally---awesome---!"
R> paste(qux[2],qux[3],"totally",qux[1],"!",sep="")
[1] "Ristotallyawesome!"
```

The same behavior would occur for `cat`. Note that if you don't want any separation, you set `sep=""`, an empty string, as shown in the second example.

The empty string separator can be used to achieve correct sentence spacing; note the gap between `awesome` and the exclamation mark in the previous code when you first used `paste` and `cat`.

For example, using manual insertion of spaces where necessary, you can write the following:

```
R> cat("Do you think ",qux[2]," ",qux[3]," ",qux[1],"?",sep="")
Do you think R is awesome?
```

Concatenation can be useful when you want to neatly summarize the results from a certain function or set of calculations. Many kinds of R objects can be passed directly to `paste` or `cat`; the software will attempt to automatically *coerce* these items into character strings. This means R will convert the input into a string so the values can be included in the final concatenated string. This works particularly well with numeric objects, as the following examples demonstrate:

```
R> a <- 3
R> b <- 4.4
R> cat("The value stored as 'a' is ",a,".",sep="")
The value stored as 'a' is 3.
R> paste("The value stored as 'b' is ",b,".",sep="")
[1] "The value stored as 'b' is 4.4."
R> cat("The result of a+b is ",a,"+",b,"=",a+b,".",sep="")
The result of a+b is 3+4.4=7.4.
R> paste("Is ",a+b," less than 10? That's totally ",a+b<10,".",sep="")
[1] "Is 7.4 less than 10? That's totally TRUE."
```

Here, the values of the non-string objects are placed where you want them in the final string output. The results of calculations can also appear as fields, as shown with the arithmetic `a+b` and the logical comparison `a+b<10`. You'll see more details about coercion from one kind of value to another in Section 6.2.

4.2.3 Escape Sequences

In Section 4.2.1, I noted that a stand-alone backslash doesn't act like a normal character within a string. The `\` is used to invoke an *escape sequence*. An escape sequence lets you enter characters that control the format and spacing of the string, rather than being interpreted as normal text. Table 4-3 describes some of the most common escape sequences, and you can find a full list by typing `?Quotes` at the prompt.

Table 4-3: Common Escape Sequences for Use in Character Strings

Escape sequence	Result
`\n`	Starts a newline
`\t`	Horizontal tab
`\b`	Invokes a backspace
`\\`	Used as a single backslash
`\"`	Includes a double quote

Escape sequences add flexibility to the display of character strings, which can be useful for summaries of results and plot annotations. Enter the sequence precisely where it should take effect. Here's an example:

```
R> cat("here is a string\nsplit\tto neww\b\n\n\tlines")
here is a string
split    to new

    lines
```

Since the signal for an escape is `\` and the signal to begin and end a string is `"`, if you want either of these characters to be included in a string, you must also use an escape to have them be interpreted as a normal character.

```
R> cat("I really want a backslash: \\\nand a double quote: \"")
I really want a backslash: \
and a double quote: "
```

These escape sequences prohibit using a stand-alone backslash in file path strings in R. As noted in Section 1.2.3 (where you used `getwd` to print the current working directory and `setwd` to change it), folder separation must use a forward slash `/` and not a backslash.

```
R> setwd("/folder1/folder2/folder3/")
```

File path specification crops up when reading and writing files, which you'll explore in Chapter 8.

4.2.4 Substrings and Matching

Occasionally, you need to do a little more with character strings than simply tailor them for output. *Pattern matching* lets you inspect a given string to identify smaller strings.

The function `substr` takes a string `x` and extracts the part of the string between character number `start` and character number `stop` (inclusive). Let's try it on the object `foo` from Section 4.2.1.

```
R> foo <- "This is a character string!"
R> substr(x=foo,start=21,stop=27)
[1] "string!"
```

Here, you've extracted the characters between positions 21 and 27, inclusive, to get `"string!"`. The function `substr` can also be used with the assignment operator to directly substitute in a new set of characters. In this case, the replacement string should contain the same number of characters as the selected area.

```
R> substr(x=foo,start=1,stop=4) <- "Here"
R> foo
[1] "Here is a character string!"
```

If the replacement string is of a different number of characters than indicated by `start` and `stop`, then replacement still takes place, beginning at `start`. Should the replacement string be shorter, then replacement ends when the desired string is fully inserted, leaving the original characters up to `stop` untouched. If it is longer, then replacement uses as many characters as possible from the string to insert, truncating the replacement when it reaches `stop`.

Substitution is more flexible using the functions `sub` and `gsub`. The `sub` function searches a given string `x` for a smaller string `pattern` contained within. It then replaces the first instance with a new string, `replacement`. The `gsub` function does the same thing, but it replaces *every* instance of `pattern`. Here's an example:

```
R> bar <- "How much wood could a woodchuck chuck"
R> sub(pattern="chuck",replacement="hurl",x=bar)
[1] "How much wood could a woodhurl chuck"
R> gsub(pattern="chuck",replacement="hurl",x=bar)
[1] "How much wood could a woodhurl hurl"
```

With `sub` and `gsub`, the `replacement` value need not have the same number of characters as the `pattern` being replaced. These functions also have options to control the matching, such as whether it should be case-sensitive. The help files `?substr` and `?sub` have more details, as well as noting a handful of other pattern-matching functions and techniques.

Exercise 4.4

a. Re-create exactly the following output:

```
"The quick brown fox
    jumped over
        the lazy dogs"
```

b. Suppose you have stored the values `num1 <- 4` and `num2 <- 0.75`. Write a line of R code that returns the following character string:

```
[1] "The result of multiplying 4 by 0.75 is 3"
```

Make sure your code produces a string with the correct multiplication result for *any* two numbers stored as `num1` and `num2`.

c. On my local machine, the directory for this book's LaTeX files is specified in R as `"/Users/tdavies/Documents/RBook/"`. Imagine it is your machine—write a line of code that replaces *tdavies* in the previous string with your first initial and surname.

d. In Section 4.2.4, you stored the following string:

```
R> bar <- "How much wood could a woodchuck chuck"
```

i. Store a new string by gluing onto `bar` the words `"if a woodchuck could chuck wood"`.
ii. In the result of (i), replace all instances of `wood` with `metal`.

e. Store the string `"Two 6-packs for $16.99"`. Then do the following:
i. Use a check for equality to confirm that the substring beginning with character 5 and ending with character 10 is `"6-pack"`.
ii. Make it a better deal by changing the price to $14.99.

4.3 Factors

In this section, you'll look at some simple functions related to creating, handling, and inspecting *factors*. Factors are R's most natural way of representing data points that fit in only one of a finite number of distinct categories, rather than belonging to a continuum. Categorical data like this can play an important role in data science, and you'll look at factors again in more detail from a statistical perspective in Chapter 13.

4.3.1 Identifying Categories

To see how factors work, let's start with a simple data set. Suppose you find eight people and record their first name, gender, and month of birth in Table 4-4.

Table 4-4: An Example Data Set of Eight Individuals

Person	Gender	Month of birth
Liz	Female	April
Jolene	Female	January
Susan	Female	December
Boris	Male	September
Rochelle	Female	November
Tim	Male	July
Simon	Male	July
Amy	Female	June

There is really only one sensible way to represent the name of each person in R—as a vector of character strings.

```
R> firstname <- c("Liz","Jolene","Susan","Boris","Rochelle","Tim","Simon",
                  "Amy")
```

You have more flexibility when it comes to recording gender, however. Coding females as 0 and males as 1, a numeric option would be as follows:

```
R> sex.num <- c(0,0,0,1,0,1,1,0)
```

Of course, character strings are also possible, and many prefer this because there's no need to remember the numeric codes associated with the different groups.

```
R> sex.char <- c("female","female","female","male","female","male","male",
                 "female")
```

There is, however, a fundamental difference between the name of an individual and their gender in terms of how you would treat them in any kind of analysis. Where a person's name is a unique identifier that can take any one of an infinite number of possibilities, there are only two options for recording a person's gender. These kinds of data, where all possible values fall into a finite number of categories, is best represented in R using factors.

Factors are typically created from a numeric or a character vector (note that you cannot fill matrices or multidimensional arrays using factor values; factors can only take the form of vectors). To create a factor vector, use the function `factor`, as in this example working with `sex.num` and `sex.char`:

```
R> sex.num.fac <- factor(x=sex.num)
R> sex.num.fac
[1] 0 0 0 1 0 1 1 0
Levels: 0 1
R> sex.char.fac <- factor(x=sex.char)
R> sex.char.fac
```

```
[1] female female female male   female male   male   female
Levels: female male
```

Here, you obtain factor versions of the two vectors storing gender values.

At first glance, these objects don't look much different from the numeric and character vectors from which they were created. Indeed, factor objects work in much the same way as vectors, but with a little extra information attached (R's internal representation of factor objects is a little different as well). Functions like `length` and `which` work the same way on factor objects as with vectors, for example.

The most important extra piece of information (or *attribute*; see Section 6.2.1) that a factor object contains is its *levels*, which store the possible values in the factor. These levels are printed at the bottom of each factor vector. You can extract the levels as a vector of character strings using the `levels` function.

```
R> levels(x=sex.num.fac)
[1] "0" "1"
R> levels(x=sex.char.fac)
[1] "female" "male"
```

You can also relabel a factor using `levels`. Here's an example:

```
R> levels(x=sex.num.fac) <- c("1","2")
R> sex.num.fac
[1] 1 1 1 2 1 2 2 1
Levels: 1 2
```

This relabels the females `1` and the males `2`.

Factor-valued vectors are subsetted in the same way as any other vector.

```
R> sex.char.fac[2:5]
[1] female female male   female
Levels: female male
R> sex.char.fac[c(1:3,5,8)]
[1] female female female female female
Levels: female male
```

Note that after subsetting a factor object, the object continues to store *all* defined levels even if some of the possible levels are no longer represented after the extraction.

If you want to subset from a factor using a logical flag vector, keep in mind that the levels of a factor are stored as character strings (even if the original data vector was numeric). To, for example, identify all the men (using the newly relabeled `sex.num.fac`), use this:

```
R> sex.num.fac=="2"
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

Since the element positions in `firstname` correspond to the element positions in the factor vectors for gender, you can then use this logical vector to obtain the names of all the men (this time using the `"male"`/`"female"` factor vector).

```
R> firstname[sex.char.fac=="male"]
[1] "Boris" "Tim"   "Simon"
```

Of course, this simple subsetting from the previous example could have been achieved in much the same way with the raw numeric vector `sex.num` or the raw character vector `sex.char`. In the next section, you'll explore some more distinctive advantages to having categorical data represented as a factor in R. This includes being able to define possible levels even if there are no observations for those categories, assigning the levels an explicit order, or modifying them in other ways.

4.3.2 Defining and Ordering Levels

In the previous section, you examined the representation of gender as a factor. This is the simplest kind of factor variable—there are only two possible levels with no ordering (one level is not considered "higher than" or "following" the other in any qualitative sense). The same cannot be said for month of birth (MOB), where there are 12 levels that have a natural order. Let's store the observed MOB data from earlier as a character vector.

```
R> mob <- c("Apr","Jan","Dec","Sep","Nov","Jul","Jul","Jun")
```

There are two problems with the data in this vector. First, not all possible categories are represented since `mob` contains only six unique months. Second, this vector doesn't reflect the natural order of the months.

```
R> mob[2]
[1] "Jan"
R> mob[3]
[1] "Dec"
R> mob[2]<mob[3]
[1] FALSE
```

Alphabetically, this result is of course correct—*J* does not occur before *D*. But in terms of the order of the calendar months (which is what we're interested in), the `FALSE` result is incorrect.

If you create a factor object from these values, you can deal with both of these problems by supplying additional arguments to the `factor` function. Additional levels can be defined by supplying a character vector of all possible values to the `levels` argument. R may also be instructed to order the values precisely as they appear in `levels` by setting the argument `ordered` to `TRUE`.

```
R> ms <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov",
          "Dec")
R> mob.fac <- factor(x=mob,levels=ms,ordered=TRUE)
R> mob.fac
[1] Apr Jan Dec Sep Nov Jul Jul Jun
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

Here, the `mob.fac` vector contains the same values in the same order as the `mob` vector from earlier. But notice that this variable does in fact have 12 levels, even though you have not made any observations for the levels `"Feb"`, `"Mar"`, `"May"`, `"Aug"`, or `"Oct"`. (Note that if your R console window is too narrow to print all the levels to the screen, one or more may be suppressed and replaced with …, simply indicating there's more output that's been hidden. This is easily remedied by widening the window and reinspecting the `mob.fac` object.) Also, the strict order of these levels is shown by the `<` symbol in the object output. Using this new factor object, the relational comparison from earlier now produces the desired result.

```
R> mob.fac[2]
[1] Jan
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
R> mob.fac[3]
[1] Dec
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
R> mob.fac[2]<mob.fac[3]
[1] TRUE
```

These improvements are far from just cosmetic. There's a big difference, for example, between a data set with zero observations in some of the categories and the same data set defined with fewer categories to begin with. The choice of whether to instruct R to formally order a factor vector can also have important consequences in the implementation of various statistical methods, such as regression and other types of modeling.

4.3.3 Combining and Cutting

As you've seen, it's usually simple to combine multiple vectors of the same kind (whether numeric, logical, or character) using the `c` function. Here's an example:

```
R> foo <- c(5.1,3.3,3.1,4)
R> bar <- c(4.5,1.2)
R> c(foo,bar)
[1] 5.1 3.3 3.1 4.0 4.5 1.2
```

This combines the two numeric vectors into one.

Note, however, that the c function does not work the same way with factor-valued vectors. Let's see what happens when you use it on the data in Table 4-4 and the associated MOB factor vector mob.fac, from Section 4.3.2. Suppose you now observe three more individuals with MOB values "Oct", "Feb", and "Feb", which are stored as a factor object.

```
R> new.values <- factor(x=c("Oct","Feb","Feb"),levels=levels(mob.fac),
ordered=TRUE)
R> new.values
[1] Oct Feb Feb
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

Now you have mob.fac with the original eight observations and new.values with an additional three. Both are factor objects, defined with identical, ordered levels. You might expect that you can just use c to combine the two as follows:

```
R> c(mob.fac,new.values)
 [1]  4  1 12  9 11  7  7  6 10  2  2
```

Clearly, this has not done what you want it to do. Combining the two factor objects resulted in a numeric vector. This is because the c function interprets factors as integers. If you compare the previous integer output directly with the defined levels, you can see that the numbers refer to the index of each month within the ordered levels.

```
R> levels(mob.fac)
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

This means you can use these integers with levels(mob.fac) to retrieve a character vector of the complete observed data—the original eight observations plus the additional three.

```
R> levels(mob.fac)[c(mob.fac,new.values)]
 [1] "Apr" "Jan" "Dec" "Sep" "Nov" "Jul" "Jul" "Jun" "Oct" "Feb" "Feb"
```

Now you have all the observations stored in a vector, but they are currently stored as strings, not factor values. The final step is to turn this vector into a factor object.

```
R> mob.new <- levels(mob.fac)[c(mob.fac,new.values)]
R> mob.new.fac <- factor(x=mob.new,levels=levels(mob.fac),ordered=TRUE)
R> mob.new.fac
 [1] Apr Jan Dec Sep Nov Jul Jul Jun Oct Feb Feb
Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

As this example shows, combining factors requires you to essentially deconstruct the two objects and then rebuild them. This helps ensure that the levels are consistent and the observations are valid in the final product.

Factors are also often created from data that was originally measured on a continuum, for example the weight of a set of adults or the amount of a drug given to a patient. Sometimes it's required to group (or *bin*) these types of observations into categories, like Small/Medium/Large or Low/High. In R, you can mold this kind of data into discrete factor categories using the `cut` function. Consider the following numeric vector of length 10:

```
R> Y <- c(0.53,5.4,1.5,3.33,0.45,0.01,2,4.2,1.99,1.01)
```

Suppose you want to bin the data as follows: *Small* refers to observations in the interval $[0,2)$, *Medium* refers to $[2,4)$, and *Large* refers to $[4,6]$ (a square bracket refers to *inclusion* of its nearest value, and a parenthesis indicates *exclusion*). That is to say, an observation y will fall in the Small interval if $0 \leq y < 2$, in Medium if $2 \leq y < 4$, or in Large if $4 \leq y \leq 6$. The function `cut` can create a factor object based on this numeric data. Supply the intervals as a vector (the object named `br` directly below) to the argument `breaks`.

```
R> br <- c(0,2,4,6)
R> cut(x=Y,breaks=br)
 [1] (0,2] (4,6] (0,2] (2,4] (0,2] (0,2] (0,2] (4,6] (0,2] (0,2]
Levels: (0,2] (2,4] (4,6]
```

This gives you a factor, with each observation now assigned an interval. But note that the boundary values are on the right side; you want them on the left (in other words, you want the first level to be $[0,2)$ instead of $(0,2]$). You can fix this by setting the logical argument `right` to `right=F`.

```
R> cut(x=Y,breaks=br,right=F)
 [1] [0,2) [4,6) [0,2) [2,4) [0,2) [0,2) [2,4) [4,6) [0,2) [0,2)
Levels: [0,2) [2,4) [4,6)
```

Now you've swapped which boundaries are inclusive and exclusive. This is important because it affects the category of any values that equal one of these boundaries (note the seventh observation in the two factor outputs shown previously). But there's still a problem: the final interval currently *excludes* 6, and you want this maximum value to be *included* in the highest level. You can fix this with another logical argument: `include.lowest` (even though it is called '`include.lowest`', the help file `?cut` indicates that this should be interpreted as including the *highest* value if `right=F`, as is the case here).

```
R> cut(x=Y,breaks=br,right=F,include.lowest=T)
 [1] [0,2) [4,6] [0,2) [2,4) [0,2) [0,2) [2,4) [4,6] [0,2) [0,2)
Levels: [0,2) [2,4) [4,6]
```

The intervals are now defined how you want, but by default, the intervals themselves have been used as the labels for each category. You can add better labels by passing a character string vector to the `labels` argument. The order of labels must match the order of the levels in the factor object.

```
R> lab <- c("Small","Medium","Large")
R> cut(x=Y,breaks=br,right=F,include.lowest=T,labels=lab)
 [1] Small  Large  Small  Medium Small  Small  Medium Large  Small  Small
Levels: Small Medium Large
```

Exercise 4.5

The New Zealand government consists of the political parties National, Labour, Greens, and Māori, with several smaller parties labeled as Other. Suppose you asked 20 New Zealanders which of these they identified most with and obtained the following data:

- There were 12 males and 8 females; the individuals numbered 1, 5–7, 12, and 14–16 were females.
- The individuals numbered 1, 4, 12, 15, 16, and 19 identified with Labour; no one identified with Māori; the individuals numbered 6, 9, and 11 identified with Greens; 10 and 20 identified with Other; and the rest identified with National.

a. Use your knowledge of vectors (for example, subsetting and overwriting) to create two character vectors: `sex` with entries `"M"` (male) and `"F"` (female) and `party` with entries `"National"`, `"Labour"`, `"Greens"`, `"Maori"`, and `"Other"`. Make sure the entries are placed in the correct positions as outlined earlier.

b. Create two different factor vectors based on `sex` and `party`. Does it make any sense to use `ordered=TRUE` in either case? How has R appeared to arrange the levels?

c. Use factor subsetting to do the following:
   i. Return the factor vector of chosen parties for only the male participants.
   ii. Return the factor vector of genders for those who chose National.

d. Another six people joined the survey, with the results `c("National","Maori","Maori","Labour","Greens","Labour")` for the preferred party and `c("M","M","F","F","F","M")` as their gender. Combine these results with the original factors from (b).

Suppose you also asked all individuals to state how confident they were that Labour will win more seats in Parliament than National in the next election and to attach a subjective percentage to that

confidence. The following 26 results were obtained: 93, 55, 29, 100, 52, 84, 56, 0, 33, 52, 35, 53, 55, 46, 40, 40, 56, 45, 64, 31, 10, 29, 40, 95, 18, 61.

e. Create a factor with levels of confidence as follows: Low for percentages [0,30]; Moderate for percentages (30,70]; and High for percentages (70,100].

f. From (e), extract the levels corresponding to those individuals who originally said they identified with Labour. Do this also for National. What do you notice?

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`TRUE, FALSE`	Reserved logical values	Section 4.1.1, p. 58
`T, F`	Unreserved versions of above	Section 4.1.1, p. 58
`==, !=, >, <, >=, <=`	relational operators	Section 4.1.2, p. 59
`any`	Checks whether any entries are `TRUE`	Section 4.1.2, p. 61
`all`	Checks whether all entries are `TRUE`	Section 4.1.2, p. 61
`&&, &, ||, |, !`	logical operators	Section 4.1.3, p. 63
`which`	Determines indexes of `TRUE`s	Section 4.1.5, p. 67
`" "`	Creates a character string	Section 4.2.1, p. 71
`nchar`	Gets number of characters in a string	Section 4.2.1, p. 71
`cat`	Concatenates strings (no return)	Section 4.2.2, p. 72
`paste`	Pastes strings (returns a string)	Section 4.2.2, p. 72
`\`	String escape	Section 4.2.3, p. 74
`substr`	Subsets a string	Section 4.2.4, p. 76
`sub, gsub`	String matching and replacement	Section 4.2.4, p. 76
`factor`	Creates a factor vector	Section 4.3.1, p. 78
`levels`	Gets levels of a factor	Section 4.3.1, p. 79
`cut`	Creates factor from continuum	Section 4.3.3, p. 83

5

LISTS AND DATA FRAMES

Vectors, matrices, and arrays are efficient and convenient data storage structures in R, but they have one distinct limitation: each instance of these structures can store data or observations of only one type. In this chapter, we'll explore two more data structures, lists and data frames, which can store multiple types of values at once.

5.1 Lists of Objects

The *list* is an incredibly useful data structure in R. It can be used to group together any mix of R structures and objects of any type. A single list could contain a numeric matrix, a logical array, a single character string, and a factor object. You can even have yet another list as a component of a list. In this section, you'll see how to create, modify, and access components of these flexible structures.

5.1.1 Definition and Component Access

Creating a list is much like creating a vector. You supply the elements that you want to include to the `list` function, separated by commas.

```
R> foo <- list(matrix(data=1:4,nrow=2,ncol=2),c(T,F,T,T),"hello")
R> foo
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1]  TRUE FALSE  TRUE  TRUE

[[3]]
[1] "hello"
```

In the list `foo`, we've stored a 2×2 numeric matrix, a logical vector, and a character string. These are printed in the order they were supplied to `list`. Just as with vectors, you can use the `length` function to check the number of components in a list.

```
R> length(x=foo)
[1] 3
```

You can also retrieve components from a list using indexes, which are entered in double square brackets.

```
R> foo[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
R> foo[[3]]
[1] "hello"
```

This action is known as a *member reference*. Once you retrieve a component this way, you can treat it just like a stand-alone object in the workspace; there is nothing special that needs to be done.

```
R> foo[[1]] + 5.5
     [,1] [,2]
[1,]  6.5  8.5
[2,]  7.5  9.5
R> foo[[1]][1,2]
[1] 3
R> foo[[1]][2,]
[1] 2 4
```

```
R> cat(foo[[3]],"you!")
hello you!
```

Of course, these members are only actually overwritten in `foo` if you use the assignment operator.

```
R> foo[[3]]
[1] "hello"
R> foo[[3]] <- paste(foo[[3]],"you!")
R> foo
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1]  TRUE FALSE  TRUE  TRUE

[[3]]
[1] "hello you!"
```

Suppose now you want to access the second and third components of `foo` and store them as one object. Your first instinct might be to try something like this:

```
R> foo[[c(2,3)]]
[1] TRUE
```

But R has not done what you wanted. Instead, it returned the third element of the second component. Fortunately, member referencing with the double square brackets is not the only way to access components of a list. You can also use single square bracket notation. This is referred to as *list slicing*, and it lets you select multiple list items at once.

```
R> bar <- foo[c(2,3)]
R> bar
[[1]]
[1]  TRUE FALSE  TRUE  TRUE

[[2]]
[1] "hello you!"
```

Note that the result `bar` is itself a list with the two components stored in the order in which they were requested.

5.1.2 Naming

You can *name* list components to make the elements more recognizable and easy to work with. Just like the information stored about factor levels (as you saw in Section 4.3.1), a name is an R *attribute*.

Let's start by adding names to the list `foo` from earlier.

```
R> names(foo) <- c("mymatrix","mylogicals","mystring")
R> foo
$mymatrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$mylogicals
[1]  TRUE FALSE  TRUE  TRUE

$mystring
[1] "hello you!"
```

This has changed how the object is printed to the console. Where earlier it printed `[[1]]`, `[[2]]`, and `[[3]]` before each component, now it prints the names you specified: `$mymatrix`, `$mylogicals`, and `$mystring`. You can now perform member referencing using these names and the *dollar* operator, rather than the double square brackets.

```
R> foo$mymatrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

This style of extraction is the same as calling `foo[[1]]`. In fact, even when an object is named, you can still use the numeric index to obtain a member.

```
R> foo[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Subsetting named members also works the same way as earlier.

```
R> all(foo$mymatrix[,2]==foo[[1]][,2])
[1] TRUE
```

This confirms (using the `all` function you saw in Section 4.1.2) that these two ways of extracting the second column of the matrix in `foo` provide an identical result.

To name the components of a list as it's being created, assign a label to each component in the `list` command. Using some components of `foo`, create a new, named list.

```
R> baz <- list(tom=c(foo[[2]],T,T,T,F),dick="g'day mate",harry=foo$mymatrix*2)
R> baz
$tom
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

$dick
[1] "g'day mate"

$harry
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

The object `baz` now contains the three named components `tom`, `dick`, and `harry`.

```
R> names(baz)
[1] "tom"   "dick"  "harry"
```

If you want to rename these members, you can simply assign a character vector of length 3 to `names(baz)`, the same way you did for `foo` earlier.

NOTE *When using the `names` function, the component names are always provided and returned as character strings, in double quotes. However, if you are specifying names when a list is created (inside the `list` function) or using names to extract members with the dollar operator, the names are entered without quotes (in other words, they are* not *given as strings).*

5.1.3 Nesting

As noted earlier, a member of a list can itself be a list. The only complication in nesting lists like this is keeping track of the depth of any given member for subsetting or extraction.

Note that you can add components to any existing list by using the dollar operator and a *new* name. Here's an example using `foo` and `baz` from earlier:

```
R> baz$bobby <- foo
R> baz
$tom
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

$dick
[1] "g'day mate"
```

```
$harry
     [,1] [,2]
[1,]    2    6
[2,]    4    8

$bobby
$bobby$mymatrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$bobby$mylogicals
[1]  TRUE FALSE  TRUE  TRUE

$bobby$mystring
[1] "hello you!"
```

Here we defined a fourth component to the list `baz` called `bobby`. The member `bobby` is assigned the entire list `foo`. As you can see by printing the new `baz`, there are now three components in `bobby`. Naming and indexes are now both layered, and you can use either (or combine them) to retrieve members of the inner list.

```
R> baz$bobby$mylogicals[1:3]
[1]  TRUE FALSE  TRUE
R> baz[[4]][[2]][1:3]
[1]  TRUE FALSE  TRUE
R> baz[[4]]$mylogicals[1:3]
[1]  TRUE FALSE  TRUE
```

These all instruct R to return the first three elements of the logical vector stored as the second component (`[[2]]`, also named `mylogicals`) of the list `bobby`, which in turn is the fourth component of the list `baz`.

This is a complex example, but the general principle is intuitive. As long as you're aware of what is returned at each layer of a subset, you can continue to subset as needed using names and/or numeric indexes. Consider the third line in the previous code. The first layer of the subset is `baz[[4]]`, which is a list with three components. The second layer of subsetting extracts the component `mylogicals` from that list by calling `baz[[4]]$mylogicals`. This component represents a vector of length 4, so the third layer of subsetting retrieves the first three elements of that vector with the line `baz[[4]]$mylogicals[1:3]`.

Lists are often used to return output from various R functions. But they can quickly become rather large objects (in terms of system resources) to store. Where possible (that is, when a data set has only one kind of value), it is therefore generally recommended to use vector, matrix, or array structures to record and store data.

Exercise 5.1

a. Create a list that contains, in this order, a sequence of 20 evenly spaced numbers between −4 and 4; a 3 × 3 matrix of the logical vector `c(F,T,T,T,F,T,T,F,F)` filled column-wise; a character vector with the two strings `"don"` and `"quixote"`; and a factor vector containing the observations `c("LOW","MED","LOW","MED","MED","HIGH")`. Then, do the following:
   i. Extract row elements `2` and `1` of columns `2` and `3`, in that order, of the logical matrix.
   ii. Use `sub` to overwrite `"quixote"` with `"Quixote"` and `"don"` with `"Don"` inside the list. Then, using the newly overwritten list member, concatenate to the console screen the following statement exactly:

```
"Windmills! ATTACK!"
    -\Don Quixote/-
```

   iii. Obtain all values from the sequence between −4 and 4 that are greater than 1.
   iv. Determine `which` values in the factor vector are assigned the `"MED"` level.

b. Create a new list with the factor vector from (a) as a component named `"facs"`; the numeric vector `c(3,2.1,3.3,4,1.5,4.9)` as a component named `"nums"`; and a nested list comprised of the first three members of the list from (a) (use list slicing to obtain this), named `"oldlist"`. Then, do the following:
   i. Extract the elements of `"facs"` that correspond to elements of `"nums"` that are greater than or equal to 3.
   ii. Add a new member to the list named `"flags"`. This member should be a logical vector of length 6, obtained as a twofold repetition of the third column of the logical matrix in the `"oldlist"` component.
   iii. Use `"flags"` and the logical negation operator `!` to extract the entries of `"num"` corresponding to `FALSE`.
   iv. Overwrite the character string vector component of `"oldlist"` with the single character string `"Don Quixote"`.

5.2 Data Frames

A *data frame* is R's most natural way of presenting a data set with one or more variables recorded for multiple subjects. Like lists, data frames have no restriction on the data types of the variables; you can store numeric data, factor data, and so on. The R data frame can be thought of as a list with

some extra rules attached. The most important distinction is that in a data frame (unlike a list), the members must all be vectors of equal length.

The data frame is one of the most important and frequently used tools in R for statistical data analysis. In this section, you'll look at how to create data frames and learn about their general characteristics.

5.2.1 Construction

To create a data frame from scratch, use the `data.frame` function. You supply your data, grouped by variable, as vectors of the same length—the same way you would construct a named list. Consider the following example data set (where the line of code is split for better readability):

```
R> mydata <- data.frame(person=c("Peter","Lois","Meg","Chris","Stewie"),
                        age=c(42,40,17,14,1),
                        sex=factor(c("M","F","F","M","M")))
R> mydata
  person age sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
4  Chris  14   M
5 Stewie   1   M
```

Here, you constructed a data frame with the first name, age in years, and sex of five individuals. The returned object should make it clear why vectors passed to `data.frame` must be of equal length: vectors of differing lengths wouldn't make sense in this context. If you pass vectors of unequal length to `data.frame`, then R will attempt to recycle any shorter vectors to match the longest. Notice that data frames are printed to the console in rows and columns–they look more like a matrix than a named list. This spreadsheet style makes it easy to read and manipulate data sets. Each row in a data frame is called a *record*, and each column is a *variable*.

You can extract portions of the data by specifying row and column index positions (much as with a matrix). Here's an example:

```
R> mydata[2,2]
[1] 40
```

This gives you the element at row 2, column 2—the age of Lois.

```
R> mydata[3:5,3]
[1] F M M
Levels: F M
```

This returns a factor vector with the third, fourth, and fifth row elements of the third column—the sex of Meg, Chris, and Stewie.

```
R> mydata[,c(3,1)]
  sex person
1   M  Peter
2   F   Lois
3   F    Meg
4   M  Chris
5   M Stewie
```

This results in another data frame based on the entire third and first columns (in that order) of `mydata`.

You can also use the names of the vectors that were passed to `data.frame` to access variables even if you do not know their column index positions (which can be useful for large data sets). You use the same dollar operator you used for member referencing named lists.

```
R> mydata$age
[1] 42 40 17 14  1
```

You can subset this returned vector, too:

```
R> mydata$age[2]
[1] 40
```

This returns the same thing as the earlier call of `mydata[2,2]`.

You can report the dimensions of a data frame (the number of records and variables) using the same functions as for matrices (first shown in Section 3.1).

```
R> nrow(mydata)
[1] 5
R> ncol(mydata)
[1] 3
R> dim(mydata)
[1] 5 3
```

The `nrow` function retrieves the number of rows (records), `ncol` retrieves the number of columns, and `dim` retrieves both.

R's default behavior for character vectors passed to `data.frame` is to convert each variable into a factor object. Observe the following:

```
R> mydata$person
[1] Peter  Lois   Meg    Chris  Stewie
Levels: Chris Lois Meg Peter Stewie
```

Notice that this variable has levels that show it's being treated as a factor. But this is not what we intended when we defined `mydata` earlier—we explicitly defined `sex` to be a factor, but we left `person` as a vector of character strings. To prevent this automatic conversion of character strings to factors when using `data.frame`, set the optional argument `stringsAsFactors` to `FALSE` (otherwise, it defaults to `TRUE`). Reconstructing `mydata` with this in place looks like this:

```
R> mydata <- data.frame(person=c("Peter","Lois","Meg","Chris","Stewie"),
                        age=c(42,40,17,14,1),
                        sex=factor(c("M","F","F","M","M")),
                        stringsAsFactors=FALSE)
R> mydata
  person age sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
4  Chris  14   M
5 Stewie   1   M
R> mydata$person
[1] "Peter"  "Lois"   "Meg"    "Chris"  "Stewie"
```

You now have `person` in the desired, nonfactor form.

5.2.2 Adding Data Columns and Combining Data Frames

Say you want to add data to an existing data frame. This could be a set of observations for a new variable (adding to the number of columns), or it could be more records (adding to the number of rows). Once again, you can use some of the functions you've already seen applied to matrices.

Recall the `rbind` and `cbind` functions from Section 3.1, which let us append rows and columns, respectively. These same functions can be used to extend data frames intuitively. For example, suppose you had another record to include in `mydata`: the age and sex of another individual named Brian. The first step is to create a new data frame that contains Brian's information.

```
R> newrecord <- data.frame(person="Brian",age=7,
                           sex=factor("M",levels=levels(mydata$sex)))
R> newrecord
  person age sex
1  Brian   7   M
```

To avoid any confusion, it's important to make sure the variable names and the data types match the data frame you're planning to add this to. Note that for a factor, you can extract the levels of the existing factor variable as shown earlier with `sex`.

Now, you can simply call the following:

```
R> mydata <- rbind(mydata,newrecord)
R> mydata
  person age sex
1  Peter  42   M
2   Lois  40   F
3    Meg  17   F
4  Chris  14   M
5 Stewie   1   M
6  Brian   7   M
```

Using `rbind`, you combined `mydata` with the new record and overwrote `mydata` with the result. Adding a variable to a data frame is also quite straightforward. Let's say you now are given data on the classification of how funny these six individuals are, defined as a "degree of funniness." The degree of funniness can take three possible values: Low, Medium, and High. From Section 4.3.2, you know that this variable is best represented in R as a factor. Suppose Peter, Lois, and Stewie have a high degree of funniness, Chris and Brian have a medium degree of funniness, and Meg has a low degree of funniness. In R, you'd have a factor vector like this:

```
R> funny <- c("High","High","Low","Med","High","Med")
R> funny <- factor(x=funny,levels=c("Low","Med","High"))
R> funny
[1] High High Low  Med  High Med
Levels: Low Med High
```

The first line creates the basic character vector as `funny`, and the second line overwrites `funny` by turning it into an ordered factor. Note that the order of these elements must correspond to the records in your data frame. Now, you can simply use `cbind` to append this factor vector as a column to the existing `mydata`.

```
R> mydata <- cbind(mydata,funny)
R> mydata
  person age sex funny
1  Peter  42   M  High
2   Lois  40   F  High
3    Meg  17   F   Low
4  Chris  14   M   Med
5 Stewie   1   M  High
6  Brian   7   M   Med
```

The `rbind` and `cbind` functions aren't the only ways to extend a data frame. One useful alternative for adding a variable is to use the dollar operator (much like adding a new member to a named list, as in Section 5.1.3). Suppose now you want to add another variable to `mydata` by including a

column with the age of the individuals in months, not years, calling this new variable `age.mon`.

```
R> mydata$age.mon <- mydata$age*12
R> mydata
  person age sex funny age.mon
1  Peter  42   M  High     504
2   Lois  40   F  High     480
3    Meg  17   F   Low     204
4  Chris  14   M   Med     168
5 Stewie   1   M  High      12
6  Brian   7   M   Med      84
```

This creates a new `age.mon` column with the dollar operator and at the same time assigns it the vector of ages in years (already stored as `age`) multiplied by 12.

5.2.3 Logical Record Subsets

In Section 4.1.5, you saw how to use logical flag vectors to subset data structures. This is a particularly useful technique with data frames, where you'll often want to examine a subset of entries that meet certain criteria. For example, when working with data from a clinical drug trial, a researcher might want to examine the results for just male participants and compare them to the results for females. Or the researcher might want to look at the characteristics of individuals who responded most positively to the drug.

Let's continue to work with `mydata` as it was left previously. Say you want to examine all records corresponding to males. From Section 4.3.1, you know that the following line will identify the relevant positions in the `sex` factor vector:

```
R> mydata$sex=="M"
[1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE
```

This flags the male records. You can use this with the matrix-like syntax you saw in Section 5.2.1 to get the male-only subset.

```
R> mydata[mydata$sex=="M",]
  person age sex funny age.mon
1  Peter  42   M  High     504
4  Chris  14   M   Med     168
5 Stewie   1   M  High      12
6  Brian   7   M   Med      84
```

You can use the same behavior to pick and choose which variables to return in the subset. For example, since you know you are selecting the males only, you could omit `sex` from the result using a negative numeric index in the column dimension.

```
R> mydata[mydata$sex=="M",-3]
  person age funny age.mon
1  Peter  42  High     504
4  Chris  14   Med     168
5 Stewie   1  High      12
6  Brian   7   Med      84
```

If you don't have the column number or if you want to have more control over the returned columns, you can use a character vector of variable names instead.

```
R> mydata[mydata$sex=="M",c("person","age","funny","age.mon")]
  person age funny age.mon
1  Peter  42  High     504
4  Chris  14   Med     168
5 Stewie   1  High      12
6  Brian   7   Med      84
```

The logical conditions you use to subset a data frame can be as simple or as complicated as you need them to be. The logical flag vector you place in the square brackets just has to match the number of records in the data frame. Let's extract from `mydata` the full records for individuals who are female or have a high degree of funniness.

```
R> mydata[mydata$sex=="F"|mydata$funny=="High",]
  person age sex funny age.mon
1  Peter  42   M  High     504
2   Lois  40   F  High     480
3    Meg  17   F   Low     204
5 Stewie   1   M  High      12
```

Sometimes, asking for a subset will yield no records. In this case, R returns a data frame with zero rows, which looks like this:

```
R> mydata[mydata$age>45,]
[1] person  age     sex     funny   age.mon
<0 rows> (or 0-length row.names)
```

In this example, no records are returned from `mydata` because there are no individuals older than 45. To check whether a subset will contain any records, you can use `nrow` on the result—should this be equal to zero, then no records have satisfied the specified condition(s). This type of check is where the programming structure known as an *if statement* plays a natural role, and you'll explore this in depth in Chapter 10.

Exercise 5.2

a. Create and store the following data frame as `dframe` in your R workspace:

person	sex	funny
Stan	M	High
Francine	F	Med
Steve	M	Low
Roger	M	High
Hayley	F	Med
Klaus	M	Med

   The variables `person`, `sex`, and `funny` should be identical in nature to the variables in the `mydata` object studied throughout Section 5.2. That is, `person` should be a character vector, `sex` should be an unordered factor with levels `F` and `M`, and `funny` should be an unordered factor with levels `Low`, `Med`, and `High`.

b. Stan and Francine are 41 years old, Steve is 15, Hayley is 21, and Klaus is 60. Roger is extremely old—1,600 years. Append these data as a new numeric column variable in `dframe` called `age`.

c. Use your knowledge of reordering the column variables based on column index positions to overwrite `dframe`, bringing it in line with `mydata`. That is, the first column should be `person`, the second column `age`, the third column `sex`, and the fourth column `funny`.

d. Turn your attention to `mydata` as it was left following inclusion of the `age.mon` variable in Section 5.2.2. Create a new version of `mydata` called `mydata2` by deleting the `age.mon` column.

e. Now, combine `mydata2` with `dframe`, calling the resulting object `mydataframe`.

f. Write a single line of code that will extract from `mydataframe` just the names and ages of any records where the individual is female and has a level of funniness equal to `Med` *or* `High`.

g. Use your knowledge of handling character strings in R to extract all records from `mydataframe` that correspond to people whose names start with *S*. Hint: Recall `substr` from Section 4.2.4 (note that `substr` can be applied to a vector of multiple character strings).

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`list`	Create a list	Section 5.1.1, p. 88
`[[ ]]`	Unnamed member reference	Section 5.1.1, p. 88
`[ ]`	List slicing (multiple members)	Section 5.1.1, p. 89
`$`	Get named member/variable	Section 5.1.2, p. 90
`data.frame`	Create a data frame	Section 5.2.1, p. 94
`[ , ]`	Extract data frame row/columns	Section 5.2.1, p. 94

6

SPECIAL VALUES, CLASSES, AND COERCION

You've now learned about numerical values, logicals, character strings, and factors, as well as their unique properties and applications. Now you'll look at some special values in R that are not as well-defined. You'll see how they might come about and how to handle and test for them. This will lead to a more formal discussion of how data types are treated in R, as well as some general concepts about object class.

6.1 Some Special Values

Many situations in R call for special values. For example, when a data set has missing observations or when a practically infinite number is calculated, the software has some unique terms that it reserves for these situations. These special values can be used to mark abnormal or missing values in vectors, arrays, or other data structures (even though vectors and arrays allow for only one type of value in a single object).

6.1.1 Infinity

In Section 2.1, I mentioned that R imposes limits on how extreme a number can be before the software cannot reliably represent it. When a number is too large for R to represent, the value is deemed to be *infinite*. Of course, the mathematical concept of infinity (or ∞) does not correspond to a specific number—R simply has to define an extreme cutoff point since it's dependent upon limited system resources. The precise cutoff value varies from system to system and is governed in part by the amount of memory R has access to. Although the software may formally identify a value as infinite, it should really be thought of as a kind of pseudo-infinite stand-in. These values are represented with the special, case-sensitive term `Inf` (no double quotes—even though it uses alphabetic characters, it is not a character string).

```
R> foo <- Inf
R> foo
[1] Inf
R> bar <- c(3401,Inf,3.1,-555,Inf,43)
R> bar
[1] 3401.0    Inf    3.1 -555.0    Inf   43.0
R> baz <- 90000^100
R> baz
[1] Inf
```

Here, you've defined an object `foo` that is a single instance of an infinite value. There's also a numeric vector with two infinite elements, and then a number that R deems infinite is produced by raising 90,000 to a power of 100. Note that infinite values can be associated only with numeric vectors.

R can also represent negative infinity as `-Inf`.

```
R> qux <- c(-42,565,-Inf,-Inf,Inf,-45632.3)
R> qux
[1]    -42.0    565.0     -Inf     -Inf      Inf -45632.3
```

This creates a vector with two negative-infinite values and one positive-infinite value.

Though infinity does not represent any specific value, to a certain extent you can still perform mathematical operations on infinite values in R. For example, multiplying `Inf` by any negative value will result in `-Inf`.

```
R> Inf*-9
[1] -Inf
```

If you add to or multiply infinity, you also get infinity as a result if the calculation would otherwise yield a more extreme value.

```
R> Inf+1
[1] Inf
R> 4*-Inf
```

```
[1] -Inf
R> -45.2-Inf
[1] -Inf
R> Inf-45.2
[1] Inf
R> Inf+Inf
[1] Inf
R> Inf/23
[1] Inf
```

Zero and infinity go hand in hand when it comes to division. Any (finite) numeric value divided by infinity, positive or negative, will result in zero.

```
R> -59/Inf
[1] 0
R> -59/-Inf
[1] 0
```

Though it is not mathematically defined, note that in R, any nonzero value divided by zero itself will result in infinity (positive or negative depending on the sign of the numerator).

```
R> -59/0
[1] -Inf
R> 59/0
[1] Inf
R> Inf/0
[1] Inf
```

Often, you'll simply want to detect infinite values in a data structure. The functions `is.infinite` and `is.finite` take in a collection of values, typically a vector, and return for each element a logical value answering the question posed. Here's an example using `qux` from earlier:

```
R> qux
[1]    -42.0    565.0     -Inf     -Inf      Inf -45632.3
R> is.infinite(x=qux)
[1] FALSE FALSE  TRUE  TRUE  TRUE FALSE
R> is.finite(x=qux)
[1]  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

Note that these functions do not distinguish between positive or negative infinity, and the result of `is.finite` will always be the opposite (the negation) of the result of `is.infinite`.

Finally, relational operators function as you might expect.

```
R> -Inf<Inf
[1] TRUE
R> Inf>Inf
[1] FALSE
R> qux==Inf
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
R> qux==-Inf
[1] FALSE FALSE  TRUE  TRUE FALSE FALSE
```

Here, the first line confirms that `-Inf` is indeed treated as less than `Inf`, and the second line shows that `Inf` is not greater than `Inf`. The third and fourth lines, again using `qux`, test for equality (which is a useful way to distinguish between positive and negative infinity if desired).

6.1.2 NaN

In some situations, it is impossible to express the result of a calculation using a number, `Inf`, or `-Inf`. These difficult-to-quantify special values are labeled `NaN` in R, which stands for *Not a Number*.

As with infinite values, `NaN` values are associated only with numeric observations. You can define/include a `NaN` value directly, but this is rarely the way it is encountered.

```
R> foo <- NaN
R> foo
[1] NaN
R> bar <- c(NaN,54.3,-2,NaN,90094.123,-Inf,55)
R> bar
[1]      NaN    54.30    -2.00      NaN 90094.12     -Inf    55.00
```

Typically, `NaN` is the unintended result of running code that attempts a calculation that's impossible to perform with the specified values.

In Section 6.1.1, you saw that adding or subtracting from `Inf` or `-Inf`, to produce a "more extreme" result, will simply result again in `Inf` or `-Inf`. However, if you attempt to cancel representations of infinity in any way, the result will be `NaN`.

```
R> -Inf+Inf
[1] NaN
R> Inf/Inf
[1] NaN
```

Here, the first line won't result in zero because positive and negative infinity can't be interpreted in that numerical sense, so you get `NaN` as a result. Dividing `Inf` by itself is similarly indeterminate and yields the same result. In addition, although you saw earlier that a nonzero value divided by

zero will result in positive or negative infinity, NaN results when *zero* is divided by zero.

```
R> 0/0
[1] NaN
```

Note that any mathematical operation involving NaN will simply result in NaN.

```
R> NaN+1
[1] NaN
R> 2+6*(4-4)/0
[1] NaN
R> 3.5^(-Inf/Inf)
[1] NaN
```

In the first line, adding 1 to "not a number" is still NaN. In the second line, you obtain NaN from the (4-4)/0, which is clearly 0/0, so the result is also NaN. In the third line, NaN results from -Inf/Inf, so the result of the remaining calculation is again NaN. This begins to give you an idea of how NaN (or infinite values) might unintentionally crop up. If you have a function where various values are passed to a fixed calculation and you don't take care to prevent, for example, 0/0 from occurring in a given instance, then the code will return NaN.

Like with Inf, a special function (is.nan) is used to detect the presence of NaN values. Unlike infinite values, however, relational operators cannot be used with NaN. Here's an example using bar, which was defined earlier:

```
R> bar
[1]      NaN    54.30    -2.00      NaN 90094.12     -Inf    55.00
R> is.nan(x=bar)
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
R> !is.nan(x=bar)
[1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
R> is.nan(x=bar)|is.infinite(x=bar)
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
R> bar[-(which(is.nan(x=bar)|is.infinite(x=bar)))]
[1]    54.30    -2.00 90094.12    55.00
```

Using the is.nan function on bar flags the two NaN positions as TRUE. In the second example, you use the negation operator ! to flag the positions where the elements are NOT NaN. Using the element-wise OR, | (see Section 4.1.3), you then identify elements that are either NaN OR infinite. Finally, the last line uses which to convert these logical values into numeric index positions so that you can remove them with negative indexes in square brackets (see Section 4.1.5 for a refresher on using which).

You can find more details on the functionality and behavior of NaN and Inf in the R help file by entering ?Inf at the prompt.

Exercise 6.1

a. Store the following vector:

```
foo <- c(13563,-14156,-14319,16981,12921,11979,9568,8833,-12968,
         8133)
```

Then, do the following:

i. Output all elements of `foo` that, when raised to a power of 75, are not infinite.
ii. Return the elements of `foo` deleting those that, when raised to a power of 75, are negative infinity.

b. Store the following 3×4 matrix as the object `bar`:

$$\begin{bmatrix} 77875.40 & 27551.45 & 23764.30 & -36478.88 \\ -35466.25 & -73333.85 & 36599.69 & -70585.69 \\ -39803.81 & 55976.34 & 76694.82 & 47032.00 \end{bmatrix}$$

Now, do the following:

i. Identify the coordinate-specific indexes of the entries of `bar` that are `NaN` when you raise `bar` to a power of 65 and divide by infinity.
ii. Return the values in `bar` that are *not* `NaN` when `bar` is raised to a power of 67 and infinity is added to the result. Confirm this is identical to identifying those values in `bar` that, when raised to a power of 67, are not equal to negative infinity.
iii. Identify those values in `bar` that are either negative infinity *or* finite when you raise `bar` to a power of 67.

6.1.3 NA

In statistical analyses, data sets often contain missing values. For example, someone filling out a questionnaire may not respond to a particular item, or a researcher may record some observations from an experiment incorrectly. Identifying and handling missing values is important so that you can still use the rest of the associated data that is present. R provides a standard special term to represent missing values, `NA`, which reads as *Not Available*.

`NA` entries should be distinguished from `NaN` entries. Whereas `NaN` is used only with respect to numeric operations, missing values can occur for any type of observation. As such, `NA`s can exist in both numeric and non-numeric settings. Here's an example:

```
R> foo <- c("character","a",NA,"with","string",NA)
R> foo
[1] "character" "a"         NA          "with"      "string"    NA
```

```
R> bar <- factor(c("blue",NA,NA,"blue","green","blue",NA,"red","red",NA,
                   "green"))
R> bar
 [1] blue  <NA>  <NA>  blue  green blue  <NA>  red   red   <NA>  green
Levels: blue green red
R> baz <- matrix(c(1:3,NA,5,6,NA,8,NA),nrow=3,ncol=3)
R> baz
     [,1] [,2] [,3]
[1,]    1   NA   NA
[2,]    2    5    8
[3,]    3    6   NA
```

The object `foo` is a character vector with entries 3 and 6 missing; `bar` is a factor vector of length 11 with elements 2, 3, 7, and 10 missing; and `baz` is a numeric matrix with row 1, columns 2 and 3, and row 3, column 3, elements missing. In the factor vector, note that the `NA`s are printed as `<NA>`. This is to differentiate between bona fide levels of the factor and the missing observations (in other words, `NA` does not become one of the levels).

Like the other special values so far, you can identify `NA` elements using the function `is.na`. This is often useful for removing or replacing `NA` values. Consider the following numeric vector:

```
R> qux <- c(NA,5.89,Inf,NA,9.43,-2.35,NaN,2.10,-8.53,-7.58,NA,-4.58,2.01,NaN)
R> qux
 [1]    NA  5.89   Inf    NA  9.43 -2.35   NaN  2.10 -8.53 -7.58    NA -4.58
[13]  2.01   NaN
```

This vector has a total of 14 entries, including `NA`, `NaN`, and `Inf`.

```
R> is.na(x=qux)
 [1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
[13] FALSE  TRUE
```

As you can see, `is.na` flags the corresponding `NA` entries in `qux` as `TRUE`. But this is not all—note that it also flags elements 7 and 14, which are `NaN`, not `NA`. Strictly speaking, `NA` and `NaN` are different entities, but numerically they are practically the same since there is almost nothing you can do with either value. Using `is.na` labels both as `TRUE`, allowing the user to remove or recode both at the same time.

If you want to identify `NA` and `NaN` entries separately, you can use `is.nan` in conjunction with logical operators. Here's an example:

```
R> which(x=is.nan(x=qux))
[1]  7 14
```

This identifies the index positions whose elements are specifically `NaN`.

```
R> which(x=(is.na(x=qux)&!is.nan(x=qux)))
[1]  1  4 11
```

This identifies the element indexes for only the NA entries (by checking for entries where is.na is TRUE AND where NOT is.nan is TRUE).

After locating the offending elements, you could use negative indexes in square brackets to remove them, though R offers a more direct option. The function na.omit will take a structure and delete all NAs from it; like is.na, na.omit will also apply to NaNs if the elements are numeric.

```
R> quux <- na.omit(object=qux)
R> quux
[1]  5.89   Inf  9.43 -2.35  2.10 -8.53 -7.58 -4.58  2.01
attr(,"na.action")
[1]  1  4  7 11 14
attr(,"class")
[1] "omit"
```

Note that the structure passed to na.omit is given as the argument object and that some additional output is displayed in printing the returned object. These extra details are provided to inform the user that there were elements in the original vector that were removed (in this case, the element positions provided in the attribute na.action). Attributes will be discussed more in Section 6.2.1.

Similar to NaN, arithmetic calculations with an NA result in NA. Using relational operators with either NaN or NA will also result in NA.

```
R> 3+2.1*NA-4
[1] NA
R> 3*c(1,2,NA,NA,NaN,6)
[1]   3   6  NA  NA NaN  18
R> NA>76
[1] NA
R> 76>NaN
[1] NA
```

You can find more details on the usage and finer technicalities of NA values by typing ?NA.

6.1.4 NULL

The final special value in R that you'll consider is the *null* value, written as NULL. This value is often used to explicitly define an empty entity, which is quite different from a "missing" entity specified with NA. An instance of NA clearly denotes an existing position that can be accessed and/or overwritten if necessary—not so for NULL. You can see an indication of this if you compare the assignment of an NA with the assignment of a NULL.

```
R> foo <- NULL
R> foo
NULL
R> bar <- NA
R> bar
[1] NA
```

Note that `bar`, the `NA` object, is printed with an index position `[1]`. This suggests you have a measurable vector with a single element. In contrast, `foo` was explicitly instructed to be empty with `NULL`. Printing this object does not provide a position index because there is no position to access.

This interpretation of `NULL` also applies to vectors that have other well-defined items. Consider the following two lines of code:

```
R> c(2,4,NA,8)
[1]  2  4 NA  8
R> c(2,4,NULL,8)
[1] 2 4 8
```

The first line creates a vector of length 4, with the third position coded as `NA`. The second line creates a similar vector but using `NULL` instead of `NA`. The result is a vector with a length of only 3. That's because `NULL` cannot take up a position in the vector. As such, it makes no sense to assign `NULL` to multiple positions in a vector (or any other structure). Again, here's an example:

```
R> c(NA,NA,NA)
[1] NA NA NA
R> c(NULL,NULL,NULL)
NULL
```

The first line can be interpreted as "three possible slots with unrecorded observations." The second line simply provides "emptiness three times," which is interpreted as one single, unsubsettable, empty object.

At this point, you might wonder why there is even a need for `NULL`. If something is empty and doesn't exist, why define it in the first place? The answer lies in the need to be able to explicitly state and/or check whether a certain object has been defined. This occurs often when calling functions in R. For example, when a function contains optional arguments, this function must check internally which arguments have been supplied and which are missing, or empty. The `NULL` value is a useful and flexible tool for such a step.

The `is.null` function is used to check whether something is explicitly `NULL`. Suppose you have a function with an optional argument called `opt.arg`. Further assume `opt.arg`, if supplied, should be a character vector of length 3. Let's say a user calls this function with the following.

```
R> opt.arg <- c("string1","string2","string3")
```

Now consider what happens if you try to check whether the argument was supplied using NA. You might think to call this:

```
R> is.na(x=opt.arg)
[1] FALSE FALSE FALSE
```

The position-specific nature of NA means that this check is element-wise and returns an answer for each value in opt.arg. This is problematic because you want only a single answer—is opt.arg empty or is it supplied? This is when NULL comes to the party.

```
R> is.null(x=opt.arg)
[1] FALSE
```

Quite clearly opt.arg is not empty, and the function can proceed as necessary. If the argument is empty, using NULL over NA for the check is again better for these purposes.

```
R> opt.arg <- c(NA,NA,NA)
R> is.na(x=opt.arg)
[1] TRUE TRUE TRUE

R> opt.arg <- c(NULL,NULL,NULL)
R> is.null(x=opt.arg)
[1] TRUE
```

As noted earlier, filling a vector with NULL is not usual practice; it's done here just for illustration. But usage of NULL is far from specific to this particular example. It's commonly used throughout both ready-to-use and user-contributed functionality in R.

The empty NULL has an interesting effect if it is included in arithmetic or relational comparisons.

```
R> NULL+53
numeric(0)
R> 53<=NULL
logical(0)
```

Unlike with the use of NA in similar operations, the output of these operations is clearly not just another NULL. Here, the result is an "empty" vector of a certain type (determined by the nature of the operation attempted), which is expressed as shown earlier. NULL typically dominates any arithmetic, even if it includes other special values.

```
R> NaN-NULL+NA/Inf
numeric(0)
```

NULL also occurs naturally when examining lists and data frames. Consider the list foo.

```
R> foo <- list(member1=c(33,1,5.2,7),member2="NA or NULL?")
R> foo
$member1
[1] 33.0  1.0  5.2  7.0

$member2
[1] "NA or NULL?"
```

Clearly there is no member called member3. Attempting to access a member in foo with this name results in NULL.

```
R> foo$member3
NULL
```

This signals that a member called member3 in foo is empty. As such, it can be filled with whatever you want.

```
R> foo$member3 <- NA
R> foo
$member1
[1] 33.0  1.0  5.2  7.0

$member2
[1] "NA or NULL?"

$member3
[1] NA
```

The same principle applies when querying a data frame for a nonexistent variable using the dollar operator (as in Section 5.2.2).

For more technical details on how NULL and is.null are treated by R, see the help file accessed by ?NULL.

Exercise 6.2

a. Consider the following line of code:

```
foo <- c(4.3,2.2,NULL,2.4,NaN,3.3,3.1,NULL,3.4,NA)
```

Decide yourself which of the following statements are true and which are false and then use R to confirm:

i. The length of `foo` is 8.
ii. Calling `which(x=is.na(x=foo))` will *not* result in `4` and `8`.
iii. Checking `is.null(x=foo)` will provide you with the locations of the two `NULL` values that are present.
iv. Executing `is.na(x=foo[8])+4/NULL` will *not* result in `NA`.

b. Create and store a list containing a single member: the vector `c(7,7,NA,3,NA,1,1,5,NA)`. Then, do the following:

i. Name the member `"alpha"`.
ii. Formally confirm using the appropriate logical-valued function that the list does not have a member with the name `"beta"`.
iii. Incorporate into the list a new member called `beta`, which is the vector obtained by identifying the index positions of `alpha` that are `NA`.

6.2 Understanding Types, Classes, and Coercion

By now, you've examined many of the fundamental features in the R language for representing, storing, and handling data. In this section, you'll examine more closely how to formally distinguish between different kinds of values and structures and look at some simple examples of conversion from one type to another. The aim is to keep this discussion, which concerns a rather complicated and technical aspect of R, relatively short. Nevertheless, some familiarity is necessary with the way R describes objects and the associated terminology.

6.2.1 Attributes

For each R object you create, there exists additional information about the nature of the object itself. This additional information is referred to as the object's *attributes*. You've see a few attributes already. In Section 3.1, you identified the dimensions of a matrix using `dim`. In Section 4.3.1, you used `levels` to get the levels of a factor. It was also noted that `names` can get the member names of a list in Section 5.1.2, and in Section 6.1.3, that an attribute annotates the result of applying `na.omit`.

You can think of attributes as either *explicit* or *implicit*. Loosely speaking, explicit attributes are immediately visible to the user, while implicit attributes are determined by R internally. You can print explicit attributes for a given object with the `attributes` function, which takes any object directly and returns a named list. Consider, for example, the following 3×3 matrix.

```
R> foo <- matrix(data=1:9,nrow=3,ncol=3)
R> foo
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

R> attributes(foo)
$dim
[1] 3 3
```

Here, calling `attributes` returns a list with one member: `dim`. Of course, you can retrieve the contents of `dim` with `attributes(foo)$dim`, but if you know the name of an attribute, you can also use `attr`:

```
R> attr(x=foo,which="dim")
[1] 3 3
```

This abbreviated function takes the object in as `x` and the name of the desired attribute as `which`. Recall that names are specified as character strings in R. To make things even more convenient, the most common attributes have their own functions (usually named after the attribute) to access the corresponding value. For the dimensions of a matrix, you've already seen the function `dim`.

```
R> dim(foo)
[1] 3 3
```

These attribute-specific functions are useful because they also allow access to implicit attributes (see, for example, the notes on `class` coming up in Section 6.2.2). Both `names` and `levels`, functions mentioned earlier, also fit into this category.

Explicit attributes are often optional; if they aren't specified, they are `NULL`. For example, an optional argument to the `matrix` function, `dimnames`, can take in a list to annotate the row and columns of the matrix being created. This list should be made up of two members, both character vectors of the appropriate lengths—the first giving row names and the second giving column names. Let's define the matrix `bar` as follows:

```
R> bar <- matrix(data=1:9,nrow=3,ncol=3,dimnames=list(c("A","B","C"),
                 c("D","E","F")))
```

```
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Dimension names are attributes, so it should come as little surprise that the `dimnames` appear when you call `attributes(bar)`.

```
R> attributes(bar)
$dim
[1] 3 3

$dimnames
$dimnames[[1]]
[1] "A" "B" "C"

$dimnames[[2]]
[1] "D" "E" "F"
```

Note that `dimnames` is itself a list, nested inside the larger attributes list. Again, to extract the values of this attribute, you can use list member referencing, you can use `attr` as shown earlier, or you can use the attribute-specific function.

```
R> dimnames(bar)
[[1]]
[1] "A" "B" "C"

[[2]]
[1] "D" "E" "F"
```

Some attributes can be modified after an object has been created (as you saw already in Section 5.1.2, where you renamed members of a list). Here, to make `foo` match `bar` exactly, you can give `foo` some `dimnames` by assigning them to the attribute-specific function:

```
R> dimnames(foo) <- list(c("A","B","C"),c("D","E","F"))
R> foo
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Though I have used matrices in the discussion here, optional attributes for other objects in R are treated the same way. Furthermore, attributes are not limited to things such as dimensions, names, and levels, and they are not limited to built-in R objects. A user can define a new type of object that

has its own attributes and attribute-specific functions. Their role, however, should remain the same in any situation: to provide descriptive data about the object under scrutiny.

6.2.2 Object Class

One of the most useful attributes for describing an entity in R is the object's *class*. Every object you create is identified, either implicitly or explicitly, with at least one class. In the parlance of *object-oriented programming*—where entities are stored as objects and have *methods* that act upon them—this "identification" is quite important and is formally referred to as *inheritance* (a term that references the adoption of features of a specific, recognized type of object).

NOTE *This section will focus on the most common classing structure used in R, called* S3. *There is another structure,* S4, *which is essentially a more formal set of rules for the identification and treatment of different objects. For most practical intents and certainly for beginners, it suffices to understand and use S3. You can find further details in R's online documentation.*

Explicit classing is common in situations where you have user-defined object structures or an object such as a factor vector or data frame where other attributes play an important part in the handling of the object itself (for example, level labels of a factor vector, or variable names in a data frame, are modifiable attributes that play a primary role in accessing the observations of each object). Elementary R objects such as vectors, matrices, and arrays, on the other hand, are implicitly classed (which means the class is not identified with the `attributes` examined earlier). The class of a given object can always be retrieved using the attribute-specific function `class`.

Let's create some simple vectors to use as examples.

```
R> num.vec1 <- 1:4
R> num.vec1
[1] 1 2 3 4
R> num.vec2 <- seq(from=1,to=4,length=6)
R> num.vec2
[1] 1.0 1.6 2.2 2.8 3.4 4.0
R> char.vec <- c("a","few","strings","here")
R> char.vec
[1] "a"       "few"     "strings" "here"
R> logic.vec <- c(T,F,F,F,T,F,T,T)
R> logic.vec
[1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE
R> fac.vec <- factor(c("Blue","Blue","Green","Red","Green","Yellow"))
R> fac.vec
[1] Blue   Blue   Green  Red    Green  Yellow
Levels: Blue Green Red Yellow
```

You can pass any object to the `class` function, and it returns a character vector as output. Here are examples using the vectors just created:

```
R> class(num.vec1)
[1] "integer"
R> class(num.vec2)
[1] "numeric"
R> class(char.vec)
[1] "character"
R> class(logic.vec)
[1] "logical"
R> class(fac.vec)
[1] "factor"
```

The output from using `class` on the character vector, the logical vector, and the factor vector simply match the kind of data that has been stored. The output from the number vectors is a little more intricate, however. So far, I have referred to any object with an arithmetically valid set of numbers as "numeric." If all the numbers stored in a vector are whole, then R identifies the vector as `"integer"`. Numbers with decimal places (called *floating-point* numbers), on the other hand, are identified as `"numeric"`. This distinction is necessary because some specific tasks strictly require integers, not floating-point numbers. Colloquially, I'll continue to refer to both types as "numeric" (and in fact, the `is.numeric` function will return `TRUE` for both integer and floating-point structures, as you'll see in Section 6.2.3).

As mentioned earlier, R's classes are essentially designed to facilitate object-oriented programming. As such, `class` (in contrast to its behavior for stand-alone vectors) usually reports on the nature of the data *structure*, rather than the type of data, that's stored. Let's try it on some matrices.

```
R> num.mat1 <- matrix(data=num.vec1,nrow=2,ncol=2)
R> num.mat1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
R> num.mat2 <- matrix(data=num.vec2,nrow=2,ncol=3)
R> num.mat2
     [,1] [,2] [,3]
[1,]  1.0  2.2  3.4
[2,]  1.6  2.8  4.0
R> char.mat <- matrix(data=char.vec,nrow=2,ncol=2)
R> char.mat
     [,1]  [,2]
[1,] "a"   "strings"
[2,] "few" "here"
R> logic.mat <- matrix(data=logic.vec,nrow=4,ncol=2)
R> logic.mat
      [,1]  [,2]
```

```
[1,]  TRUE  TRUE
[2,] FALSE FALSE
[3,] FALSE  TRUE
[4,] FALSE  TRUE
```

Note from Section 4.3.1 that factors are used only in vector form, so `fac.vec` is not included here. Now check these matrices with `class`.

```
R> class(num.mat1)
[1] "matrix"
R> class(num.mat2)
[1] "matrix"
R> class(char.mat)
[1] "matrix"
R> class(logic.mat)
[1] "matrix"
```

You see that regardless of the data type, `class` reports the structure of the object itself—all matrices. The same is true for other object structures, like arrays, lists, and data frames.

Earlier, I noted that the class of an object is a character vector with a length of *at least one*. Certain objects will have multiple classes. A variant on a standard form of an object (for example, an ordered factor vector) will inherit the usual factor class and also contain an additional class. Both are returned if you use the `class` function.

```
R> ordfac.vec <- factor(x=c("Small","Large","Large","Regular","Small"),
                        levels=c("Small","Regular","Large"),
                        ordered=TRUE)
R> ordfac.vec
[1] Small   Large   Large   Regular Small
Levels: Small < Regular < Large
R> class(ordfac.vec)
[1] "ordered" "factor"
```

In contrast to `fac.vec` from earlier, which was identified as `"factor"` only, the class of `ordfac.vec` has two components. It is still identified as `"factor"`, but it also includes `"ordered"`, which identifies the variant of the `"factor"` class also present in the object. Here, you can think of `"ordered"` as a *subclass* of `"factor"`. In other words, it is a special case that inherits from, and therefore behaves like, a `"factor"`. For further technical details on R subclasses, I recommend Chapter 9 of Matloff (2011).

NOTE *I have focused on the `class` function here because of its direct relevance to the object-oriented programming style exercised in this text, especially in Part II. Some complexities of R's classing rules can be brought out by other, similar functions. For example, the function `typeof` reports the type of data contained within an object, not just for vectors but also for matrices and arrays. Note, however, that the terminology in the*

output of `typeof` *doesn't always match the output of* `class`*. See the help file* `?typeof` *for details on the values it returns.*

To summarize, an object's class is first and foremost a descriptor of the data structure, though for simple vectors, the `class` function reports the type of data stored. If the vector entries are exclusively whole numbers, then R classes the vector as `"integer"`, whereas `"numeric"` is used to label a vector with floating-point numbers.

6.2.3 Is-Dot Object-Checking Functions

Identifying the class of an object is essential for functions that operate on stored objects, especially those that behave differently depending on the class of the object. To retrieve an object's class, you aren't limited to using the `class` functions (or related functions like `typeof`); there are a wide range of *is-dot* functions available that operate on an object to check its class or data type. These functions simply return logical values according to whether the object passed to them "is" what is being asked.

Is-dot functions exist for almost any sensible check you can think of. For example, consider once more the `num.vec1` vector from Section 6.2.2 and the following six checks:

```
R> num.vec1 <- 1:4
R> num.vec1
[1] 1 2 3 4
R> is.integer(num.vec1)
[1] TRUE
R> is.numeric(num.vec1)
[1] TRUE
R> is.matrix(num.vec1)
[1] FALSE
R> is.data.frame(num.vec1)
[1] FALSE
R> is.vector(num.vec1)
[1] TRUE
R> is.logical(num.vec1)
[1] FALSE
```

You see the first, second, and sixth is-dot functions check the kind of data stored in the object, while the others check the structure of the object itself. The results are to be expected: `num.vec1` *is* "integer" (which also means it *is* "numeric"), and it *is* a "vector." It's not a matrix or a data frame, nor is it logical.

Briefly, it's worth noting that these checks use more general categories than the formal classes identified with `class`. Recall that `num.vec1` was identified solely as `"integer"` in Section 6.2.2, but using `is.numeric` here still returns `TRUE`. In this example, the `num.vec1` with integer data is generalized to be `"numeric"`. Similarly, for a data frame, an object of class `"data.frame"` will

return TRUE for is.data.frame *and* is.list because a data frame is intuitively generalized to a list.

There is a difference between the object is-dot functions detailed here and the functions discussed throughout Section 6.1, such as is.na. The functions to check for the special values like NA should thought of as a check for equality; they exist because it is not legal syntax to write something like foo==NA. Those functions from Section 6.1 thus operate in R's element-wise fashion, whereas the object is-dot functions, such as those in the examples here, inspect the object *itself*, returning only a single logical value.

6.2.4 As-Dot Coercion Functions

You've seen different ways to modify an object after it's been created—by accessing and overwriting elements, for example. But what about the structure of the object itself and the type of data contained within?

Converting from one object or data type to another is referred to in computing languages as *coercion*. Like other features of R you've met so far, coercion is performed either implicitly or explicitly. Implicit coercion occurs when an operation is invoked where one or more of the elements involved must be converted to another type in order for the operation to complete. In fact, you've come across this behavior already, in Section 4.1.4, for example. Remember that logical values can be thought of as integers—one for TRUE and zero for FALSE. Implicit coercion of logical values to their numeric counterparts occurs in lines of code like this:

```
R> 1:4+c(T,F,F,T)
[1] 2 2 3 5
```

In this operation, R recognizes that you're attempting an arithmetic calculation with +, so it expects numeric quantities. Since the logical vector is not in this form, the software internally coerces it to ones and zeros before completing the task.

Another frequent example of implicit coercion is when paste and cat are used to glue together character strings, as explored in Section 4.2.2. Non-character entries are automatically coerced to strings before the concatenation takes place. Here's an example:

```
R> foo <- 34
R> bar <- T
R> paste("Definitely foo: ",foo,"; definitely bar: ",bar,".",sep="")
[1] "Definitely foo: 34; definitely bar: TRUE."
```

Here, the integer 34 and the logical T are implicitly coerced to characters since R knows the output of paste must be a string.

In other situations, coercion won't happen automatically and must be carried out by the user. This explicit coercion can be achieved with the *as-dot* functions. Like is-dot functions, as-dot functions exist for most typical

R data types and object classes. The previous two examples can be coerced explicitly, as follows.

```
R> as.numeric(c(T,F,F,T))
[1] 1 0 0 1
R> 1:4+as.numeric(c(T,F,F,T))
[1] 2 2 3 5
R> foo <- 34
R> foo.ch <- as.character(foo)
R> foo.ch
[1] "34"
R> bar <- T
R> bar.ch <- as.character(bar)
R> bar.ch
[1] "TRUE"
R> paste("Definitely foo: ",foo.ch,"; definitely bar: ",bar.ch,".",sep="")
[1] "Definitely foo: 34; definitely bar: TRUE."
```

Coercions are possible in most cases that "make sense." For example, it's easy to see why R is able to read something like this:

```
R> as.numeric("32.4")
[1] 32.4
```

However, the following conversion makes no sense:

```
R> as.numeric("g'day mate")
[1] NA
Warning message:
NAs introduced by coercion
```

As a result, the entry is returned as `NA` (in this case, R has also issued a corresponding "warning" message). This means that in certain cases, multiple coercions are needed to attain the final result. Suppose, for example, you have the character vector `c("1","0","1","0","0")` and you want to coerce it to a logical-valued vector. Direct character to logical coercion is not possible (this rule is in place because even if all the character strings contained numbers, there is no guarantee in general that they would all be ones and zeros).

```
R> as.logical(c("1","0","1","0","0"))
[1] NA NA NA NA NA
```

However, as you just saw, character string numbers can be converted to a numeric data type, and you know from earlier that ones and zeros are easily coerced to logicals. So, you can perform the coercion in two steps, as follows.

```
R> as.logical(as.numeric(c("1","0","1","0","0")))
[1]  TRUE FALSE  TRUE FALSE FALSE
```

Not all data-type coercion is entirely straightforward. Factors, for example, are trickier because R treats the levels as integers. In other words, regardless of how the levels of a given factor are actually labeled, the software will refer to them internally as level 1, level 2, and so on. This is clear if you try to coerce a factor to a numeric data type.

```
R> baz <- factor(x=c("male","male","female","male"))
R> baz
[1] male   male   female male
Levels: female male
R> as.numeric(baz)
[1] 2 2 1 2
```

Here, you see that R has assigned the numeric representation of the factor in alphabetical order of the factor labels. Level 1 refers to `female`, and level 2 refers to `male`. This example is simple enough, though it's important to be aware of the behavior since coercion from factors with numeric levels can cause confusion.

```
R> qux <- factor(x=c(2,2,3,5))
R> qux
[1] 2 2 3 5
Levels: 2 3 5
R> as.numeric(qux)
[1] 1 1 2 3
```

The numeric representation of the factor `qux` is `c(1,1,2,3)`. This highlights again that the levels of `qux` are simply treated as level 1 (even though it has a label of `2`), level 2 (which has a label of `3`), and level 3 (which has a label of `5`).

Coercion between object classes and structures can also be useful. Sometimes, for example, you simply need to remove the dimensional information of a matrix, storing its contents as a single vector.

```
R> foo <- matrix(data=1:4,nrow=2,ncol=2)
R> foo
     [,1] [,2]
[1,]    1    3
[2,]    2    4
R> as.vector(foo)
[1] 1 2 3 4
```

Note that as.vector has coerced the matrix by "stacking" the columns into a single vector. The same column-wise decomposition occurs for higher-dimensional arrays, in order of layer/block.

```
R> bar <- array(data=c(8,1,9,5,5,1,3,4,3,9,8,8),dim=c(2,3,2))
R> bar
, , 1

     [,1] [,2] [,3]
[1,]    8    9    5
[2,]    1    5    1

, , 2

     [,1] [,2] [,3]
[1,]    3    3    8
[2,]    4    9    8

R> as.matrix(bar)
      [,1]
 [1,]    8
 [2,]    1
 [3,]    9
 [4,]    5
 [5,]    5
 [6,]    1
 [7,]    3
 [8,]    4
 [9,]    3
[10,]    9
[11,]    8
[12,]    8

R> as.vector(bar)
 [1] 8 1 9 5 5 1 3 4 3 9 8 8
```

Similar commonsense rules for data types apply to coercion when working with object structures. For example, coercing the following list baz to a data frame produces an error:

```
R> baz <- list(var1=foo,var2=c(T,F,T),var3=factor(x=c(2,3,4,4,2)))
R> baz
$var1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
$var2
[1]  TRUE FALSE  TRUE

$var3
[1] 2 3 4 4 2
Levels: 2 3 4

R> as.data.frame(baz)
Error in data.frame(var1 = 1:4, var2 = c(TRUE, FALSE, TRUE), var3 = c(1L,  :
  arguments imply differing number of rows: 2, 3, 5
```

The error occurs because the variables do not have matching lengths. But there is no problem with coercing the list `qux`, shown next, which has equal-length members:

```
R> qux <- list(var1=c(3,4,5,1),var2=c(T,F,T,T),var3=factor(x=c(4,4,2,1)))
R> qux
$var1
[1] 3 4 5 1

$var2
[1]  TRUE FALSE  TRUE  TRUE

$var3
[1] 4 4 2 1
Levels: 1 2 4

R> as.data.frame(qux)
  var1  var2 var3
1    3  TRUE    4
2    4 FALSE    4
3    5  TRUE    2
4    1  TRUE    1
```

The discussion in this chapter on object classes, data types, and coercion is not exhaustive, but it serves as a useful introduction to convey how R deals with issues surrounding the formal identification, description, and handling of the objects you create—issues that are present for most high-level languages. Once you're more familiar with R, the help files (like the one accessed by typing `?as` at the prompt) provide further details about object handling in the software.

Exercise 6.3

a. Identify the class of the following objects. For each object, also state whether the class is explicitly or implicitly defined.

   i. `foo <- array(data=1:36,dim=c(3,3,4))`
   ii. `bar <- as.vector(foo)`
   iii. `baz <- as.character(bar)`
   iv. `qux <- as.factor(baz)`
   v. `quux <- bar+c(-0.1,0.1)`

b. For each of the objects as defined in (a), find the sum of the result of calling `is.numeric` and `is.integer` on each of them separately. For example, the first observation would be computed with `is.numeric(foo)+is.integer(foo)`. Turn the collection of five results into an appropriately ordered factor with levels identified by the results themselves, in other words, `0`, `1`, and `2`. Compare this factor vector with the result of coercing it to a numeric vector.

c. Turn the following:

```
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    3    6    9   12
[3,]    4    7   10   13
```

   into the following:

```
[1] "2"  "5"  "8"  "11" "3"  "6"  "9"  "12" "4"  "7"  "10" "13"
```

d. Store the following matrix:

$$\begin{bmatrix} 34 & 0 & 1 \\ 23 & 1 & 2 \\ 33 & 1 & 1 \\ 42 & 0 & 1 \\ 41 & 0 & 2 \end{bmatrix}$$

   Then, do the following:

   i. Coerce the matrix to a data frame.
   ii. As a data frame, coerce the second column to be logical-valued.
   iii. As a data frame, coerce the third column to be factor-valued.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`Inf, -Inf`	Value for ±infinity	Section 6.1.1, p. 104
`is.infinite`	Element-wise check for `Inf`	Section 6.1.1, p. 105
`is.finite`	Element-wise check for finiteness	Section 6.1.1, p. 105
`NaN`	Value for invalid numerics	Section 6.1.2, p. 106
`is.nan`	Element-wise check for `NaN`	Section 6.1.2, p. 107
`NA`	Value for missing observation	Section 6.1.3, p. 108
`is.na`	Element-wise check for `NA` OR `NaN`	Section 6.1.3, p. 109
`na.omit`	Delete all `NA`s and `NaN`s	Section 6.1.3, p. 110
`NULL`	Value for "empty"	Section 6.1.4, p. 110
`is.null`	Check for `NULL`	Section 6.1.4, p. 111
`attributes`	List explicit attributes	Section 6.2.1, p. 115
`attr`	Obtain specific attribute	Section 6.2.1, p. 115
`dimnames`	Get array dimension names	Section 6.2.1, p. 116
`class`	Get object class (S3)	Section 6.2.2, p. 118
`is._`	Object-checking functions	Section 6.2.3, p. 120
`as._`	Object-coercion functions	Section 6.2.4, p. 122

7

BASIC PLOTTING

One particularly popular feature of R is its incredibly flexible plotting tools for data and model visualization. This aspect of the software is what draws many to R in the first place. Mastering R's graphical functionality does require practice, but the fundamental concepts are straightforward. In this chapter, I'll provide an overview of the `plot` function and some useful options for controlling the appearance of the final graph. Then I'll cover the basics of using `ggplot2`, a powerful library for visualizing data in R. I'll return to these plotting tools in Chapter 14, where I'll cover specialized plotting commands such as those for histograms or box-and-whisker plots, and in Part V, which covers advanced plotting techniques.

7.1 Using plot with Coordinate Vectors

The easiest way to think about generating plots in R is to treat your screen as a blank, two-dimensional canvas. You can plot points and lines using *x*- and *y*-coordinates. On paper, these coordinates are usually represented in the typical *Cartesian* sense, with points written as a pair: (*x* value, *y* value). The R function `plot`, on the other hand, takes in two vectors—one for the *x* locations and the other for the *y* locations—and opens a *graphics device*

where it displays the result. If a graphics device is already open, R's default behavior is to refresh the device, overwriting the current contents with the new plot.

For example, let's say you wanted to plot the points (1.1,2), (2,2.2), (3.5,−1.3), (3.9,0), and (4.2,0.2). In `plot`, the vector of *x* locations must be provided first, with the *y* locations second. Let's define these as `foo` and `bar`, respectively:

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
R> plot(foo,bar)
```

Figure 7-1 shows the resulting graphics device with the plot (I will use this simple data set as a working example throughout this section).

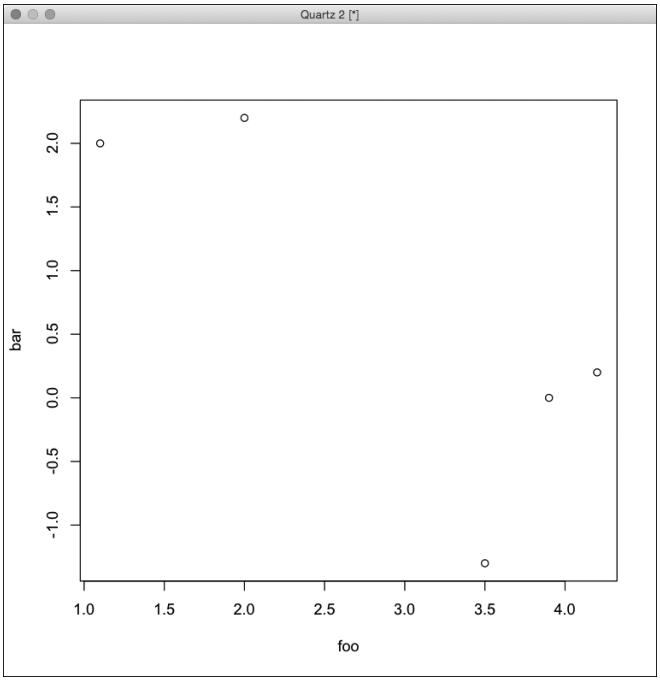


Figure 7-1: The five plotted points using R's default behavior

The *x* and *y* locations don't necessarily need to be specified as separate vectors. You can also supply coordinates in the form of a matrix (with the *x* values in the first column and the *y* values in the second column) or as a list. For example, setting up a matrix of the five points, the following code exactly reproduces Figure 7-1 (note the window pane will look slightly different depending on your operating system):

```
R> baz <- cbind(foo,bar)
R> baz
```

```
     foo  bar
[1,] 1.1  2.0
[2,] 2.0  2.2
[3,] 3.5 -1.3
[4,] 3.9  0.0
[5,] 4.2  0.2
R> plot(baz)
```

The `plot` function is one of R's versatile *generic* functions. It works differently for different objects and allows users to define their own methods for handling objects (including user-defined object classes). Technically, the version of the `plot` command that you've just used is internally identified as `plot.default`. The help file `?plot.default` provides additional details on this *scatterplot* style of data visualization.

7.2 Graphical Parameters

There are a wide range of *graphical parameters* that can be supplied as arguments to the `plot` function (or other plotting functions, such as those in Section 7.3). These parameters range from simple visual enhancements (such as the color of points and axis labels) to the very nature of the graphics device itself (such as how many separate plots to include at once). Some of the most commonly used graphical parameters are listed here; I'll briefly discuss each of these in turn in the following sections:

- `type`: Tells R how to plot the supplied coordinates (for example, as stand-alone points or joined by lines or both dots and lines).
- `main`, `xlab`, `ylab`: Options to include plot title, the horizontal axis label, and the vertical axis label, respectively.
- `col`: Color (or colors) to use for plotting points and lines.
- `pch`: Stands for *point character*. This selects which character to use for plotting individual points.
- `cex`: Stands for *character expansion*. This controls the size of plotted point characters.
- `lty`: Stands for *line type*. This specifies the type of line (such as solid, dotted, or dashed).
- `lwd`: Stands for *line width*. This controls the thickness of plotted lines.
- `xlim`, `ylim`: Provides limits for the horizontal range and vertical range (respectively) of the plotting region.

7.2.1 Automatic Plot Types

By default, the `plot` function will plot individual points, as shown in Figure 7-1. But in many cases, it makes more sense to show lines connecting each coordinate (when plotting time series data, for example). To control

this, you can specify a single character-valued option for the argument `type`. Using `foo` and `bar` from Section 7.1, the following produces the plot in the left panel of Figure 7-2:

```
R> plot(foo,bar,type="l")
```

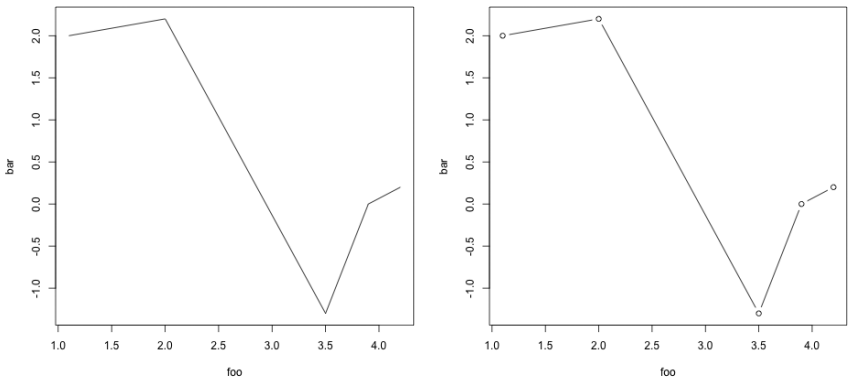


Figure 7-2: A line plot produced using five adjoined coordinates, setting `type="l"` *(left) or* `type="b"` *(right)*

The default value for `type` is `"p"`, which can be interpreted as "points only." Since you didn't specify anything different, this is what was used for the graph in Figure 7-1. In this last example, on the other hand, you've set `type="l"` (meaning "lines only"). Other options include `"b"` for both points *and* lines (shown in the right panel of Figure 7-2) and `"o"` for overplotting the points with lines (this eliminates the gaps between points and lines visible for `type="b"`). The option `type="n"` results in no points or lines plotted, creating an empty plot (which can be useful for complicated plots that must be constructed in steps).

7.2.2 Title and Axis Labels

A main title and axis names are often required to make the plotted data easier to interpret. You can add these by supplying text as character strings to `main` for a title, `xlab` for the *x*-axis, and `ylab` for the *y*-axis. Note that these strings may include escape sequences (discussed in Section 4.2.3). The following code produces the plots in Figure 7-3:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="x axis label",
        ylab="location y")
R> plot(foo,bar,type="b",main="My lovely plot\ntitle on two lines",xlab="",
        ylab="")
```

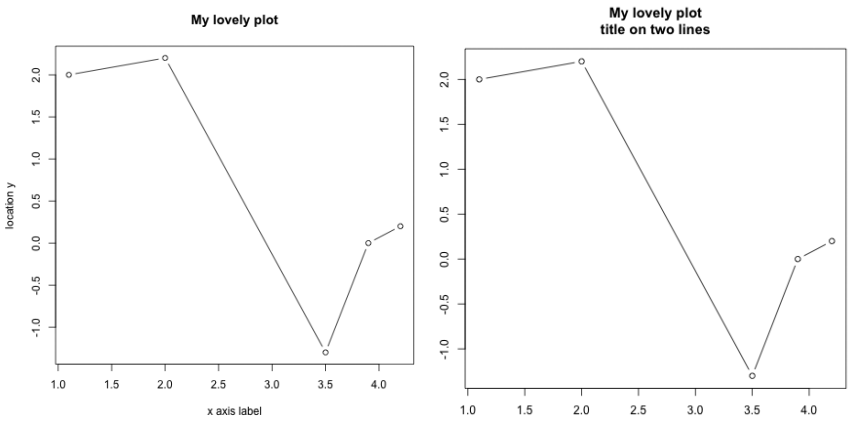


Figure 7-3: Two examples of plots with axis labels and titles

In the second plot, note how the new line escape sequence splits the title into two lines. In that plot, `xlab` and `ylab` are also set to the empty string `""` to prevent R's default of labeling the axes with the names of the *x* and *y* vectors.

7.2.3 Color

Adding color to a graph is far from just an aesthetic consideration. Color can make data much clearer—for example by distinguishing factor levels or emphasizing important numerical limits. You can set colors with the `col` parameter in a number of ways. The simplest options are to use an integer selector or a character string (see the possible color string values recognized by R by typing `colors()` at the prompt). The default color is integer `1` or the character string `"black"`. The top row of Figure 7-4 shows two examples of colored graphs, created by the following code:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",col=2)
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",col="seagreen4")
```

There are eight possible integer values (shown in the leftmost plot of Figure 7-5) and around 650 character strings to specify color. But you aren't limited to these options since you can also specify colors using RGB (red, green, and blue) levels and by creating your own palettes. I'll talk more about the last two options in Chapter 25.

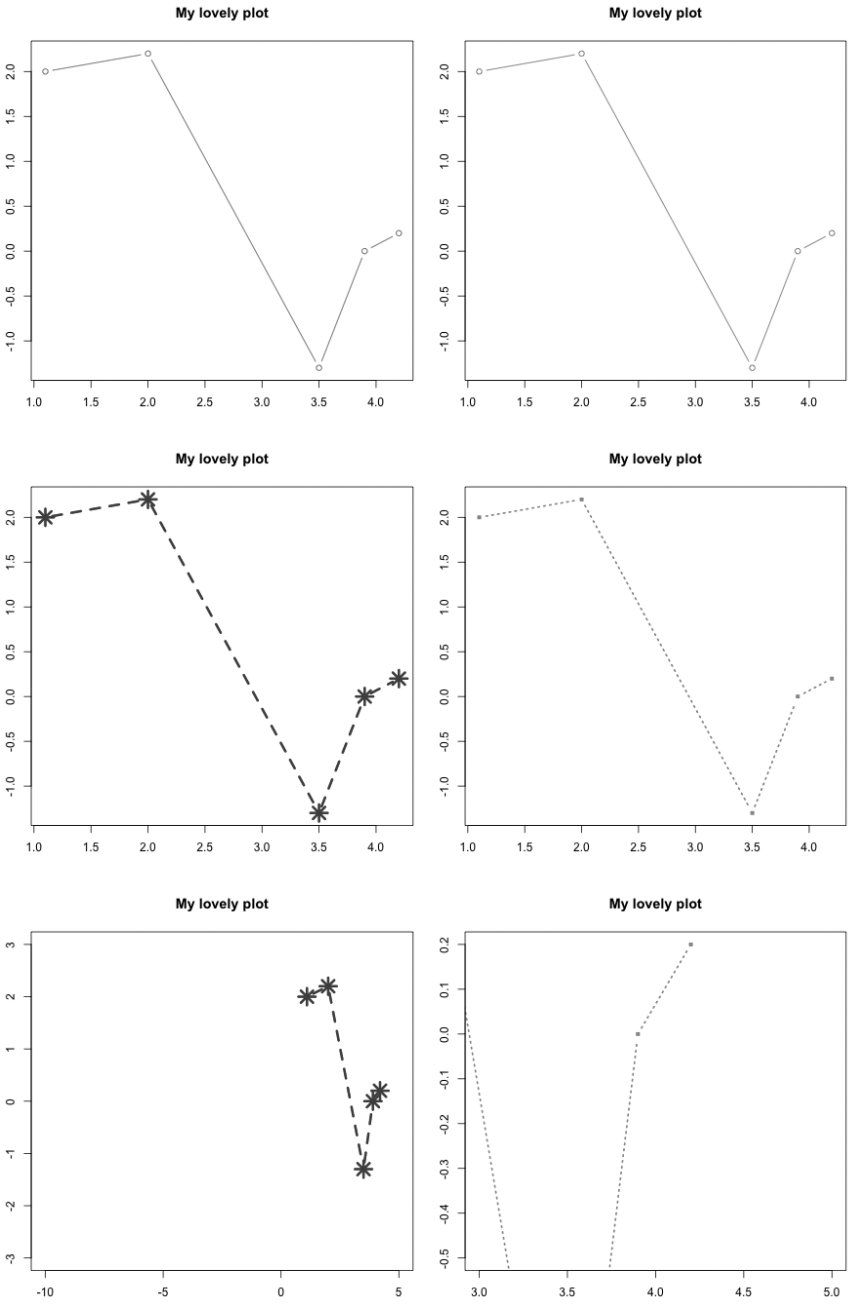


Figure 7-4: Experimenting with basic R plotting. Top row: Two examples of colored plots with `col=2` *(left) and* `col="seagreen4"` *(right). Middle row: Two further examples making use of* `pch`*,* `lty`*,* `cex`*, and* `lwd`*. Bottom row: Setting plotting region limits* `xlim=c(-10,5)`*,* `ylim=c(-3,3)` *(left), and* `xlim=c(3,5)`*,* `ylim=c(-0.5,0.2)` *(right).*

7.2.4 Line and Point Appearances

You can use the `pch` and `lty` parameters to alter the appearance of the points and lines (respectively) of the plotted data. The `pch` parameter controls the character used to plot individual data points. You can specify a single character to use for each point, or you can specify a value between `1` and `25` (inclusive). The symbols corresponding to each integer are shown in the middle plot of Figure 7-5. The `lty` parameter, which affects the type of line drawn, can take the values `1` through `6`. These options are shown in the rightmost plot of Figure 7-5.

You can also control the size of plotted points and the thickness of lines using the parameters `cex` and `lwd`, respectively. The default size and thickness for both of these is `1`. To request half-size points, for example, you'd specify `cex=0.5`; to specify double-thick lines, use `lwd=2`.

The following two lines produce the two plots in the middle row of Figure 7-4, showing off `pch`, `lty`, `cex`, and `lwd`:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=4,pch=8,lty=2,cex=2.3,lwd=3.3)
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=6,pch=15,lty=3,cex=0.7,lwd=2)
```

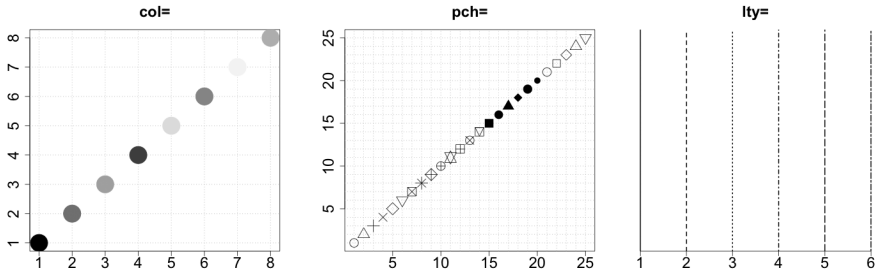


Figure 7-5: Some reference plots giving the results of possible integer options of `col` *(left),* `pch` *(middle), and* `lty` *(right)*

7.2.5 Plotting Region Limits

As you can see in the plots of `foo` and `bar`, by default R sets the range of each axis by using the range of the supplied *x* and *y* values (plus some small constant to pad a little area around the outermost points). But you might need more space than this, such as to annotate individual locations, include a legend, or plot additional points that fall outside the original ranges (as you'll see in Section 7.3). You can set custom plotting area limits using `xlim` and `ylim`. Both parameters require a numeric vector of length 2, provided as `c(`*`lower limit,upper limit`*`)`.

Consider the plots in the bottom row of Figure 7-4, created by the following two commands:

```
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=4,pch=8,lty=2,cex=2.3,lwd=3.3,xlim=c(-10,5),ylim=c(-3,3))
R> plot(foo,bar,type="b",main="My lovely plot",xlab="",ylab="",
        col=6,pch=15,lty=3,cex=0.7,lwd=2,xlim=c(3,5),ylim=c(-0.5,0.2))
```

These plots are exactly the same as the two in the middle row, except for one important difference. In the bottom-left plot of Figure 7-4, the *x*- and *y*-axes are set to be much wider than the observed data, and the plot on the right restricts the plotting window so that only a portion of the data is displayed.

7.3 Adding Points, Lines, and Text to an Existing Plot

Generally speaking, each call to `plot` will refresh the active graphics device for a new plotting region. But this is not always desired—starting with an empty plotting region and progressively adding points, lines, text, and legends to this canvas is one of the easiest ways to build more complicated plots. Here are some useful, ready-to-use functions in R that will add to a plot without refreshing or clearing the window:

- `points`: Adds points
- `lines`, `abline`, `segments`: Adds lines
- `text`: Writes text
- `arrows`: Adds arrows
- `legend`: Adds a legend

The syntax for calling and setting parameters for these functions is the same as `plot`. The best way to see how these work is through an extended example, which I'll base on some hypothetical data made up of 20 (x, y) locations.

```
R> x <- 1:20
R> y <- c(-1.49,3.37,2.59,-2.78,-3.94,-0.92,6.43,8.51,3.41,-8.23,
       -12.01,-6.58,2.87,14.12,9.63,-4.58,-14.78,-11.67,1.17,15.62)
```

Using these data, you'll build up the plot shown in Figure 7-6 (note that you may need to manually enlarge your graphics device and replot to ensure the legend doesn't overlap other features of the image). In passing, I'll also remark on a generally accepted rule in plotting: "keep it clear and simple." Admittedly, Figure 7-6 may not subscribe to this in its entirety, but I'll continue purely for the sake of demonstrating the R commands used.

In the plot of interest, the data points are plotted differently according to their *x* and *y* locations, depending on their relation to the "sweet spot" pointed out in the figure. Points with a *y* value greater than 5 are

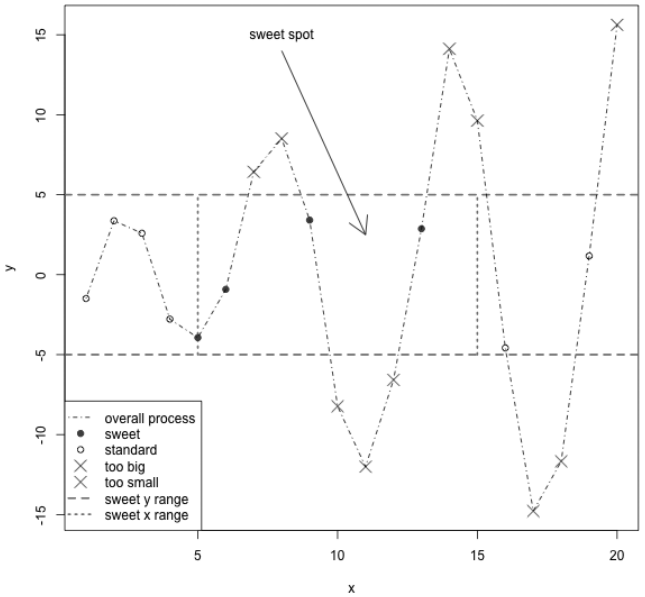


Figure 7-6: An elaborate final plot of some hypothetical data

marked with a purple ×; points with a *y* value less than −5 are marked with a green ×. Points between these two *y* values but still outside of the sweet spot are marked with a ∘. Finally, points in the sweet spot (with *x* between 5 and 15 *and* with *y* between −5 and 5) are marked as a blue •. Red horizontal and vertical lines delineate the sweet spot, which is labeled with an arrow, and there's also a legend.

Ten lines of code were used to build this plot in its entirety (plus one additional line to add the legend). The plot, as it looks at each step, is given in Figure 7-7. The corresponding lines of code are detailed next.

Line **(1)**

```
R> plot(x,y,type="n",main="")
```

The first step is to create the empty plotting region where you can add points and draw lines. This first line tells R to plot the data in `x` and `y`, though the option `type` is set to `"n"`. As mentioned in Section 7.2, this opens or refreshes the graphics device and sets the axes to the appropriate lengths (with labels and axes), but it doesn't plot any points or lines.

Line **(2)**

```
R> abline(h=c(-5,5),col="red",lty=2,lwd=2)
```

The `abline` function is a simple way to add straight lines spanning a plot. The line (or lines) can be specified with *slope* and *intercept* values (see the later discussions on regression in Chapter 20). You can also simply add horizontal or vertical lines. This line of code adds two

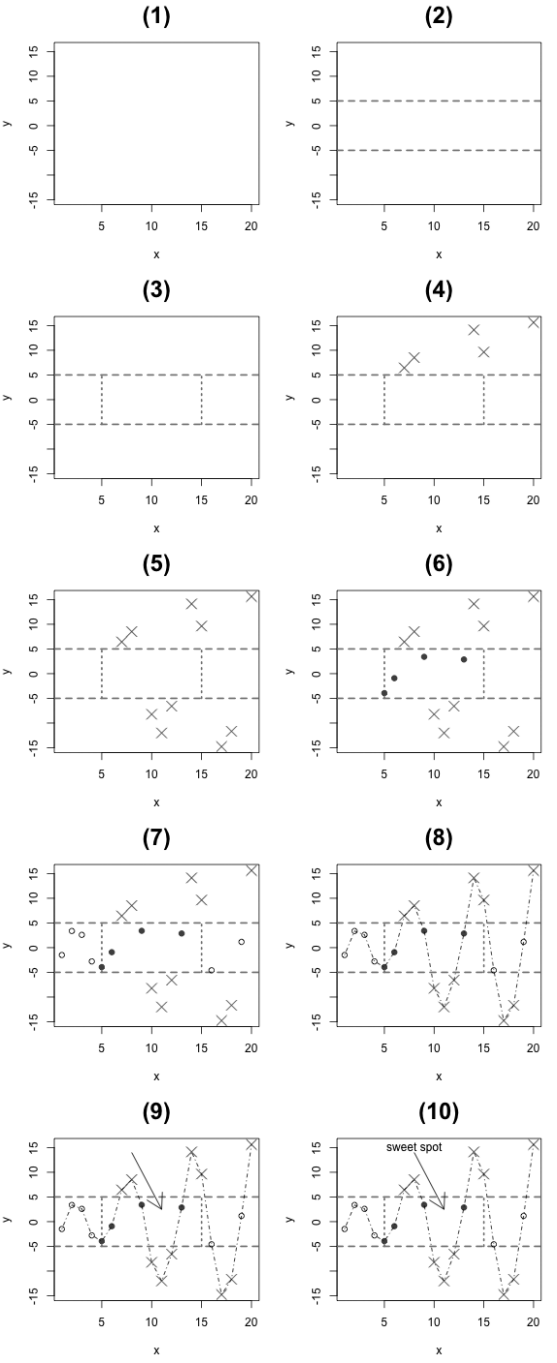


Figure 7-7: Building the final plot given in Figure 7-6. The plots (1) through (10) correspond to the itemized lines of code in the text.

separate horizontal lines, one at $y = 5$ and the other at $y = 5$, using `h=c(-5,5)`. The three parameters (covered in Section 7.2) make these two lines red, dashed, and double-thickness. For vertical lines, you could have written `v=c(-5,5)`, which would have drawn them at $x = -5$ and $x = 5$.

Line **(3)**

```
R> segments(x0=c(5,15),y0=c(-5,-5),x1=c(5,15),y1=c(5,5),col="red",lty=3,
           lwd=2)
```

The third line of code adds shorter vertical lines between the horizontal ones drawn in the previous step. For this you use `segments`, not `abline`, since you don't want these lines to span the entire plotting region. The `segments` command takes a "from" coordinate (given as `x0` and `y0`) and a "to" coordinate (as `x1` and `y1`) and draws the corresponding line. The vector-oriented behavior of R matches up the two sets of "from" and "to" coordinates. Both lines are red and dotted and have double-thickness. (You could also supply vectors of length 2 to these parameters, in which case the first segment would use the first parameter value and the second segment would use the second value.)

Line **(4)**

```
R> points(x[y>=5],y[y>=5],pch=4,col="darkmagenta",cex=2)
```

As step 4, you use `points` to begin adding specific coordinates from `x` and `y` to the plot. Just like `plot`, `points` takes two vectors of equal lengths with x and y values. In this case, you want points plotted differently according to their location, so you use logical vector subsetting (see Section 4.1.5) to identify and extract elements of `x` *and* `y` where the y value is greater than or equal to 5. These (and only these) points are added as purple × symbols and are enlarged by a factor of 2 with `cex`.

Line **(5)**

```
R> points(x[y<=-5],y[y<=-5],pch=4,col="darkgreen",cex=2)
```

The fifth line of code is much like the previous line; this time it extracts the coordinates where y values are less than or equal to -5. The same point character is used, but this time you set the color to dark green.

Line **(6)**

```
R> points(x[(x>=5&x<=15)&(y>-5&y<5)],y[(x>=5&x<=15)&(y>-5&y<5)],pch=19,
          col="blue")
```

The sixth step adds the blue "sweet spot" points, which are identified with `(x>=5&x<=15)&(y>-5&y<5)`. This slightly more complicated set of conditions extracts the points whose x location lies between 5 and 15 (inclusive) *and* whose y location lies between -5 and 5 (exclusive). Note that this line uses the "short" form of the logical operator `&` throughout since you want element-wise comparisons here (see Section 4.1.3).

Line **(7)**

```
R> points(x[(x<5|x>15)&(y>-5&y<5)],y[(x<5|x>15)&(y>-5&y<5)])
```

This command identifies the remaining points in the data set (with an x value that is either less than 5 *or* greater than 15 and a y value between −5 and 5). No graphical parameters are specified, so these points are plotted with the default black ∘.

Line **(8)**

```
R> lines(x,y,lty=4)
```

To draw lines connecting the coordinates in `x` and `y`, you use `lines`. Here you've also set `lty` to `4`, which draws a dash-dot-dash style line.

Line **(9)**

```
R> arrows(x0=8,y0=14,x1=11,y1=2.5)
```

The ninth line of code adds the arrow pointing to the sweet spot. The function `arrows` is used just like `segments`, where you provide a "from" coordinate (`x0`, `y0`) and a "to" coordinate (`x1`, `y1`). By default, the head of the arrow is located at the "to" coordinate, though this (and other options such as the angle and length of the head) can be altered using optional arguments described in the help file `?arrows`.

Line **(10)**

```
R> text(x=8,y=15,labels="sweet spot")
```

The tenth line prints a label on the plot at the top of the arrow. As per the default behavior of `text`, the string supplied as `labels` is *centered* on the coordinates provided with the arguments `x` and `y`.

As a finishing touch, you can add the legend with the `legend` function, which gives you the final product shown in Figure 7-6.

```
legend("bottomleft",
       legend=c("overall process","sweet","standard",
                "too big","too small","sweet y range","sweet x range"),
       pch=c(NA,19,1,4,4,NA,NA),lty=c(4,NA,NA,NA,NA,2,3),
       col=c("black","blue","black","darkmagenta","darkgreen","red","red"),
       lwd=c(1,NA,NA,NA,NA,2,2),pt.cex=c(NA,1,1,2,2,NA,NA))
```

The first argument sets where the legend should be placed. There are various ways to do this (including setting exact x- and y-coordinates), but it often suffices to pick a corner using one of the four following character strings: `"topleft"`, `"topright"`, `"bottomleft"`, or `"bottomright"`. Next you supply the labels/text as a vector of character strings to the `legend` argument. Then you need to supply the remaining argument values in vectors of the same length so that the right elements match up with each label.

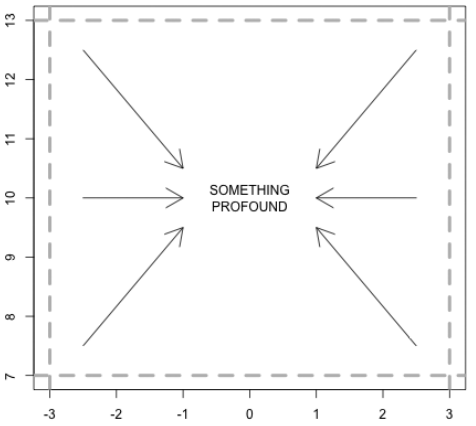
For example, for the first label (`"overall process"`), you want a line of type 4 with default thickness and color. So, in the first positions of the

remaining argument vectors, you set `pch=NA`, `lty=4`, `col="black"`, `lwd=1`, and `pt.cex=NA` (all of these are default values, except for `lty`). Here, `pt.cex` simply refers to the `cex` parameter when calling `points` (using just `cex` in `legend` would expand the text used, not the points).

Note that you have to fill in some elements in these vectors with `NA` when you don't want to set the corresponding graphical parameter. This is just to preserve the equal lengths of the vectors supplied so R can track which parameter values correspond to each particular reference.

Exercise 7.1

a. As closely as you can, re-create the following plot:



b. With the following data, create a plot of weight on the *x*-axis and height on the *y*-axis. Use different point characters or colors to distinguish between males and females and provide a matching legend. Label the axes and give the plot a title.

Weight (kg)	Height (cm)	Sex
55	161	female
85	185	male
75	174	male
42	154	female
93	188	male
63	178	male
58	170	female
75	167	male
89	181	male
67	178	female

7.4 The ggplot2 Package

This chapter so far has shown off R's built-in graphical tools (often referred to as *base R graphics* or *traditional R graphics*). Now, let's look at another important suite of graphical tools: `ggplot2`, a prominent contributed package by Hadley Wickham (Wickham 2009). Available on CRAN like any other contributed package, `ggplot2` offers particularly powerful alternatives to the standard plotting procedures in R. The *gg* stands for *grammar of graphics*—the functionality in `ggplot2` follows a particular approach to graphical production described by Wilkinson (2005). In doing so, `ggplot2` standardizes the production of different plot/graph types, streamlines some of the more fiddly aspects of adding to existing plots (such as including a legend), and lets you build plots by defining and manipulating *layers*. For the moment, let's see the elementary behavior of `ggplot2` using the same simple examples in Sections 7.1–7.3. You'll get familiar with the basic plotting function `qplot` and how it differs from the generic `plot` function used earlier. I'll return to the topic of `ggplot2` when I cover statistical plots in Chapter 14, and you'll investigate even more advanced abilities in Chapter 24.

7.4.1 A Quick plot with qplot

First, you must install the `ggplot2` package by downloading it manually or simply typing `install.packages("ggplot2")` at the prompt (see Section A.2.3). Then, load the package with the following:

```
R> library("ggplot2")
```

Now, let's go back to the five data points originally stored in Section 7.1 as `foo` and `bar`.

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
```

You can produce `ggplot2`'s version of Figure 7-1 using its "quick plot" function `qplot`.

```
R> qplot(foo,bar)
```

The result is shown in the left panel of Figure 7-8.

There are some obvious differences between this image and the one produced using `plot`, but the basic syntax of `qplot` is the same as earlier. The first two arguments passed to `qplot` are vectors of equal length, with the *x*-coordinates in `foo` supplied first, followed by the *y*-coordinates in `bar`.

Adding a title and axis labels also uses the same arguments you already saw with `plot` in Section 7.2.

```
R> qplot(foo,bar,main="My lovely qplot",xlab="x axis label",ylab="location y")
```

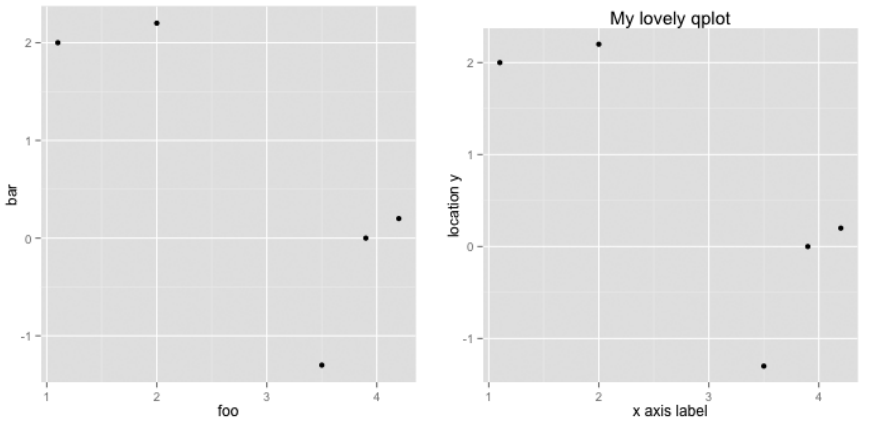


Figure 7-8: Five plotted points using `ggplot2`*'s default behavior for the* `qplot` *function (left) and with title and axis labels added (right)*

This produces the right panel of Figure 7-8.

Underneath this basic similarity in syntax, though, there is a fundamental difference between how `ggplot2` and base R graphics create plots. Constructing plots using the built-in graphics tools is essentially a live, step-by-step process. This was particularly noticeable in Section 7.3, where you treated the graphics device as an active canvas where you added points, lines, and other features one by one. By contrast, `ggplot2` plots are stored as objects, which means they have an underlying, static representation until you *change* the object—what you essentially visualize with `qplot` is the `print`ed object at any given time. To highlight this, consider the following code:

```
R> baz <- plot(foo,bar)
R> baz
NULL
R> qux <- qplot(foo,bar)
R> qux
```

The first assignment uses the built-in `plot` function. When you run that line of code, the plot in Figure 7-1 pops up. Since nothing is actually stored in the workspace, printing the supposed object `baz` yields the empty `NULL` value. On the other hand, it makes sense to store the `qplot` content (stored as the object `qux` here). This time, when you perform the assignment, no plot is displayed. The graphic, which matches Figure 7-8, is displayed only upon typing `qux` at the prompt, which invokes the `print` method for that object. This may seem like a minor point, but the fact that you can save a plot this way before displaying it opens up new ways to modify or enhance plots before displaying them (as you will see in a moment), and it can be a distinct advantage over base R graphics.

7.4.2 Setting Appearance Constants with Geoms

To add and customize points and lines in a `ggplot2` graphic, you alter the object itself, rather than using a long list of arguments or secondary functions executed separately (such as `points` or `lines`). You can modify the object using `ggplot2`'s convenient suite of *geometric modifiers* (*geoms*). Let's say you want to connect the five points in `foo` and `bar` with a line, just as you did in Section 7.1. You can first create a blank plot object and then use geometric modifiers on it like this:

```
R> qplot(foo,bar,geom="blank") + geom_point() + geom_line()
```

The resulting plot is shown on the left of Figure 7-9. In the first call to `qplot`, you create an empty plot object by setting the initial geometric modifier as `geom="blank"` (if you displayed this plot, you would just see the gray background and the axes). Then you layer on the two other geoms as `geom_point()` and `geom_line()`. As indicated by the parentheses, these geoms are functions that result in their own specialized objects. You can add geoms to the `qplot` object using the + operator. Here, you have not supplied any arguments to either geom, which means they will operate on the same data originally supplied to `qplot` (`foo` and `bar`) and they will stick with default settings for any other features such as color or point/line type. You can control those features as well, as shown here:

```
R> qplot(foo,bar,geom="blank") + geom_point(size=3,shape=6,color="blue")
                               + geom_line(color="red",linetype=2)
```

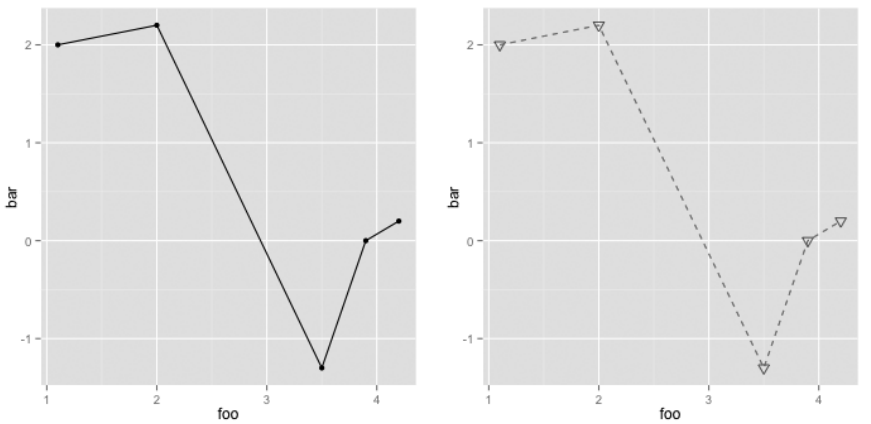


Figure 7-9: Two simple plots that use geometric modifiers to alter the appearance of a `qplot` *object. Left: adding points and lines using default settings. Right: using the geoms to affect point character, size, and color, and line type and color.*

Note that some of ggplot2's argument names used here for things such as point characters and size (shape and size) are different from the base R graphics arguments (pch and cex). But ggplot2 is actually compatible with many of the common graphical parameters used in R's standard plot function, so you can use those arguments here too if you prefer. For example, setting cex=3 and pch=6 in geom_point in the previous example would result in the same image.

The object-oriented nature of ggplot2 graphics means tweaking a plot or experimenting with different visual features no longer requires you to rerun every plotting command each time you change something. This is facilitated by geoms. Say you like the line type used on the right side of Figure 7-9 but want a different point character. To experiment, you could first store the qplot object you created earlier and then use geom_point with that object to try different point styles.

```
R> myqplot <- qplot(foo,bar,geom="blank") + geom_line(color="red",linetype=2)
R> myqplot + geom_point(size=3,shape=3,color="blue")
R> myqplot + geom_point(size=3,shape=7,color="blue")
```

The second and third lines, which alter the shape argument of geom_point, produce the graphics on the left and right of Figure 7-10, respectively.

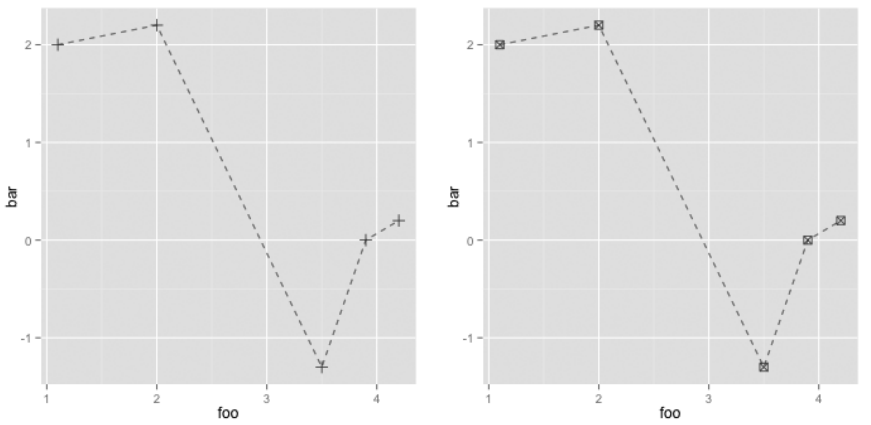


Figure 7-10: Using the object-oriented nature of ggplot2 *graphics to experiment with different point characters*

At the time of writing, there are more than 35 geometric modifiers that can be used this way (that is, with a function name beginning with geom_) in ggplot2. To obtain a list, simply ensure the package is loaded and enter ??"geom_" as a help search at the prompt.

7.4.3 Aesthetic Mapping with Geoms

Geoms and ggplot2 also provide efficient, automated ways to apply different styles to different subsets of a plot. If you split a data set into categories using a factor object, ggplot2 can automatically style each category uniquely. In ggplot2's documentation, the factor that holds these categories is called a *variable*, which ggplot2 can *map* to varying *aesthetic* values. This gets rid of a lot of the effort that goes into isolating subsets of data and plotting them separately using base R graphics (as you did in Section 7.3).

All this is best illustrated with an example. Let's return to the 20 observations you manually plotted, step-by-step, to produce the elaborate plot in Figure 7-6.

```
R> x <- 1:20
R> y <- c(-1.49,3.37,2.59,-2.78,-3.94,-0.92,6.43,8.51,3.41,-8.23,
       -12.01,-6.58,2.87,14.12,9.63,-4.58,-14.78,-11.67,1.17,15.62)
```

In Section 7.3, you defined several hypothetical categories that classified each observation as either "standard," "sweet," "too big," or "too small" based on their x and y values. Using those same classification rules, let's explicitly define a factor to correspond to x and y.

```
R> ptype <- rep(NA,length(x=x))
R> ptype[y>=5] <- "too_big"
R> ptype[y<=-5] <- "too_small"
R> ptype[(x>=5&x<=15)&(y>-5&y<5)] <- "sweet"
R> ptype[(x<5|x>15)&(y>-5&y<5)] <- "standard"
R> ptype <- factor(x=ptype)
R> ptype
 [1] standard  standard  standard  standard  sweet     sweet     too_big
 [8] too_big   sweet     too_small too_small too_small sweet     too_big
[15] too_big   standard  too_small too_small standard  too_big
Levels: standard sweet too_big too_small
```

Now you have a factor with 20 values sorted into four levels. You'll use this factor to tell qplot how to map your aesthetics. Here's a simple way to do that:

```
R> qplot(x,y,color=ptype,shape=ptype)
```

This produces the left plot in Figure 7-11. Note that already this single line of code has produced a plot that separates the four categories by color and point character and has even provided a legend. This was all done by the aesthetic mapping in the call to qplot, where you set color and shape to be mapped to the ptype variable. Now, let's replot these data using the same qplot object along with a suite of geom modifications in order to get something more like Figure 7-6. The following line produces the plot on the right of Figure 7-11.

```
R> qplot(x,y,color=ptype,shape=ptype) + geom_point(size=4)
    + geom_line(mapping=aes(group=1),color="black",lty=2)
    + geom_hline(mapping=aes(yintercept=c(-5,5)),color="red")
    + geom_segment(mapping=aes(x=5,y=-5,xend=5,yend=5),color="red",lty=3)
    + geom_segment(mapping=aes(x=15,y=-5,xend=15,yend=5),color="red",lty=3)
```

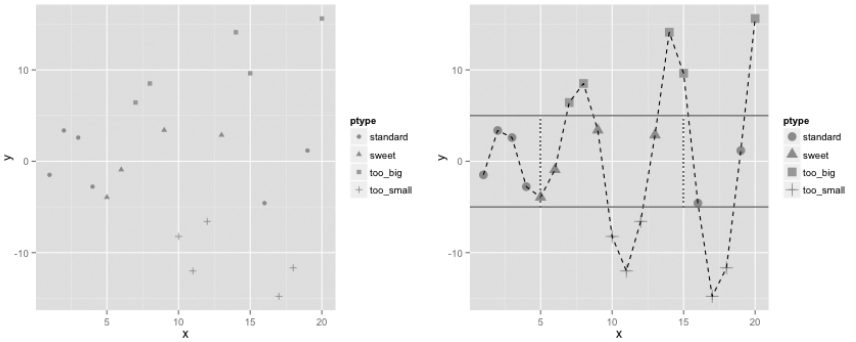


Figure 7-11: Demonstration of aesthetic mapping using `qplot` *and geoms in* `ggplot2`*. Left: the initial call to* `qplot`*, which maps point character and color using* `ptype`*. Right: augmenting the left plot using various geoms to override the default mappings.*

In the first line, you add `geom_point(size=4)` to increase the size of all the points on the graph. In the lines that follow, you add a line connecting all the points, plus horizontal and vertical lines to mark out the sweet spot. Note that for those last four lines, you have to set alternate aesthetic mappings via `aes`. Let's look a little closer at what's going on there.

Since you used `ptype` for aesthetic mapping in the initial call to `qplot`, by default all other geoms will be mapped to each category in the same way, *unless* you override that default mapping. For example, when you call `geom_line` to connect all the points, if you were to stick with the default mapping to `ptype` (that is, if you omitted `mapping=aes(group=1)`), this geom would draw lines connecting points within each category. You would see four separate dashed lines—one connecting all "standard" points, another connecting all "sweet" points, and so on. But that's not what you want here; you want a line that connects all of the points, from left to right. So, you tell `geom_line` to treat all the observations as one group by entering `aes(group=1)`.

After that, you use the `geom_hline` function to draw horizontal lines at $y = -5$ and $y = 5$ using its `yintercept` argument, again passed to `aes` to redefine that geom's `mapping`. In this case, you need to redefine the mapping to operate on the vector `c(-5,5)`, rather than using the observed data in `x` and `y`. Similarly, you end by using `geom_segment` to draw the two vertical dotted line segments. `geom_segment` operates much like `segments`—you redefine the mapping based on a "from" coordinate (arguments `x` and `y`) and a "to" coordinate (`xend` and `yend` here). This wasn't a concern for the first geom, `geom_point(size=4)`, which sets a constant enlarged size for each plotted point.

There it doesn't matter how the geom is mapped since it simply makes a uniform change to each point.

Plotting in R, from base graphics to contributed package tools such as those in the flexible `ggplot2` library, stays true to the nature of the language. As you've seen, element-wise matching is primarily responsible for the ability to create relatively intricate plots with a handful of straightforward and intuitive functions. Once you display a plot, you can save it to the hard drive by selecting the graphics device and choosing File > Save. However, you can also write plots to a file directly, as you'll see momentarily in Section 8.3.

The graphical capabilities explored in this section are merely the tip of the iceberg, and you'll continue to use data visualizations from this point onward.

Exercise 7.2

In Exercise 7.1 (b), you used base R graphics to plot some weight and height data, distinguishing males and females using different points or colors. Repeat this task using `ggplot2`.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`plot`	Create/display base R plot	Section 7.1, p. 130
`type`	Set plot type	Section 7.2.1, p. 132
`main, xlab, ylab`	Set axis labels	Section 7.2.2, p. 132
`col`	Set point/line color	Section 7.2.3, p. 133
`pch, cex`	Set point type/size	Section 7.2.4, p. 135
`lty, lwd`	Set line type/width	Section 7.2.4, p. 135
`xlim, ylim`	Set plot region limits	Section 7.2.5, p. 136
`abline`	Add vertical/horizontal line	Section 7.3, p. 137
`segments`	Add specific line segments	Section 7.3, p. 139
`points`	Add points	Section 7.3, p. 139
`lines`	Add lines following coords	Section 7.3, p. 140
`text`	Add text	Section 7.3, p. 140
`legend`	Add/control legend	Section 7.3, p. 140
`qplot`	Create `ggplot2` "quick plot"	Section 7.4.1, p. 142
`geom_point`	Add points geom	Section 7.4.2, p. 144
`geom_line`	Add lines geom	Section 7.4.2, p. 144
`size, shape, color`	Set geom constants	Section 7.4.2, p. 144
`linetype`	Set geom line type	Section 7.4.2, p. 144
`mapping, aes`	Geom aesthetic mapping	Section 7.4.3, p. 146
`geom_hline`	Add horizontal lines geom	Section 7.4.3, p. 147
`geom_segment`	Add line segments geom	Section 7.4.3, p. 147

8

READING AND WRITING FILES

Now I'll cover one more fundamental aspect of working with R: getting data into and out of an active workspace by reading and writing files. Typically, to work with a large data set, you'll need to read in the data from an external file, whether it's stored as plain text, in a spreadsheet file, or on a website. R provides command line functions you can use to import these data sets, usually as a data frame object. You can also export data frames from R by writing a new file on your computer, plus you can save any plots you create as image files. In this chapter, I'll go over some useful command-based read and write operations for importing and exporting data.

8.1 R-Ready Data Sets

First, let's take a brief look at some of the data sets that are built into the software or are part of user-contributed packages. These data sets are useful samples to practice with and to experiment with functionality.

Enter `data()` at the prompt to bring up a window listing these "R-ready" data sets along with a one-line description. These data sets are organized in alphabetical order by name and grouped by package (the exact list that

appears will depend on what contributed packages have been installed from CRAN; see Section A.2).

8.1.1 Built-in Data Sets

There are a number of data sets contained within the built-in, automatically loaded package `datasets`. You can use the `library` function to bring up a window that summarizes and provides an index of the contents of the package:

```
R> library(help="datasets")
```

R-ready data sets have a corresponding help file where you can find important details about the data and how it's organized. For example, one of the built-in data sets is called `ChickWeight`. If you enter `?ChickWeight` at the prompt, you'll see the window in Figure 8-1.

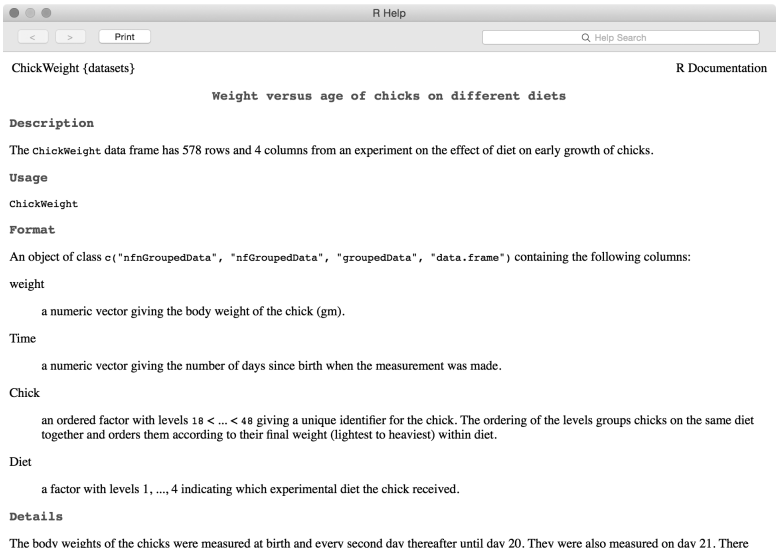


Figure 8-1: The help file for the `ChickWeight` *data set*

As you can see, this file explains the variables and their values, and it notes that the data is stored in a data frame with 578 rows and 4 columns. Since the objects in `datasets` are built in, all you have to do to access `ChickWeight` is type its name at the prompt. Let's look at the first 15 records.

```
R> ChickWeight[1:15,]
   weight Time Chick Diet
1      42    0     1    1
2      51    2     1    1
3      59    4     1    1
4      64    6     1    1
5      76    8     1    1
```

```
6      93   10     1     1
7     106   12     1     1
8     125   14     1     1
9     149   16     1     1
10    171   18     1     1
11    199   20     1     1
12    205   21     1     1
13     40    0     2     1
14     49    2     2     1
15     58    4     2     1
```

You can treat this data set like any other data frame you've created in R.

8.1.2 Contributed Data Sets

There are many more R-ready data sets that come as part of contributed packages. To access them, first install and load the relevant package. Consider the data set `ice.river`, which is in the contributed package `tseries`. First, you have to install the package, which you can do by running the line `install.packages("tseries")` at the prompt. Then, to access the components of the package, load it using `library`:

```
R> library("tseries")

    'tseries' version: 0.10-32

    'tseries' is a package for time series analysis and computational finance.

    See 'library(help="tseries")' for details.
```

Now you can enter `library(help="tseries")` to see the list of data sets in this package, and you can enter `?ice.river` to find more details about the data set you want to work with here. The help file describes `ice.river` as a "time series object" comprised of river flow, precipitation, and temperature measurements—data initially reported in Tong (1990). To finally access this object itself, you must explicitly load it using the `data` function. Then you can work with `ice.river` in your workspace as usual. Here are the first five records:

```
R> data(ice.river)
R> ice.river[1:5,]
      flow.vat flow.jok prec temp
 [1,]    16.10     30.2  8.1  0.9
 [2,]    19.20     29.0  4.4  1.6
 [3,]    14.50     28.4  7.0  0.1
 [4,]    11.00     27.8  0.0  0.6
 [5,]    13.60     27.8  0.0  2.0
```

The availability and convenience of these R-ready data sets make it easy to test code, and I'll use them in subsequent chapters for demonstrations. To analyze your own data, however, you'll often have to import them from some external file. Let's see how to do that.

8.2 Reading in External Data Files

R has a variety of functions for reading characters from stored files and making sense of them. Let's start by looking at how to read *table-format* files, which are among the easiest for R to read and import.

8.2.1 The Table Format

Table-format files are best thought of as plain-text files with three key features that fully define how R should read the data.

- **Header** If a *header* is present, it's always the first line of the file. This optional feature is used to provide names for each column of data. When importing a file into R, you need to tell the software whether a header is present so that it knows whether to treat the first line as variable names or, alternatively, observed data values.
- **Delimiter** The all-important *delimiter* is a unique character not used elsewhere in the file that separates the entries in each line. This tells R when a specific entry begins and ends (in other words, its exact position in the table).
- **Missing value** This is another unique character string used exclusively to denote a missing value. When reading the file, R will turn these entries into the form it recognizes: `NA`.

Typically, these files have a *.txt* extension (highlighting the plain-text style) or *.csv* (for *comma-separated values*).

Let's try an example, using a variation on the data frame `mydata` as defined at the end of Section 5.2.2. Figure 8-2 shows an appropriate table-format file called *mydatafile.txt*, which has the data from that data frame with a few values now marked as missing.

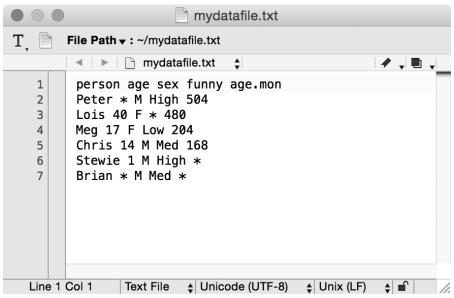


Figure 8-2: A plain-text table-format file

Note that the first line is the header, the values are delimited with a single space, and missing values are denoted with an asterisk (*). Also, note that each new record is required to start on a new line. Suppose you're handed this plain-text file for data analysis in R. The ready-to-use command `read.table` imports table-format files, producing a data frame object, as follows:

```
R> mydatafile <- read.table(file="/Users/tdavies/mydatafile.txt",
                            header=TRUE,sep=" ",na.strings="*",
                            stringsAsFactors=FALSE)
R> mydatafile
  person age sex funny age.mon
1  Peter  NA   M  High     504
2   Lois  40   F  <NA>     480
3    Meg  17   F   Low     204
4  Chris  14   M   Med     168
5 Stewie   1   M  High      NA
6  Brian  NA   M   Med      NA
```

In a call to `read.table`, `file` takes a character string with the filename and folder location (using forward slashes), `header` is a logical value telling R whether `file` has a header (`TRUE` in this case), `sep` takes a character string providing the delimiter (a single space, `" "`, in this case), and `na.strings` requests the characters used to denote missing values (`"*"` in this case).

If you're reading in multiple files and don't want to type the entire folder location each time, it's possible to first set your working directory via `setwd` (Section 1.2.3) and then simply use the filename (and its extension) as the character string supplied to the `file` argument. However, this approach and the previous code both require you to know exactly where your file is located when you're typing at the R prompt. Fortunately, R possesses some useful additional tools should you forget your file's precise location. You can view textual output of the contents of any folder by using `list.files`. The following example betrays the messiness of my local user directory.

```
R> list.files("/Users/tdavies")
 [1] "bands-SCHIST1L200.txt" "Brass"                 "Desktop"
 [4] "Documents"             "DOS Games"             "Downloads"
 [7] "Dropbox"               "Exercise2-20Data.txt"  "Google Drive"
[10] "iCloud"                "Library"               "log.txt"
[13] "Movies"                "Music"                 "mydatafile.txt"
[16] "OneDrive"              "peritonitis.sav"       "peritonitis.txt"
[19] "Personal9414"          "Pictures"              "Public"
[22] "Research"              "Rintro.tex"            "Rprofile.txt"
[25] "Rstartup.R"            "spreadsheetfile.csv"   "spreadsheetfile.xlsx"
[28] "TakeHome_template.tex" "WISE-P2L"              "WISE-P2S.txt"
[31] "WISE-SCHIST1L200.txt"
```

One important feature to note here, though, is that it can be difficult to distinguish between files and folders. Files will typically have an extension, and folders won't; however, `ISE-P2L`, a file that happens to have no extension, looks no different from any of the listed folders.

You can also find files interactively from R. The `file.choose` command opens your filesystem viewer directly from the R prompt—just as any other program does when you want to open something. Then, you can navigate to the folder of interest, and after you select your file (see Figure 8-3), only a character string is returned.

```
R> file.choose()
[1] "/Users/tdavies/mydatafile.txt"
```

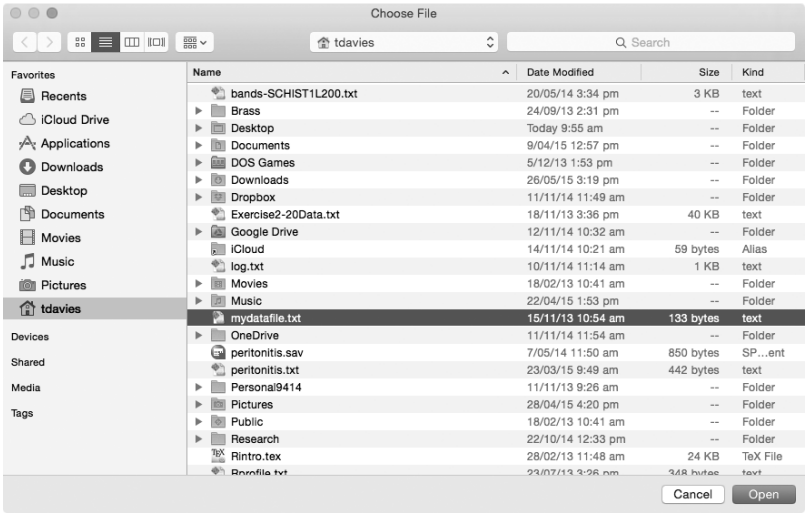


Figure 8-3: My local file navigator opened as the result of a call to `file.choose`. *When the file of interest is opened, the R command returns the full file path to that file as a character string.*

The usefulness of the returned object as a simple character string is clear—it is in precisely the format that's required for a command such as `read.table`. So, calling the following line and selecting *mydatafile.txt*, as in Figure 8-3, will produce an identical result to the explicit use of the file path in `file`, shown earlier:

```
R> mydatafile <- read.table(file=file.choose(),header=TRUE,sep=" ",
                            na.strings="*",stringsAsFactors=FALSE)
```

When importing data into data frames, keep in mind the difference between character string observations and factor observations. No factor attribute information is stored in the plain-text file, but `read.table` will convert non-numeric values into factors by default. Here, you want to keep some

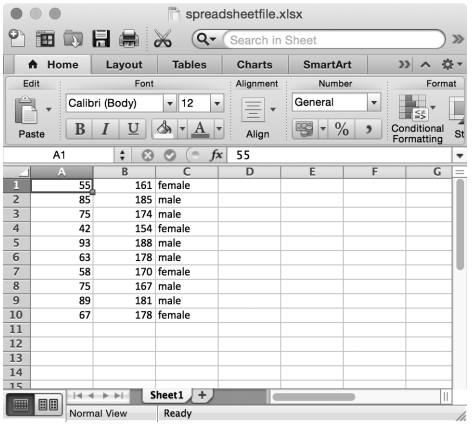
of your data saved as strings, so set `stringsAsFactors=FALSE`, which prevents R from treating all non-numeric elements as factors. This way, `person`, `sex`, and `funny` are all stored as character strings.

You can then overwrite `sex` and `funny` with factor versions of themselves if you want them as that data type.

```
R> mydatafile$sex <- as.factor(mydatafile$sex)
R> mydatafile$funny <- factor(x=mydatafile$funny,levels=c("Low","Med","High")
```

8.2.2 Spreadsheet Workbooks

Next, let's examine some ubiquitous spreadsheet software file formats. The standard file format for Microsoft Office Excel is *.xls* or *.xlsx*. In general, these files are not directly compatible with R. There are some contributed package functions that attempt to bridge this gap (see, for example, `gdata` or `XLConnect`), but it is generally preferable to first export the spreadsheet file to a table format, such as CSV. Consider the hypothetical data from Exercise 7.1 (b), which has been stored in an Excel file called *spreadsheetfile.xlsx*, shown in Figure 8-4.



	A	B	C
1	55	161	female
2	85	185	male
3	75	174	male
4	42	154	female
5	93	188	male
6	63	178	male
7	58	170	female
8	75	167	male
9	89	181	male
10	67	178	female

Figure 8-4: A spreadsheet file of the data from Exercise 7.1 (b)

To read this spreadsheet with R, you should first convert it to a table format. In Excel, File > Save As... provides a wealth of options. Suppose you elect to save the spreadsheet as a comma-separated file, called *spreadsheet.csv*. R has a shortcut version of `read.table`, `read.csv`, for these files.

```
R> spread <- read.csv(file="/Users/tdavies/spreadsheetfile.csv",
                      header=FALSE,stringsAsFactors=TRUE)
R> spread
   V1  V2     V3
1  55 161 female
2  85 185   male
```

```
3  75 174   male
4  42 154 female
5  93 188   male
6  63 178   male
7  58 170 female
8  75 167   male
9  89 181   male
10 67 178 female
```

Here, the `file` argument again specifies the desired file, which has no header, so `header=FALSE`. You set `stringsAsFactors=TRUE` because you do want to treat the `sex` variable (the only non-numeric variable) as a factor. There are no missing values, so you don't need to specify `na.strings`, and by definition, *.csv* files are comma-delimited, which `read.csv` correctly implements by default. The resulting data frame, `spread`, can then be printed in your R console.

As you can see, reading data that's in table format into R is fairly straightforward—you just need to be aware of how the data file is headed and delimited and how missing entries are identified. The simple table format is a natural and common way for data sets to be stored, but if you need to read in a file with a more complicated structure, R and its contributed packages make available some more sophisticated functions. See, for example, the documentation for the `scan` and `readLines` functions, which provide advanced control over how to parse a file. You can also find documentation on `read.table` and `read.csv` by accessing `?read.table` from the prompt.

8.2.3 Web-Based Files

Given an Internet connection, R can read in files from a website with the same `read.table` command. All the same rules concerning headers, delimiters, and missing values remain in place; you just have to specify the URL address of the file instead of a local folder location.

As an example, you'll use the online repository of data sets made available by the *Journal of Statistics Education (JSE)* through the American Statistical Association at *http://www.amstat.org/publications/jse/jse_data_archive.htm*.

One of the first files linked to at the top of this page is the table-format data set *4cdata.txt* (*http://www.amstat.org/publications/jse/v9n2/4cdata.txt*), which contains data on the characteristics of 308 diamonds from an analysis by Chu (2001) based on an advertisement in a Singaporean newspaper.

Figure 8-5 shows the data.

You can look at the documentation file (*4c.txt*) and the accompanying article linked from the JSE site for details on what is recorded in this table. Note that of the five columns, the first and fifth are numeric, and the others would be well represented by factors. The delimiter is blank whitespace, there's no header, and there are no missing values (so you don't have to specify a value used to represent them).

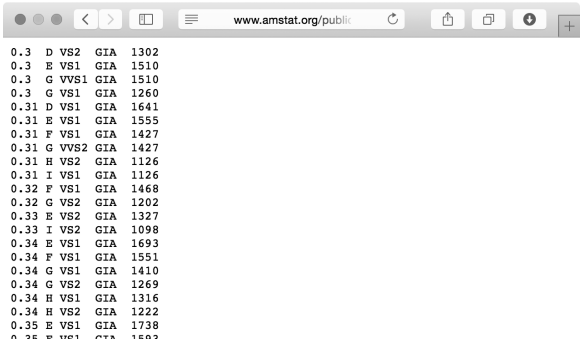


Figure 8-5: A table-format data file found online

With this in mind, you can create a data frame directly from the R prompt simply with the following line:

```
R> diamonds <- read.table("http://www.amstat.org/publications/jse/v9n2/
                          4cdata.txt")
```

Note that you haven't supplied any extra values in this call to `read.table` because the defaults all work just fine. Because there's no header in the table, you can leave the default `header` value `FALSE`. The default value for `sep` is `""`, meaning whitespace, which is exactly what this table uses. The default value for `stringsAsFactors` is `TRUE`, which is what you want for your character string columns. Following the import, you can supply names (based on the information in the documentation) to each column as follows:

```
R> names(diamonds) <- c("Carat","Color","Clarity","Cert","Price")
R> diamonds[1:5,]
  Carat Color Clarity Cert Price
1  0.30     D     VS2  GIA  1302
2  0.30     E     VS1  GIA  1510
3  0.30     G    VVS1  GIA  1510
4  0.30     G     VS1  GIA  1260
5  0.31     D     VS1  GIA  1641
```

Viewing the first five records indicates the data frame is as you would expect.

8.2.4 Other File Formats

Common as they may be, you're not restricted to *.txt* or *.csv* files when it comes to reading your data into R. For example, another common data file has a *.dat* extension. These files can also be imported using `read.table`, though they may contain extra information at the top that must be skipped

using the optional `skip` argument. (This asks for the number of lines at the top of the file that should be ignored before R begins the import.)

As mentioned in Section 8.2.2, there are also contributed packages that can cope with other statistical software files; this task is complicated in part by the possibility of multiple worksheets within a file. The R package `foreign` (R Core Team 2014), available from CRAN, provides support for reading data files used by statistical programs such as Stata, SAS, Minitab, and SPSS.

Other contributed packages on CRAN can help R handle files from various database management systems (DBMSs). For example, the `RODBC` package (Ripley & Lapsley 2013) lets you query Microsoft Access databases and return the results as a data frame object. Other interfaces include the packages `RMySQL` (James & DebRoy 2012) and `RJDBC` (Urbanek 2013).

8.3 Writing Out Data Files and Plots

Writing out new files from data frame objects with R is just as easy as reading in files. R's vector-oriented behavior is a fast and convenient way to recode data sets, and sometimes you might just use the software for that very purpose—reading in data, restructuring it, and writing it back out to a file.

8.3.1 Data Sets

The function for writing table-format files to your computer is `write.table`. You supply a data frame object as `x`, and this function writes its contents to a new file with a specified name, delimiter, and missing value string. For example, the following line takes the `mydatafile` object from Section 8.2 and writes it to a file:

```
R> write.table(x=mydatafile,file="/Users/tdavies/somenewfile.txt",
               sep="@",na="??",quote=FALSE,row.names=FALSE)
```

This command creates a new table-format file called *somenewfile.txt* in the specified folder location, delimited by `@` and with missing values denoted with `??`. Since `mydatafile` has variable names, these are automatically written to the file as a header. The optional logical argument `quote` determines whether to encapsulate each non-numeric entry in double quotes; request no quotes by setting the argument to `FALSE`. Another optional logical argument, `row.names`, asks whether to include the row names of `mydatafile` (in this example, this would just be the numbers `1` to `6`), which you also omit with `FALSE`. The resulting file, shown in Figure 8-6, can be opened in a text editor.

Like `read.csv`, `write.csv` is a shortcut version of the `write.table` function designed specifically for *.csv* files.

Figure 8-6: The contents of somenewfile.txt

8.3.2 Plots and Graphics Files

Like data files, plots can also be written directly to a file. In Chapter 7, you created and displayed plots in an active graphics device. This graphics device needn't be a screen window; it can be a specified file. Instead of displaying the plot immediately on the screen, you can have R follow these steps: open a "file" graphics device, run any/all plotting commands to create the final plot, and close the device. R supports direct writing to *.jpeg*, *.bmp*, *.png*, and *.tiff* files using functions of the same names. For example, the following code uses these three steps to create a *.jpeg* file:

```
R> jpeg(filename="/Users/tdavies/myjpegplot.jpeg",width=600,height=600)
R> plot(1:5,6:10,ylab="a nice ylab",xlab="here's an xlab",
        main="a saved .jpeg plot")
R> points(1:5,10:6,cex=2,pch=4,col=2)
R> dev.off()
null device
          1
```

The file graphics device is opened by a call to `jpeg`, where you provide the intended name of the file and its folder location as `filename`. By default, the dimensions of the device are set to 480×480 pixels, but here you change them to 600×600. You could also set these dimensions by supplying other units (inches, centimeters, or millimeters) to `width` and `height` and by specifying the unit with an optional `units` argument. Once the file is opened, any number of R commands may follow—this example plotted some points and then included some additional points with a second command. The final graphical result is written directly to the file just as it would have been displayed on the screen. When you've finished plotting, you must explicitly close the file device with a call to `dev.off()`, which prints information on the remaining active device (here, "null device" can be loosely interpreted as "nothing is left open"). If `dev.off()` isn't called, then R will continue to output/overwrite any subsequent plotting commands to the aforementioned file. The left panel of Figure 8-7 shows the resulting file.

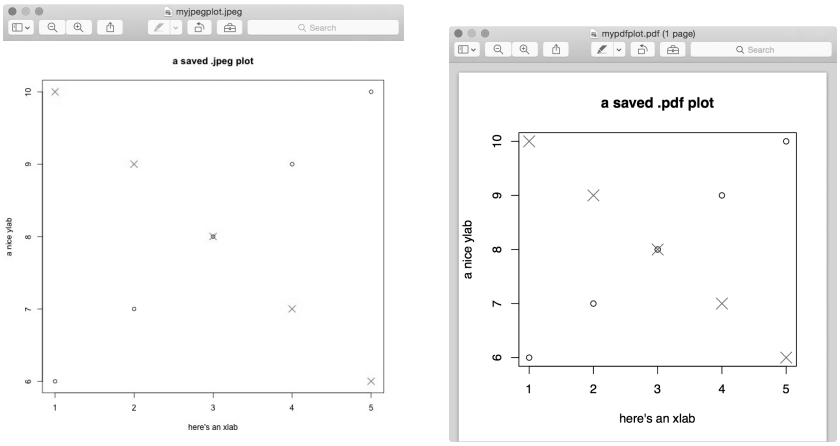


Figure 8-7: R plots that have been written directly to disk: a .jpeg *version (left) and a* .pdf *version (right) of the same plotting commands*

You can also store R plots in other file types, such as PDFs (using the `pdf` function) and EPS files (using the `postscript` function). Though some argument names and default values are different for these functions, they follow the same basic premise. You specify a folder location, a filename, and width/height dimensions; enter your plotting commands; and then close the device with `dev.off()`. The right panel of Figure 8-7 shows the *.pdf* file created with the following code:

```
R> pdf(file="/Users/tdavies/mypdfplot.pdf",width=5,height=5)
R> plot(1:5,6:10,ylab="a nice ylab",xlab="here's an xlab",
       main="a saved .pdf plot")
R> points(1:5,10:6,cex=2,pch=4,col=2)
R> dev.off()
null device
          1
```

Here, you use the same plotting commands as before, and there are just a few minor differences in the code. The argument for the file is `file` (as opposed to `filename`), and the units for `width` and `height` default to inches in `pdf`. The difference of appearance between the two images in Figure 8-7 results primarily from these differences in width and height.

This same process also works for `ggplot2` images. True to style, however, `ggplot2` provides a convenient alternative. The `ggsave` function can be used to write the most recently plotted `ggplot2` graphic to file and performs the device open/close action in one line.

For example, the following code creates and displays a `ggplot2` object from a simple data set.

```
R> foo <- c(1.1,2,3.5,3.9,4.2)
R> bar <- c(2,2.2,-1.3,0,0.2)
R> qplot(foo,bar,geom="blank")
         + geom_point(size=3,shape=8,color="darkgreen")
         + geom_line(color="orange",linetype=4)
```

Now, to save this plot to a file, all you need is the following line:

```
R> ggsave(filename="/Users/tdavies/mypngqplot.png")
Saving 7 x 7 in image
```

This writes the image to a *.png* file in the specified `filename` directory. (Note that dimensions are reported if not set using `width` and `height`; these will vary depending on the size of your graphics device.) The result is shown in Figure 8-8.

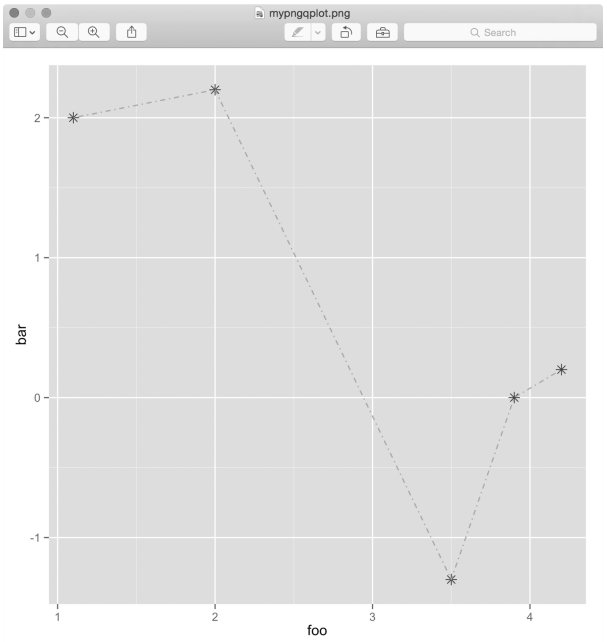


Figure 8-8: The .png *file created using* `ggplot2`*'s* `ggsave` *command*

Beyond just being concise, `ggsave` is convenient in a few other ways. For one, you can use the same command to create a variety of image file types—the type is simply determined by the extension you supply in the `filename` argument. Also, `ggsave` has a range of optional arguments if you want to control the size of the image and the quality or scaling of the graphic.

For more details on saving images from base R graphics, see the `?jpeg`, `?pdf`, and `?postscript` help files. You can consult `?ggsave` for more on saving images with `ggplot2`.

8.4 Ad Hoc Object Read/Write Operations

For the typical R user, the most common input/output operations will probably revolve around data sets and plot images. But if you need to read or write other kinds of R objects, you'll need the `dput` and `dget` commands, which can handle objects in a more ad hoc style.

Suppose, for example, you create this list in the current session:

```
R> somelist <- list(foo=c(5,2,45),
                    bar=matrix(data=c(T,T,F,F,F,F,T,F,T),nrow=3,ncol=3),
                    baz=factor(c(1,2,2,3,1,1,3),levels=1:3,ordered=T))
R> somelist
$foo
[1]  5  2 45

$bar
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE  TRUE
[2,]  TRUE FALSE FALSE
[3,] FALSE FALSE  TRUE

$baz
[1] 1 2 2 3 1 1 3
Levels: 1 < 2 < 3
```

This object can itself be written to a file, which is useful if you want to pass it to a colleague or open it in a new R session elsewhere. Using `dput`, the following line stores the object as a plain-text file that is interpretable by R:

```
R> dput(x=somelist,file="/Users/tdavies/myRobject.txt")
```

In technical terms, this command creates an American Standard Code for Information Interchange (ASCII) representation of the object. As you call `dput`, the object you want to write is specified as `x`, and the folder location and name of the new plain-text file are passed to `file`. Figure 8-9 shows the contents of the resulting file.

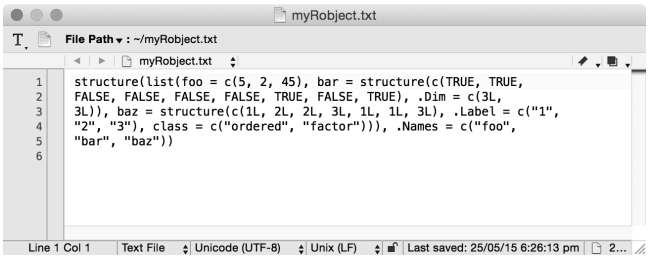


Figure 8-9: myRobject.txt *created by using* `dput` *on* `somelist`

Notice that dput stores all of the members of the object plus any other relevant information, such as attributes. The third element of somelist, for example, is an ordered factor, so it is not enough to simply represent it in the text file as a stand-alone vector. Note also that the name of the object in the R workspace is not stored in *myRobject.txt*.

Now, let's say you want to import this list into an R workspace. If a file has been created with dput, then it can be read into any other workspace using dget.

```
R> newobject <- dget(file="/Users/tdavies/myRobject.txt")
R> newobject
$foo
[1]  5  2 45

$bar
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE  TRUE
[2,]  TRUE FALSE FALSE
[3,] FALSE FALSE  TRUE

$baz
[1] 1 2 2 3 1 1 3
Levels: 1 < 2 < 3
```

The object read from the *myRobject.txt* file using dget (and now assigned to newobject) is the same as the original R object somelist, with all structures and attributes present.

There are some drawbacks to using these commands. For starters, dput is not as reliable a command as write.table because it's sometimes quite difficult for R to create the necessary plain-text representation for an object (fundamental object classes typically cause no problems, but complex user-defined classes can). Also, because they need to store structural information, files created using dput are relatively inefficient both in terms of required space and in terms of how long it takes to execute read and write operations. This becomes more noticeable for objects that contain a lot of data. Nevertheless, dput and dget are useful ways to store or transfer specific objects without having to save an entire workspace.

Exercise 8.1

a. In R's built-in datasets library is the data frame quakes. Make sure you can access this object and view the corresponding help file to get an idea of what this data represents. Then, do the following:
   i. In an existing folder on your machine, write to a table-format file called *q5.txt* the data set obtained by selecting only those

records that correspond to a `mag` of greater than or equal to `5`. Use a delimiting character of `!` and do not include any row names.

ii. Read the file back into your R workspace, calling the object `q5.dframe`.

b. In the contributed package `car`, there's a data frame called `Duncan`, which provides historical data on perceived job prestige in 1950. Install the `car` package and access the `Duncan` data set and its help file. Then, do the following:

i. Write R code that will plot `education` on the *x*-axis and `income` on the *y*-axis, with both *x*- and *y*-axis limits fixed to be [0,100]. Provide appropriate axis labels. For jobs with a value of `prestige` of less than or equal to `80`, use a black ∘ as the point character. For jobs with `prestige` greater than 80, use a blue ●.

ii. Add a legend explaining the difference between the two types of points and then save a 500 × 500 pixel *.png* file of the image.

c. Create a list called `exer` that contains the three data sets `quakes`, `q5.dframe`, and `Duncan`. Then, do the following:

i. Write the list object directly to disk, calling it *Exercise8-1.txt*. Briefly inspect the contents of the file in a text editor.

ii. Read *Exercise8-1.txt* back into your workspace; call the resulting object `list.of.dataframes`. Check that `list.of.dataframes` does indeed contain the previous three data frame objects.

d. In Section 7.4.3, you created a `ggplot2` graphic of 20 observations displayed as the bottom panel of Figure 7-11. Use `ggsave` to save a copy of this plot as a *.tiff* file.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`data`	Load contributed data set	Section 8.1.2, p. 151
`read.table`	Import table-format data file	Section 8.2.1, p. 153
`list.files`	Print specific folder contents	Section 8.2.1, p. 153
`file.choose`	Interactive file selection	Section 8.2.1, p. 154
`read.csv`	Import comma-delimited file	Section 8.2.2, p. 155
`write.table`	Write table-format file to disk	Section 8.3.1, p. 158
`jpeg, bmp, png, tiff`	Write image/plot file to disk	Section 8.3.2, p. 159
`dev.off`	Close file graphics device	Section 8.3.2, p. 159
`pdf, postscript`	Write image/plot file to disk	Section 8.3.2, p. 160
`ggsave`	Write `ggplot2` plot file to disk	Section 8.3.2, p. 161
`dput`	Write R object to file (ASCII)	Section 8.4, p. 162
`dget`	Import ASCII object file	Section 8.4, p. 163

PART II

PROGRAMMING

9

CALLING FUNCTIONS

Before you start writing your own functions in R, it's useful to understand some formalities about how functions are called and interpreted in an active R session. First, you'll look at how *scope* affects how R deals with variable names. You'll see R's rules for naming arguments and objects, and how R locates arguments and other variables when a function is called. Then you'll look at different ways to specify arguments when calling a function.

9.1 Scoping

In comparison to other programming languages, R is fairly relaxed when it comes to rules about naming objects in the workspace. For example, you've used the argument `data` when calling `matrix` (Section 3.1), but `data` is also the name of a ready-to-use function that loads data sets from contributed packages (Section 8.1.2). R doesn't get confused by duplicate names like

this because of its *scoping rules*, which determine precisely how the language compartmentalizes objects and retrieves them in a given situation.

9.1.1 Environments

R enforces scoping rules with virtual *environments*. You can think of environments as separate compartments where data structures and functions are stored. They allow R to distinguish between identical names that are associated with different scopes. Environments are dynamic entities—new environments can be created, and existing environments can be manipulated or removed.

NOTE *Technically speaking, environments don't actually contain items. Rather, they have* pointers *to the location of those items in the computer's memory. But using the "compartment" metaphor and thinking of objects "belonging to" these compartments suffices when you're first getting a general sense of how environments work.*

There are three important kinds of environments: global environments, package environments and namespaces, and local or lexical environments.

Global Environment

The *global environment* is the compartment set aside for the user to define their own objects. Every object you've created or overwritten so far has resided in the global environment of the current R session. In Section 1.3.1, I mentioned that a call to `ls()` lists all the objects, variables, and user-defined functions in the active workspace. More precisely, it prints out the names of everything in the global environment.

Starting with a new R workspace, the following code creates two objects and confirms their existence in the global environment:

```
R> foo <- 4+5
R> bar <- "stringtastic"
R> ls()
[1] "bar" "foo"
```

But what about all the ready-to-use objects and functions in the R workspace? Why aren't those printed alongside `foo` and `bar` as members of this environment? In fact, those objects and functions belong to package-specific environments, described next.

Package Environments and Namespaces

For simplicity, I'll use the term *package environment* rather loosely to refer to the items made available by each package in R. In fact, the structure of R packages in terms of scoping is a bit more complicated. Each package environment as referred to here actually represents several environments that control different aspects of object searching. A package *namespace*, for example, essentially defines the visibility of its functions. (A package can have functions that are both visible and invisible, the latter usually providing

internal support for the former.) Also part of a package environment is an environment related to an *imports* designation, which details any functions or objects belonging to other libraries that are required by the package for its own functionality.

To clarify this, you can think of all the ready-to-use functionality you're working with in this book as belonging to respective package environments. The same is true for the functionality of any contributed packages you've explicitly loaded with a call to `library`. You can use `ls` to list the items in a package environment as follows (using the automatically loaded package `graphics` as an example):

```
R> ls("package:graphics")
 [1] "abline"          "arrows"          "assocplot"       "axis"
 [5] "Axis"            "axis.Date"       "axis.POSIXct"    "axTicks"
 [9] "barplot"         "barplot.default" "box"             "boxplot"
[13] "boxplot.default" "boxplot.matrix"  "bxp"             "cdplot"
[17] "clip"            "close.screen"    "co.intervals"    "contour"
[21] "contour.default" "coplot"          "curve"           "dotchart"
[25] "erase.screen"    "filled.contour"  "fourfoldplot"    "frame"
[29] "grconvertX"      "grconvertY"      "grid"            "hist"
[33] "hist.default"    "identify"        "image"           "image.default"
[37] "layout"          "layout.show"     "lcm"             "legend"
[41] "lines"           "lines.default"   "locator"         "matlines"
[45] "matplot"         "matpoints"       "mosaicplot"      "mtext"
[49] "pairs"           "pairs.default"   "panel.smooth"    "par"
[53] "persp"           "pie"             "plot"            "plot.default"
[57] "plot.design"     "plot.function"   "plot.new"        "plot.window"
[61] "plot.xy"         "points"          "points.default"  "polygon"
[65] "polypath"        "rasterImage"     "rect"            "rug"
[69] "screen"          "segments"        "smoothScatter"   "spineplot"
[73] "split.screen"    "stars"           "stem"            "strheight"
[77] "stripchart"      "strwidth"        "sunflowerplot"   "symbols"
[81] "text"            "text.default"    "title"           "xinch"
[85] "xspline"         "xyinch"          "yinch"
```

Note that this list includes some of the functions you used in Chapter 7, such as `arrows`, `plot`, and `segments`.

Local or Lexical Environments

Each time a function is called in R, a new environment is created called the *local environment* or *lexical environment*. This local environment contains all the objects and variables created in and visible to the function, including any arguments supplied to it upon execution. It's this feature that allows the presence of argument names that are identical to other object names accessible in a given workspace.

For example, say you call `matrix` and pass in the argument `data`, as follows:

```
R> youthspeak <- matrix(data=c("OMG","LOL","WTF","YOLO"),nrow=2,ncol=2)
R> youthspeak
     [,1]  [,2]
[1,] "OMG" "WTF"
[2,] "LOL" "YOLO"
```

Calling this function creates a local environment where `data` refers to the vector you passed in. When the function actually goes about filling the matrix, it begins by looking for `data` in this local environment. That means R isn't confused by other objects or functions named `data` in other environments (such as the `data` function automatically loaded from the `utils` package environment). If a required item isn't found in the local environment, only then does R begin to widen its search for that item (I'll discuss this feature a little more in Section 9.1.2). Once the function has completed, this local environment is automatically removed. The same comments apply to the `nrow` and `ncol` arguments.

9.1.2 Search Path

To access data structures and functions from environments other than the user's global environment, R follows a *search path*. The search path lays out all the environments that a given R session has available to it.

The search path is little more than an instructive order of the present environments that R adheres to when the user (or a function) requests some object. If the object isn't found in one environment, R proceeds to the next one. You can view R's search path at any time using `search()`.

```
R> search()
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

From the command prompt, this path will always begin at the global user environment (`.GlobalEnv`) and end after the `base` package environment (`package:base`). You can think of these as belonging to a hierarchy, with the global environment at the bottom and the `base` package at the top. Alternatively, you can simply imagine an arrow pointing from left to right between each pair of environments. For my current session, if I request a certain object at the R prompt, the program will inspect `.GlobalEnv` → `tools:RGUI"` → `package:stats` → ... → `package:base` in turn, stopping the search when the desired object is found and retrieved. (Note that the presence/absence of `tools:RGUI`, which does not affect the type of functionality discussed in this book, depends on your operating system and use of the built-in graphical user interface.)

9.1.3 Enclosures and the Empty Environment

Whenever you execute code that refers to an object outside the current environment, R starts looking for that variable in the *enclosing* environment. If the function comes from a package, typically its enclosing environment (or *enclosure*) is that same package (or to be more precise, the *namespace* for the package). For a user-defined function or object, the enclosure is the global environment.

If R doesn't find what it's looking for in the immediate enclosure, the search continues along the search path (that is, according to the left-to-right arrows mentioned earlier) until the *empty environment* is reached. The empty environment is not explicitly listed in the output from `search()`, but it's always the final destination after `package:base`. The empty, or *null*, environment is special because it's the only environment without an enclosure, and so it marks the end of the search path.

For example, if you call the following, a number of things happen internally.

```
R> baz <- seq(from=0,to=3,length.out=5)
R> baz
[1] 0.00 0.75 1.50 2.25 3.00
```

R first searches the global environment for a function called `seq`, and when this isn't found, it goes on to search in the enclosing environment, which is the next level up in the search path. If it's not there, R keeps going through the path to the next enclosing environment, searching the packages that have been loaded (automatically or otherwise) until it finds what it's looking for. In this example, R locates `seq` in the built-in `base` package environment. Then it executes the `seq` function (thereby creating a temporary local environment) and assigns the results to a new object, `baz`, which resides in the global environment. In the subsequent call to print `baz`, R begins by searching the global environment and immediately finds and retrieves the requested object.

You can look up the enclosing environment properties of any function using `environment`, as follows:

```
R> environment(seq)
<environment: namespace:base>
R> environment(arrows)
<environment: namespace:graphics>
```

Here, I've identified the package namespace of `base` as the enclosure of the `seq` function and the `graphics` package as the enclosure of the `arrows` function.

The order of the search path is enforced by the fact that each environment possesses information on its enclosure, also referred to as its *parent*. Examining the earlier output from the call `search()`, you can see that the parent of `package:stats`, for example, is `package:graphics`. The specific parent-

child structure is dynamic in the sense that the search path changes when additional libraries are loaded or data frames are `attached`. When you load a contributed package with a call to `library`, this essentially just inserts the desired package in the search path. For example, in Exercise 8.1, you installed the contributed package `car`. After loading this package, your search path will include its contents.

```
R> library("car")
R> search()
 [1] ".GlobalEnv"        "package:car"       "tools:RGUI"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

Note the position of the `car` package environment in the path—inserted directly after the global environment. This is where each subsequently loaded package will be placed (followed by any additional packages it depends upon for its own functionality).

As noted earlier, R will stop searching once it has exhausted the entire search path and reached the empty environment, which has no enclosure. If you request a function or object that you haven't defined, that doesn't exist, or that is perhaps in a contributed package that you've forgotten to load (this is quite a common little mistake), then an error is thrown. These "cannot find" errors are recognizable for both functions and other objects.

```
R> neither.here()
Error: could not find function "neither.here"
R> nor.there
Error: object 'nor.there' not found
```

Environments exist to facilitate the virtual compartmentalization of R's wealth of functionality. This becomes particularly important when there are functions with the same name in different packages in the search path. At that point, *masking*, discussed in Section 12.3, comes into play.

As you get more comfortable with R and want more precise control over how it operates, it's worth investigating in full how R handles environments. For more technical details on this, "How R Searches and Finds Stuff" by Suraj Gupta (*http://obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/*) is a particularly well-written article, complete with diagrams.

9.1.4 Reserved and Protected Names

A few key terms are strictly forbidden from being used as object names in R. These *reserved* names are necessary in order to protect fundamental operations and data types frequently used in the language.

The following identifiers are reserved:

- `if` and `else`
- `for`, `while`, and `in`
- `function`
- `repeat`, `break`, and `next`
- `TRUE` and `FALSE`
- `Inf` and `-Inf`
- `NA`, `NaN`, and `NULL`

I haven't yet covered some of the terms on this list. These items represent the core tools for programming in the R language, and you'll begin to explore them in the following chapter. The last three bullet points include the familiar logical values (Section 4.1) and special terms used to represent things like infinity and missing entries (Section 6.1).

If you try to assign a new value to any of these reserved terms, an error occurs.

```
R> NaN <- 5
Error in NaN <- 5 : invalid (do_set) left-hand side to assignment
```

Since R is case sensitive, you're allowed to assign values to any case-variant of the previous key values, but this can be confusing and is generally not advisable.

```
R> False <- "confusing"
R> nan <- "this is"
R> cat(nan,False)
this is confusing
```

Also be wary of assigning values to `T` and `F`, the abbreviations of `TRUE` and `FALSE`. The full identifiers `TRUE` and `FALSE` are reserved, but the abbreviated versions are not.

```
R> T <- 42
R> F <- TRUE
R> F&&TRUE
[1] TRUE
```

Assigning values to `T` and `F` this way will affect any subsequent code that intends to use `T` and `F` to refer to full-length logical counterparts. The second assignment (`F <- TRUE`) is perfectly legal in R's eyes, but it's extremely confusing given the normal usage of `F` as an abbreviation: the line `F&&TRUE` now represents a `TRUE&&TRUE` comparison! It's best to simply avoid these types of assignments.

The experimentation in this section has resulted in some potentially confusing assigned objects, so if you've been following along, it's prudent at this point to clear the global environment (thereby deleting the objects `False`, `nan`, `T`, and `F` from your workspace). To do this, use the `rm` function as shown next. Using `ls()`, supply a character vector of all objects in the global environment as the argument `list`.

```
R> ls()
[1] "bar"        "baz"        "F"          "False"      "foo"        "nan"
[7] "T"          "youthspeak"
R> rm(list=ls())
R> ls()
character(0)
```

Now the global environment is empty, and calling `ls()` returns an empty character vector (`character(0)`).

Exercise 9.1

a. Identify the first 20 items contained in the built-in and automatically loaded `methods` package. How many items are there in total?

b. Determine the enclosing environment of each of the following functions:
   i. `read.table`
   ii. `data`
   iii. `matrix`
   iv. `jpeg`

c. Use `ls` and a test for character string equality to confirm the function `smoothScatter` is part of the `graphics` package.

9.2 Argument Matching

Another set of rules that determine how R interprets function calls has to do with *argument matching*. So far, you've spelled out all argument names (technically called *tags*) explicitly in each function call, but here you'll see there are various shortcuts that let you call functions more concisely.

9.2.1 Exact

For *exact* matching of arguments, each argument tag is written out fully to specify its value. This is the most exhaustive way to call a function, and for the most part, this is how you've been coding so far. It's helpful to write out full argument names this way when first getting to know R or a new function.

Other benefits of exact matching include the following:

- Exact matching is less prone to mis-specification of arguments than other matching styles.
- The order in which arguments are supplied doesn't matter.
- Exact matching is useful when a function has many possible arguments but you want to specify only a few.

The main drawbacks of exact matching are clear:

- It can be cumbersome for relatively simple operations.
- Exact matching requires the user to remember or look up the full, case-sensitive tags.

As an example, in Section 6.2.1, you used exact matching to execute the following:

```
R> bar <- matrix(data=1:9,nrow=3,ncol=3,dimnames=list(c("A","B","C"),
                                                     c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

This creates a 3×3 matrix object `bar` with a `dimnames` attribute for the rows and columns. Since the argument tags are fully specified, the order of the arguments doesn't matter. You could switch around the arguments, and the function still has all the information it requires.

```
R> bar <- matrix(nrow=3,dimnames=list(c("A","B","C"),c("D","E","F")),ncol=3,
                 data=1:9)
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

This behaves the same way as the previous function call. For the sake of consistency, you usually won't switch around arguments each time you call a function, but this example shows a benefit of exact matching: you don't have to worry about the order of any optional arguments or about skipping them.

9.2.2 Partial

Partial matching lets you identify arguments with an abbreviated tag. This can shorten your code, and it still lets you provide arguments in any order.

Here's another way to call `matrix` that takes advantage of partial matching:

```
R> bar <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,dat=1:9)
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

Notice I've shortened the `nrow`, `dimnames`, and `ncol` argument tags to the first two letters and shortened the `data` argument to the first three. For partial matching, there's no set number of letters you have to provide, as long as each argument is still uniquely identifiable by R for the function being called. Partial matching has the following benefits:

- It requires less code than exact matching.
- Argument tags are still visible (thereby limiting the possibility of misspecification).
- The order of supplied arguments still doesn't matter.

But partial matching also has some limitations. For one, it gets trickier if there are multiple arguments whose tags start with the same letters. Here's an example:

```
R> bar <- matrix(nr=3,di=list(c("A","B","C"),c("D","E","F")),nc=3,d=1:9)
Error in matrix(nr = 3, di = list(c("A", "B", "C"), c("D", "E", "F")),  :
  argument 4 matches multiple formal arguments
```

An error has occurred. The fourth argument tag is designated simply as `d`, which is meant to stand for `data`. This is illegal because another argument, namely `dimnames`, also starts with `d`. Even though `dimnames` is specified separately as `di` earlier in the same line, the call isn't valid.

Drawbacks of partial matching include the following:

- The user must be aware of other potential arguments that can be matched by the shortened tag (even if they aren't specified in the call or have a default value assigned).
- Each tag must have a unique identification.

9.2.3 Positional

The most compact mode of function calling in R is called *positional* matching. This is when you supply arguments without tags, and R interprets them based solely on their order.

Positional matching is usually used for relatively simple functions with only a few arguments, or functions that are very familiar to the user. For this type of matching, you *must* be aware of the precise positions of each

argument you want to supply in the definition of the function. You can find that information in the "Usage" section of the function's help file, or it can be printed to the console with the `args` function. Here's an example:

```
R> args(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
NULL
```

This shows the defined order of arguments of the `matrix` function, as well as the default value for each argument. To construct the matrix `bar` with positional matching once more, execute the following:

```
R> bar <- matrix(1:9,3,3,F,list(c("A","B","C"),c("D","E","F")))
R> bar
  D E F
A 1 4 7
B 2 5 8
C 3 6 9
```

The benefits of positional matching are as follows:

- Shorter, cleaner code, particularly for routine tasks
- No need to remember specific argument tags

Notice that in the previous two matching styles, you didn't need to supply anything for the `byrow` argument, which, by default, is set to `FALSE`. In the current case, you have to provide a value (given here as `F`) for `byrow` as the fourth argument because R relies on position alone to interpret the function call. If you leave out the argument, you get an error, as follows:

```
R> bar <- matrix(1:9,3,3,list(c("A","B","C"),c("D","E","F")))
Error in matrix(1:9, 3, 3, list(c("A", "B", "C"), c("D", "E", "F"))) :
  invalid 'byrow' argument
```

Here R has tried to assign the fourth argument (the list you intended for `dimnames`) as the value for the logical `byrow` argument. This brings us to the drawbacks of positional matching:

- You must look up and exactly match the defined order of arguments.
- Reading code written by someone else can be more difficult, especially when it includes unfamiliar functions.

9.2.4 Mixed

Since each matching style has pros and cons, it's quite common, and perfectly legal, to mix these three styles in a single function call.

For example, you can avoid the type of error shown in the previous example.

```
R> bar <- matrix(1:9,3,3,dim=list(c("A","B","C"),c("D","E","F")))
R> bar
```

Here I've used positional matching for the first three arguments, which are by now familiar to you. At the same time, I've used partial matching to explicitly tell R that the list is meant as a `dimnames` value, not for `byrow`.

9.2.5 Dot-Dot-Dot: Use of Ellipses

Many functions exhibit *variadic* behavior. That is, they can accept any number of arguments, and it's up to the user to decide how many arguments to provide. The functions `c`, `data.frame`, and `list` are all like this. When you call a function like `list`, you can specify any number of members as arguments.

This flexibility is achieved in R through the special *dot-dot-dot* designation (`...`), also called the *ellipsis*. You can see whether an ellipsis is used in a function on the corresponding help page or with `args`. Looking at `data.frame`, notice the first argument slot is an ellipsis:

```
R> args(data.frame)
function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
    stringsAsFactors = default.stringsAsFactors())
NULL
```

This construct is what allows the user to supply any number of data vectors (to become the columns in the final data frame).

When you call a function and supply an argument that can't be matched with one of the function's defined argument tags, normally this would produce an error. But if the function is defined with an ellipsis, any arguments that aren't matched to other argument tags are matched to the ellipsis.

Functions that employ ellipses generally fall into two groups. The first group includes functions such as `c`, `data.frame`, and `list`, where the ellipsis always represents the "main ingredients" in the function call. That is, the objective of the function is to use contents of the ellipsis in the resulting object or output. The second group consists of functions where the ellipsis is meant as a *supplementary* or *potential* repository of optional arguments. This is common when the function of interest calls other *subfunctions* that themselves require additional arguments depending upon the originally supplied items. Rather than explicitly copy all the arguments desired by the subfunction into the argument list of the "parent" function, the latter can be defined including an ellipsis that is subsequently provided to the former.

You can see an example of the ellipsis used in a supplementary role with the generic `plot` function.

```
R> args(plot)
function (x, y, ...)
NULL
```

From examining the arguments, it's clear that optional arguments such as point size (argument tag `cex`) or line type (argument tag `lty`), if supplied, are matched to the ellipsis. These optional arguments are then passed in to the function to be used by various methods that tweak graphical parameters.

Ellipses are a convenient programming tool for writing variadic functions or functions where an unknown number of arguments may be supplied. This will become clearer when you start writing your own functions in Chapter 11. However, when writing functions like this, it's important to properly document the intended use of `...` so the potential users of the function know exactly which arguments can be passed to it, and what those arguments are subsequently used for in execution.

Exercise 9.2

a. Use positional matching with `seq` to create a sequence of values between −4 and 4 that progresses in steps of 0.2.

b. In each of the following lines of code, identify which style of argument matching is being used: exact, partial, positional, or mixed. If mixed, identify which arguments are specified in each style.
   i. `array(8:1,dim=c(2,2,2))`
   ii. `rep(1:2,3)`
   iii. `seq(from=10,to=8,length=5)`
   iv. `sort(decreasing=T,x=c(2,1,1,2,0.3,3,1.3))`
   v. `which(matrix(c(T,F,T,T),2,2))`
   vi. `which(matrix(c(T,F,T,T),2,2),a=T)`

c. Suppose you explicitly ran the plotting function `plot.default` and supplied values to arguments tagged `type`, `pch`, `xlab`, `ylab`, `lwd`, `lty`, and `col`. Use the function documentation to determine which of these arguments fall under the umbrella of the ellipsis.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`ls`	Inspect environment objects	Section 9.1.1, p. 169
`search`	Current search path	Section 9.1.2, p. 170
`environment`	Function environment properties	Section 9.1.3, p. 171
`rm`	Delete objects in workspace	Section 9.1.4, p. 174
`args`	Show function arguments	Section 9.2.3, p. 177

10

CONDITIONS AND LOOPS

To write more sophisticated programs with R, you'll need to control the flow and order of code execution. One fundamental way to do this is by checking a *condition* (or set of conditions) to determine what code should be executed. Another basic control mechanism is the *loop*, which repeats a block of code a certain number of times. In this chapter, we'll explore these core programming techniques using `if-else` statements, `for` and `while` loops, and other control structures.

10.1 What if?

The `if` statement is the key to controlling exactly which operations are carried out in a given chunk of code. An `if` statement runs a block of code only if a certain condition is true. In a functional sense, these constructs allow a program to respond differently depending on whether a condition is `TRUE` or `FALSE`.

10.1.1 Stand-Alone Statement

Let's start with the stand-alone `if` statement, which looks something like this:

```
if(condition){
    do any code here
}
```

The *condition* is placed in parentheses after the `if` keyword. This condition must be an expression that yields a single logical value (`TRUE` or `FALSE`). If it's `TRUE`, the code in the curly braces, {}, will be executed. If the condition isn't satisfied, the code in the curly braces is skipped, and R does nothing (or continues on to execute any code after the closing brace).

Here's a simple example. In the console, store the following:

```
R> a <- 3
R> mynumber <- 4
```

Now, in the R editor, write the following code chunk:

```
if(a<=mynumber){
    a <- a^2
}
```

When this chunk is executed, what will the value of `a` be? It depends on the condition defining the `if` statement, as well as what's actually specified in the braced area. In this case, when the condition `a<=mynumber` is evaluated, the result is `TRUE` since 3 is indeed less than or equal to 4. That means the code inside the curly brackets is executed, which sets `a` to `a^2`, or 9.

Now highlight the entire chunk of code in the editor and send it to the console for evaluation. Remember, you can do this in several ways:

- Copy and paste the selected text from the editor directly into the console.
- From the menu, select **Edit → Run line or selection** in Windows or select **Edit → Execute** in OS X.
- Use the keystroke shortcut such as CTRL-R in Windows or ⌘-RETURN on a Mac.

Once you select and execute the code in the console, you'll see something like this:

```
R> if(a<=mynumber){
+   a <- a^2
+ }
```

Then, look at the object a, shown here:

```
R> a
[1] 9
```

Next, suppose you execute the same if statement again right away. Will a be squared once more, giving 81? Nope! Since a is now 9 and mynumber is still 4, the condition a<=mynumber will be FALSE, and the code in the braces will not be executed; a will remain at 9.

Note that after you send the if statement to the console, each line after the first is prefaced by a +. These + signs do not represent any kind of arithmetic addition; rather, they indicate that R is expecting more input before it begins execution. For example, when a left brace is opened, R will not begin any kind of execution until that section is closed with a right brace. You can change the + symbol by assigning a different character string to the continue component of R's options command, in the same way you reset the prompt in Section 1.2.1. To avoid redundancy, in future examples I'll suppress this repetition of code when it's sent to the console.

The if statement offers a huge amount of flexibility—you can place any kind of code in the braced area, including more if statements (see the upcoming discussion of nesting in Section 10.1.4), enabling your program to make a sequence of decisions.

To illustrate a more complicated if statement, consider the following two new objects:

```
R> myvec <- c(2.73,5.40,2.15,5.29,1.36,2.16,1.41,6.97,7.99,9.52)
R> myvec
 [1] 2.73 5.40 2.15 5.29 1.36 2.16 1.41 6.97 7.99 9.52
R> mymat <- matrix(c(2,0,1,2,3,0,3,0,1,1),5,2)
R> mymat
     [,1] [,2]
[1,]    2    0
[2,]    0    3
[3,]    1    0
[4,]    2    1
[5,]    3    1
```

Use these two objects in the code chunk given here:

```
if(any((myvec-1)>9)||matrix(myvec,2,5)[2,1]<=6){
    cat("Condition satisfied --\n")
    new.myvec <- myvec
    new.myvec[seq(1,9,2)] <- NA
    mylist <- list(aa=new.myvec,bb=mymat+0.5)
    cat("-- a list with",length(mylist),"members now exists.")
}
```

Send this to the console, and it produces the following output:

```
Condition satisfied --
-- a list with 2 members now exists.
```

Indeed, an object `mylist` has been created that you can examine.

```
R> mylist
$aa
 [1]   NA 5.40   NA 5.29   NA 2.16   NA 6.97   NA 9.52

$bb
     [,1] [,2]
[1,]  2.5  0.5
[2,]  0.5  3.5
[3,]  1.5  0.5
[4,]  2.5  1.5
[5,]  3.5  1.5
```

In this example, the condition consists of two parts separated by an OR statement (note the use of the long form `||`, which produces a single logical result). Let's walk through it.

- The first part of the condition looks at `myvec`, takes `1` away from each element in it, and checks whether any of the results are greater than 9. If you run this part on its own, it yields `FALSE`.

  ```
  R> myvec-1
  [1] 1.73 4.40 1.15 4.29 0.36 1.16 0.41 5.97 6.99 8.52
  R> (myvec-1)>9
  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
  R> any((myvec-1)>9)
  [1] FALSE
  ```

- The second part of the condition uses positional matching in a call to `matrix` to construct a two-row, five-column, column-filled matrix using entries of the original `myvec`. Then, the number in the second row of the first column of that result is checked to see whether it's less than or equal to 6, which it is.

  ```
  R> matrix(myvec,2,5)
       [,1] [,2] [,3] [,4] [,5]
  [1,] 2.73 2.15 1.36 1.41 7.99
  [2,] 5.40 5.29 2.16 6.97 9.52
  R> matrix(myvec,2,5)[2,1]
  [1] 5.4
  R> matrix(myvec,2,5)[2,1]<=6
  [1] TRUE
  ```

This means the overall condition being checked by the `if` statement will be `FALSE||TRUE`, which evaluates as `TRUE`.

```
R> any((myvec-1)>9)||matrix(myvec,2,5)[2,1]<=6
[1] TRUE
```

As a result, the code inside the braces is accessed and executed. First, it prints the `"Condition satisfied"` string and copies `myvec` to `new.myvec`. Using `seq`, it then accesses the odd-numbered indexes of `new.myvec` and overwrites them with `NA`. Next, it creates `mylist`. In this list, `new.myvec` is stored in a member named `aa`, and then it takes the original `mymat`, increases all its elements by 0.5, and stores the result in `bb`. Lastly, it prints the length of the resulting list.

Note that `if` statements don't have to match the exact style I'm using here. Some programmers, for example, prefer to open the left brace on a new line after the condition, or some may prefer a different amount of indentation.

10.1.2 Do It, or else...

The `if` statement executes a chunk of code if and only if a defined condition is `TRUE`. If you want something different to happen when the condition is `FALSE`, you can add an `else` declaration. Here's an example in pseudocode:

```
if(condition){
    do any code in here if condition is TRUE
} else {
    do any code in here if condition is FALSE
}
```

After the first set of braces that contains the code to execute if the condition is `TRUE`, you declare `else` followed by a new set of braces where you can place code to run if the condition is `FALSE`.

Let's return to the first example in Section 10.1.1, once more storing these values at the console prompt.

```
R> a <- 3
R> mynumber <- 4
```

In the editor, create a new version of the earlier `if` statement.

```
if(a<=mynumber){
    cat("Condition was",a<=mynumber)
    a <- a^2
} else {
    cat("Condition was",a<=mynumber)
    a <- a-3.5
```

```
}
a
```

Here, you again square `a` if the condition `a<=mynumber` is `TRUE`, but if `FALSE`, `a` is overwritten by itself minus 3.5. You also print text to the console stating whether the condition was met. After resetting `a` and `mynumber` to their original values, the first run of the `if` loop computes `a` as 9, just as earlier, outputting the following:

```
Condition was TRUE
R> a
[1] 9
```

Now, immediately highlight and execute the entire statement again. This time around, `a<=mynumber` will evaluate to `FALSE` and execute the code after `else`.

```
Condition was FALSE
R> a
[1] 5.5
```

10.1.3 Using ifelse for Element-wise Checks

An `if` statement can check the condition of only a single logical value. If you pass in, for example, a vector of logicals for the condition, the `if` statement will only check (and hence operate based on) the very first element. It will issue a warning saying as much, as the following dummy example shows:

```
R> if(c(FALSE,TRUE,FALSE,TRUE,TRUE)){}
Warning message:
In if (c(FALSE, TRUE, FALSE, TRUE, TRUE)) { :
  the condition has length > 1 and only the first element will be used
```

There is, however, a shortcut function available, `ifelse`, which can perform this kind of vector-oriented check in relatively simple cases. To demonstrate how it works, consider the objects `x` and `y` defined as follows:

```
R> x <- 5
R> y <- -5:5
R> y
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

Now, suppose you want to produce the result of `x/y` but with any instance of `Inf` (that is, any instance where `x` is divided by zero) replaced with `NA`. In other words, for each element in `y`, you want to check whether `y` is zero. If so, you want the code to output `NA`, and if not, it should output the result of `x/y`.

As you've just seen, a simple `if` statement won't work here. Since it accepts only a single logical value, it can't run through the entire logical vector produced by `y==0`.

```
R> y==0
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Instead, you can use the element-wise `ifelse` function for this kind of scenario.

```
R> result <- ifelse(test=y==0,yes=NA,no=x/y)
R> result
 [1] -1.000000 -1.250000 -1.666667 -2.500000 -5.000000       NA  5.000000  2.500000
 [9]  1.666667  1.250000  1.000000
```

Using exact matching, this command creates the desired `result` vector in one line. Three arguments must be specified: `test` takes a logical-valued data structure, `yes` provides the element to return if the condition is satisfied, and `no` gives the element to return if the condition is `FALSE`. As noted in the function documentation (which you can access with `?ifelse`), the returned structure will be of "the same length and attributes as `test`."

Exercise 10.1

a. For the two vectors

```
vec1 <- c(2,1,1,3,2,1,0)
vec2 <- c(3,8,2,2,0,0,0)
```

determine, without execution, which of the following `if` statements would result in the string being printed to the console. Then confirm your answers in R.

i. `if((vec1[1]+vec2[2])==10){ cat("Print me!") }`

ii. `if(vec1[1]>=2&&vec2[1]>=2){ cat("Print me!") }`

iii. `if(all((vec2-vec1)[c(2,6)]<7)){ cat("Print me!") }`

iv. `if(!is.na(vec2[3])){ cat("Print me!") }`

b. Using `vec1` and `vec2` from (a), write and execute a line of code that multiplies the corresponding elements of the two vectors together *if* their sum is greater than 3. Otherwise, the code should simply sum the two elements.

c. In the editor, write R code that takes a square character matrix and checks *if* any of the character strings on the diagonal (top left to bottom right) begin with the letter *g*, lowercase or uppercase. If satisfied, these specific entries should be overwritten with the string `"HERE"`. Otherwise, the entire matrix should be replaced with an identity matrix of the same dimensions. Then, try your code on the following matrices, checking the result each time:

i.
```
mymat <- matrix(as.character(1:16),4,4)
```

ii.
```
mymat <- matrix(c("DANDELION","Hyacinthus","Gerbera",
                  "MARIGOLD","geranium","ligularia",
                  "Pachysandra","SNAPDRAGON","GLADIOLUS"),3,3)
```

iii.
```
mymat <- matrix(c("GREAT","exercises","right","here"),2,2,
                  byrow=T)
```

Hint: This requires some thought—you will find the functions `diag` from Section 3.2 and `substr` from Section 4.2.4 useful.

10.1.4 Nesting and Stacking Statements

An `if` statement can itself be placed within the outcome of another `if` statement. By *nesting* or *stacking* several statements, you can weave intricate paths in terms of your programmatic decision-making by checking a number of conditions at various stages during execution.

In the editor, let's modify the `mynumber` example once more as follows:

```
if(a<=mynumber){
    cat("First condition was TRUE\n")
    a <- a^2
    if(mynumber>3){
        cat("Second condition was TRUE")
        b <- seq(1,a,length=mynumber)
    } else {
        cat("Second condition was FALSE")
        b <- a*mynumber
    }
} else {
    cat("First condition was FALSE\n")
    a <- a-3.5
    if(mynumber>=4){
        cat("Second condition was TRUE")
        b <- a^(3-mynumber)
    } else {
        cat("Second condition was FALSE")
        b <- rep(a+mynumber,times=3)
```

```
    }
}
a
b
```

Here you see the same initial decision being made as earlier. The value `a` is squared if it's less than or equal to `mynumber`; if not, it has 3.5 subtracted from it. But now there's another `if` statement within each braced area. If the first condition is satisfied and `a` is squared, you then check whether `mynumber` is greater than 3. If `TRUE`, `b` is assigned `seq(1,a,length=mynumber)`. If `FALSE`, `b` is assigned `a*mynumber`.

If the first condition fails and you subtract 3.5 from `a`, then you check a second condition to see whether `mynumber` is greater than or equal to 4. If it is, then `b` becomes `a^(3-mynumber)`. If it's not, `b` becomes `rep(a+mynumber,times=3)`. Note that I've indented the code within each subsequent braced area to make it easier to see which lines are relevant to each possible decision.

Now, reset `a <- 3` and `mynumber <- 4` either directly in the console or from the editor. Running the previous code chunk, you'll get the following output:

```
First condition was TRUE
Second condition was TRUE
R> a
[1] 9
R> b
[1] 1.000000 3.666667 6.333333 9.000000
```

The result indicates exactly which code was invoked—the first condition and second condition were both `TRUE`. Trying another run of the same code, after first setting

```
R> a <- 6
R> mynumber <- 4
```

you see this output:

```
First condition was FALSE
Second condition was TRUE
R> a
[1] 2.5
R> b
[1] 0.4
```

This time the first condition fails, but the second condition checked inside the `else` statement is `TRUE`.

Alternatively, you could accomplish the same thing by sequentially *stacking* `if` statements and using a combination of logical expressions in each condition. In the following example, you check for the same four situations,

but this time you stack `if` statements by placing a new `if` declaration immediately following an `else` declaration:

```
if(a<=mynumber && mynumber>3){
    cat("Same as 'first condition TRUE and second TRUE'")
    a <- a^2
    b <- seq(1,a,length=mynumber)
} else if(a<=mynumber && mynumber<=3){
    cat("Same as 'first condition TRUE and second FALSE'")
    a <- a^2
    b <- a*mynumber
} else if(mynumber>=4){
    cat("Same as 'first condition FALSE and second TRUE'")
    a <- a-3.5
    b <- a^(3-mynumber)
} else {
    cat("Same as 'first condition FALSE and second FALSE'")
    a <- a-3.5
    b <- rep(a+mynumber,times=3)
}
a
b
```

Just as before, only one of the four braced areas will end up being executed. Comparing this to the nested version, the first two braced areas correspond to what was originally the first condition (`a<=mynumber`) being satisfied, but this time you use `&&` to check two expressions at once. If neither of those two situations is met, this means the first condition is false, so in the third statement, you just have to check whether `mynumber>=4`. For the final `else` statement, you don't need to check any conditions because that statement will be executed only if all the previous conditions were not met.

If you again reset `a` and `mynumber` to 3 and 4, respectively, and execute the stacked statements shown earlier, you get the following result:

```
Same as 'first condition TRUE and second TRUE'
R> a
[1] 9
R> b
[1] 1.000000 3.666667 6.333333 9.000000
```

This produces the same values for `a` and `b` as earlier. If you execute the code again using the second set of initial values (`a` as 6 and `mynumber` as 4), you get the following:

```
Same as 'first condition FALSE and second TRUE'
R> a
[1] 2.5
```

```
R> b
[1] 0.4
```

This again matches the results of using the nested version of the code.

10.1.5 The Old switch-eroo

Let's say you need to choose which code to run based on the value of a single object (a common scenario). You could use a series of `if` statements, where you compare the object with various possible values to produce a logical value for each condition. Here's an example:

```
if(mystring=="Homer"){
    foo <- 12
} else if(mystring=="Marge"){
    foo <- 34
} else if(mystring=="Bart"){
    foo <- 56
} else if(mystring=="Lisa"){
    foo <- 78
} else if(mystring=="Maggie"){
    foo <- 90
} else {
    foo <- NA
}
```

The goal of this code is simply to assign a numeric value to an object `foo`, where the exact number depends on the value of `mystring`. The `mystring` object can take one of the five possibilities shown, or if `mystring` doesn't match any of these, `foo` is assigned `NA`.

This code works just fine as it is. For example, setting

```
R> mystring <- "Lisa"
```

and executing the chunk, you'll see this:

```
R> foo
[1] 78
```

Setting the following

```
R> mystring <- "Peter"
```

and executing the chunk again, you'll see this:

```
R> foo
[1] NA
```

This setup using `if-else` statements is quite cumbersome for such a basic operation, though. R can handle this type of multiple-choice decision in a far more compact form via the `switch` function. For example, you could rewrite the stacked `if` statements as a much shorter `switch` statement as follows:

```
R> mystring <- "Lisa"
R> foo <- switch(EXPR=mystring,Homer=12,Marge=34,Bart=56,Lisa=78,Maggie=90,NA)
R> foo
[1] 78
```

and

```
R> mystring <- "Peter"
R> foo <- switch(EXPR=mystring,Homer=12,Marge=34,Bart=56,Lisa=78,Maggie=90,NA)
R> foo
[1] NA
```

The first argument, `EXPR`, is the object of interest (which can be a numeric or character string). The remaining arguments (formally represented with an ellipsis in the definition of `switch`) provide the values or operations to carry out based on the value of `EXPR`. If `EXPR` is a string, these argument tags must *exactly* match the possible results of `EXPR`. In the previous example, the `switch` statement evaluates to 12 if `mystring` is `"Homer"`, 34 if `mystring` is `"Marge"`, and so on. The final, untagged value, `NA`, indicates the result if `mystring` doesn't match any of the preceding items.

The integer version of `switch` works in a slightly different way. Instead of using tags, the outcome is determined purely with positional matching. Consider the following example:

```
R> mynum <- 3
R> foo <- switch(mynum,12,34,56,78,NA)
R> foo
[1] 56
```

Here, you provide an integer `mynum` as the first argument, and it's positionally matched to `EXPR`. The example code then shows five untagged arguments: `12` to `NA`. The `switch` function simply returns the value in the specific position requested by `mynum`. Since `mynum` is 3, the statement assigns 56 to `foo`. Had `mynum` been 1, 2, 4, or 5, `foo` would've been assigned 12, 34, 78, or `NA`, respectively. Any other value of `mynum` (less than 1 or greater than 5) will return `NULL`.

```
R> mynum <- 0
R> foo <- switch(mynum,12,34,56,78,NA)
R> foo
NULL
```

In these types of situations, the `switch` function behaves the same way as a set of stacked `if` statements, so it can serve as a convenient shortcut. However, if you need to examine multiple conditions at once or you need to execute a more complicated set of operations based on this decision, you'll need to use the explicit `if` and `else` control structures.

Exercise 10.2

a. Write an explicit stacked set of `if` statements that does the same thing as the integer version of the `switch` function illustrated earlier. Test it with `mynum <- 3` and `mynum <- 0`, as in the text.

b. Suppose you are tasked with computing the precise dosage amounts of a certain drug in a collection of hypothetical scientific experiments. These amounts depend upon some predetermined set of "dosage thresholds" (`lowdose`, `meddose`, and `highdose`), as well as a predetermined dose level factor vector named `doselevel`. Look below at items (i–iv) to see the intended form of these objects. Then write a set of nested `if` statements that produce a new numeric vector called `dosage`, according to the following rules:
   - First, *if* there are *any* instances of `"High"` in `doselevel`, perform the following operations:
     * Check *if* `lowdose` is greater than or equal to 10. If so, overwrite `lowdose` with 10; *otherwise*, overwrite `lowdose` by itself divided by 2.
     * Check *if* `meddose` is greater than or equal to 26. If so, overwrite `meddose` by 26.
     * Check *if* `highdose` is less than 60. If so, overwrite `highdose` with 60; *otherwise*, overwrite `highdose` by itself multiplied by 1.5.
     * Create a vector called `dosage` with the value of `lowdose` repeated (`rep`) to match the `length` of `doselevel`.
     * Overwrite the elements in `dosage` corresponding to the index positions of instances of `"Med"` in `doselevel` by `meddose`.
     * Overwrite the elements in `dosage` corresponding to the index positions of instances of `"High"` in `doselevel` by `highdose`.
   - *Otherwise* (in other words, if there are no instances of `"High"` in `doselevel`), perform the following operations:
     * Create a new version of `doselevel`, a factor vector `"Low"` and `"Med"` only, and *label* these with `"Small"` and `"Large"`, respectively (refer to Section 4.3 for details or see `?factor`).

* Check to see *if* `lowdose` is less than 15 *and* `meddose` is less than 35. If so, overwrite `lowdose` by itself multiplied by 2 and overwrite `meddose` by itself plus `highdose`.
* Create a vector called `dosage`, which is the value of `lowdose` repeated (`rep`) to match the `length` of `doselevel`.
* Overwrite the elements in `dosage` corresponding to the index positions of instances of `"Large"` in `doselevel` by `meddose`.

Now, confirm the following:

i. Given

```
lowdose <- 12.5
meddose <- 25.3
highdose <- 58.1
doselevel <- factor(c("Low","High","High","High","Low","Med",
                      "Med")
```

the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1] 10.0 60.0 60.0 60.0 10.0 25.3 25.3
```

ii. Using the same `lowdose`, `meddose`, and `highdose` thresholds as in (i), given

```
doselevel <- factor(c("Low","Low","Low","Med","Low","Med",
                      "Med")
```

the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1] 25.0 25.0 25.0 83.4 25.0 83.4 83.4
```

Also, `doselevel` has been overwritten as follows:

```
R> doselevel
[1] Small Small Small Large Small Large Large
Levels: Small Large
```

iii. Given

```
lowdose <- 9
meddose <- 49
highdose <- 61
doselevel <- factor(c("Low","Med","Med"),
                    levels=c("Low","Med","High"),ordered=T)
```

the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1]  9 49 49
```

Also, `doselevel` has been overwritten as follows:

```
R> doselevel
[1] Small Large Large
Levels: Small Large
```

iv. Using the same `lowdose`, `meddose`, and `highdose` thresholds as (iii), as well as the same `doselevel` as (i), the result of `dosage` after running the nested `if` statements is as follows:

```
R> dosage
[1]  4.5 91.5 91.5 91.5  4.5 26.0 26.0
```

c. Assume the object `mynum` will only ever be a single integer between 0 and 9. Use `ifelse` and `switch` to produce a command that takes in `mynum` and returns a matching character string for all possible values 0, 1, . . . , 9. Supplied with `3`, for example, it should return `"three"`; supplied with `0`, it should return `"zero"`.

10.2 Coding Loops

Another core programming mechanism is the *loop*, which repeats a specified section of code, often while incrementing an index or counter. There are two styles of looping: the `for` loop repeats code as it works its way through a vector, element by element; the `while` loop simply repeats code until a specific condition evaluates to `FALSE`. Looplike behavior can also be achieved with R's suite of `apply` functions, which are discussed in Section 10.2.3.

10.2.1 for Loops

The R `for` loop always takes the following general form:

```
for(loopindex in loopvector){
    do any code in here
}
```

Here, the *`loopindex`* is a placeholder that represents an element in the *`loopvector`*—it starts off as the first element in the vector and moves to the next element with each loop repetition. When the `for` loop begins, it runs the code in the braced area, replacing any occurrence of the *`loopindex`* with the first element of the *`loopvector`*. When the loop reaches the closing brace, the *`loopindex`* is incremented, taking on the second element of the

loopvector, and the braced area is repeated. This continues until the loop reaches the final element of the *loopvector*, at which point the braced code is executed for the final time, and the loop exits.

Here's a simple example written in the editor:

```
for(myitem in 5:7){
    cat("--BRACED AREA BEGINS--\n")
    cat("the current item is",myitem,"\n")
    cat("--BRACED AREA ENDS--\n\n")
}
```

This loop prints the current value of the *loopindex* (which I've named `myitem` here) as it increments from 5 to 7. Here's the output after sending to the console:

```
--BRACED AREA BEGINS--
the current item is 5
--BRACED AREA ENDS--

--BRACED AREA BEGINS--
the current item is 6
--BRACED AREA ENDS--

--BRACED AREA BEGINS--
the current item is 7
--BRACED AREA ENDS--
```

You can manipulate objects that exist outside the loop. Consider the following example:

```
R> counter <- 0
R> for(myitem in 5:7){
+   counter <- counter+1
+   cat("The item in run",counter,"is",myitem,"\n")
+ }
The item in run 1 is 5
The item in run 2 is 6
The item in run 3 is 7
```

Here, I've initially defined an object, `counter`, and set it to zero in the workspace. Then, inside the loop, `counter` is overwritten by itself plus 1. Each time the loop repeats, `counter` increases, and the current value is printed to the console.

Note the difference between using the *loopindex* to directly represent elements in the *loopvector* and using it to represent *indexes* of a vector. The following two loops use these two different approaches to `print` double each number in `myvec`.

```
R> myvec <- c(0.4,1.1,0.34,0.55)
R> for(i in myvec){
+   print(2*i)
+ }
[1] 0.8
[1] 2.2
[1] 0.68
[1] 1.1
R> for(i in 1:length(myvec)){
+   print(2*myvec[i])
+ }
[1] 0.8
[1] 2.2
[1] 0.68
[1] 1.1
```

The first loop uses the *loopindex* `i` to directly represent the elements in `myvec`, printing the value of each element times 2. In the second loop, on the other hand, you use `i` to represent integers in the sequence `1:length(myvec)`. These integers form all the possible index positions of `myvec`, and you use these indexes to extract `myvec`'s elements (once again multiplying each element by 2 and printing the result). Though it takes a slightly longer form, using vector index positions provides more flexibility in terms of how you can use the *loopindex*. This will become clearer when your needs demand more complicated `for` loops, such as in the next example.

Suppose you want to write some code that will inspect any list object and gather information about any matrix objects stored as members in the list. Consider the following list:

```
R> foo <- list(aa=c(3.4,1),bb=matrix(1:4,2,2),cc=matrix(c(T,T,F,T,F,F),3,2),
               dd="string here",ee=matrix(c("red","green","blue","yellow")))
R> foo
$aa
[1] 3.4 1.0

$bb
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$cc
      [,1]  [,2]
[1,]  TRUE  TRUE
[2,]  TRUE FALSE
[3,] FALSE FALSE
```

```
$dd
[1] "string here"

$ee
     [,1]
[1,] "red"
[2,] "green"
[3,] "blue"
[4,] "yellow"
```

Here you've created `foo`, which contains three matrices of varying dimensions and data types. You'll write a `for` loop that goes through each member of a list like this one and checks whether the member is a matrix. If so, it will retrieve the number of rows and columns and the data type of the matrix.

Before you write the `for` loop, you should create some vectors that will store information about the list members: `name` for the list member names, `is.mat` to indicate whether each member is a matrix (with `"Yes"` or `"No"`), `nc` and `nr` to store the number of rows and columns for each matrix, and `data.type` to store the data type of each matrix.

```
R> name <- names(foo)
R> name
[1] "aa" "bb" "cc" "dd" "ee"
R> is.mat <- rep(NA,length(foo))
R> is.mat
[1] NA NA NA NA NA
R> nr <- is.mat
R> nc <- is.mat
R> data.type <- is.mat
```

Here, you store the names of the members of `foo` as `name`. You also set up `is.mat`, `nr`, `nc`, and `data.type`, which are all assigned vectors of length `length(foo)` filled with `NA`s. These values will be updated as appropriate by your `for` loop, which you're now ready to write. Consider the following code in the editor.

```
for(i in 1:length(foo)){
    member <- foo[[i]]
    if(is.matrix(member)){
        is.mat[i] <- "Yes"
        nr[i] <- nrow(member)
        nc[i] <- ncol(member)
        data.type[i] <- class(as.vector(member))
    } else {
        is.mat[i] <- "No"
    }
```

```
}
bar <- data.frame(name,is.mat,nr,nc,data.type,stringsAsFactors=FALSE)
```

Initially, set up the *loopindex* `i` so that it will increment through the index positions of `foo` (the sequence `1:length(foo)`). In the braced code, the first command is to write the member of `foo` at position `i` to an object `member`. Next, you can check whether that member is a matrix using `is.matrix` (refer to Section 6.2.3). If `TRUE`, you do the following: the `i`th position of `is.mat` vector is set as `"Yes"`; the `i`th element of `nr` and `nc` is set as the number of rows and number of columns of `member`, respectively; and the `i`th element of `data.type` is set as the result of `class(as.vector(member))`. This final command first coerces the matrix into a vector with `as.vector` and then uses the `class` function (covered in Section 6.2.2) to find the data type of the elements.

If `member` isn't a matrix and the `if` condition fails, the corresponding entry in `is.mat` is set to `"No"`, and the entries in the other vectors aren't changed (so they will remain `NA`).

After the loop is run, a data frame `bar` is created from the vectors (note the use of `stringsAsFactors=FALSE` in order to prevent the character string vectors in `bar` being automatically converted to factors; see Section 5.2.1). After executing the code, `bar` looks like this:

```
R> bar
  name is.mat nr nc data.type
1   aa     No NA NA      <NA>
2   bb    Yes  2  2   integer
3   cc    Yes  3  2   logical
4   dd     No NA NA      <NA>
5   ee    Yes  4  1 character
```

As you can see, this matches the nature of the matrices present in the list `foo`.

You can also nest `for` loops, just like `if` statements. When a `for` loop is nested in another `for` loop, the inner loop is executed in full before the outer loop *loopindex* is incremented, at which point the inner loop is executed all over again. Consider the following objects:

```
R> loopvec1 <- 5:7
R> loopvec1
[1] 5 6 7
R> loopvec2 <- 9:6
R> loopvec2
[1] 9 8 7 6
R> foo <- matrix(NA,length(loopvec1),length(loopvec2))
R> foo
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
```

```
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

The following nested loop fills `foo` with all possible pairwise multiples of the integers in `loopvec1` and `loopvec2`:

```
R> for(i in 1:length(loopvec1)){
+   for(j in 1:length(loopvec2)){
+       foo[i,j] <- loopvec1[i]*loopvec2[j]
+   }
+ }
R> foo
     [,1] [,2] [,3] [,4]
[1,]   45   40   35   30
[2,]   54   48   42   36
[3,]   63   56   49   42
```

Note that nested loops require a unique *`loopindex`* for each use of `for`. In this case, the *`loopindex`* is `i` for the outer loop and `j` for the inner loop. When the code is executed, `i` is first assigned `1`, the inner loop begins, and then `j` is also assigned `1`. The only command in the inner loop is to take the product of the `i`th element of `loopvec1` and the `j`th element of `loopvec2` and assign it to row `i`, column `j` of `foo`. The inner loop repeats until `j` reaches `length(loopvec2)` and fills the first row of `foo`; then `i` increments, and the inner loop is started up again. The entire procedure is complete after `i` reaches `length(loopvec1)` and the matrix is filled.

Inner *`loopvector`*s can even be defined to match the current value of the *`loopindex`* of the outer loop. Using `loopvec1` and `loopvec2` from earlier, here's the code:

```
R> foo <- matrix(NA,length(loopvec1),length(loopvec2))
R> foo
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
R> for(i in 1:length(loopvec1)){
+   for(j in 1:i){
+       foo[i,j] <- loopvec1[i]+loopvec2[j]
+   }
+ }
R> foo
     [,1] [,2] [,3] [,4]
[1,]   14   NA   NA   NA
[2,]   15   14   NA   NA
[3,]   16   15   14   NA
```

Here, the `i`th row, `j`th column element of `foo` is filled with the sum of `loopvec1[i]` and `loopvec2[j]`. However, the inner loop values for `j` are now decided based on the value of `i`. For example, when `i` is `1`, the inner *`loopvector`* is `1:1`, so the inner loop executes only once before returning to the outer loop. With `i` as `2`, the inner *`loopvector`* is then `1:2`. This makes it so each row of `foo` is only partially filled. Extra care must be taken when programming loops this way. Here, for example, the values for `j` depend on the length of `loopvec1`, so an error will occur if `length(loopvec1)` is greater than `length(loopvec2)`.

Any number of `for` loops can be nested, but the computational expense can become a problem if nested loops are used unwisely. Loops in general add some computational cost, so to produce more efficient code in R, you should always ask "Can I do this in a vector-oriented fashion?" Only when the individual operations are not possible or straightforward to achieve en masse should you explore an iterative, looped approach. You can find some relevant and valuable comments on R loops and associated best-practice coding in Ligges & Fox (2008).

Exercise 10.3

a. In the interests of efficient coding, rewrite the nested loop example from this section, where the matrix `foo` was filled with the multiples of the elements of `loopvec1` and `loopvec2`, using only a single `for` loop.

b. In Section 10.1.5, you used the command

```
switch(EXPR=mystring,Homer=12,Marge=34,Bart=56,Lisa=78,Maggie=90,
       NA)
```

to return a number based on the supplied value of a single character string. This line won't work if `mystring` is a character vector. Write some code that will take a character vector and return a vector of the appropriate numeric values. Test it on the following vector:

```
c("Peter","Homer","Lois","Stewie","Maggie","Bart")
```

c. Suppose you have some list `mylist` that can contain other lists as possible members, though these "member lists" cannot themselves contain member lists. Write nested loops that can search any possible `mylist` defined in this way and count how many matrices are present. Hint: Simply set up a counter before commencing the loops that is incremented each time a matrix is found, regardless of whether it is a straightforward member of `mylist` or it is a member of a member list of `mylist`.

Then confirm the following:

i. That the answer is 4 if you have the following:

```
mylist <- list(aa=c(3.4,1),bb=matrix(1:4,2,2),
               cc=matrix(c(T,T,F,T,F,F),3,2),dd="string here",
               ee=list(c("hello","you"),matrix(c("hello",
                                                 "there"))),
               ff=matrix(c("red","green","blue","yellow")))
```

ii. That the answer is 0 if you have the following:

```
mylist <- list("tricked you",as.vector(matrix(1:6,3,2)))
```

iii. That the answer is 2 if you have the following:

```
mylist <- list(list(1,2,3),list(c(3,2),2),list(c(1,2),
               matrix(c(1,2))),
               rbind(1:10,100:91))
```

10.2.2 while Loops

To use `for` loops, you must know, or be able to easily calculate, the number of times the loop should repeat. In situations where you don't know how many times the desired operations need to be run, you can turn to the `while` loop, which takes the following general form:

```
while(loopcondition){
    do any code in here
}
```

A `while` loop uses a single logical-valued *`loopcondition`* to control how many times it repeats. Upon execution, the *`loopcondition`* is evaluated. If `TRUE`, the braced area code is executed line by line as usual until complete, at which point the *`loopcondition`* is checked again. The code block is repeated as long as the *`loopcondition`* evaluates to `TRUE`. The loop terminates only when the condition evaluates to `FALSE` (and it does so immediately—the braced code is *not* run one last time).

This means the operations carried out in the braced area must somehow cause the loop to exit (either by affecting the *`loopcondition`* somehow or by declaring `break`, which you'll see a little later). If not, the loop will keep repeating forever, creating an *infinite loop*, which will freeze the console (and, depending on the operations specified inside the braced area, R can crash because of memory constraints). If that occurs, you can terminate the loop in the R user interface by clicking the Stop button in the top menu or by pressing ESC.

As a simple example of a `while` loop, consider the following code:

```
myval <- 5
while(myval<10){
    myval <- myval+1
    cat("\n'myval' is now",myval,"\n")
    cat("'mycondition' is now",myval<10,"\n")
}
```

Here, you set a new object `myval` to `5`. Then you start a `while` loop with the condition `myval<10`. Since this is `TRUE` to begin with, you enter the braced area. Inside the loop you increment `myval` by 1, print its current value, and print the logical value of the condition `myval<5`. The loop continues until the condition `myval<10` is `FALSE` at the next evaluation. Execute the code chunk, and you see the following:

```
'myval' is now 6
'mycondition' is now TRUE

'myval' is now 7
'mycondition' is now TRUE

'myval' is now 8
'mycondition' is now TRUE

'myval' is now 9
'mycondition' is now TRUE

'myval' is now 10
'mycondition' is now FALSE
```

As expected, the loop repeats until `myval` is set to `10`, at which point `myval<10` returns `FALSE` and the loop exits.

In more complicated settings, it's often useful to set the *`loopcondition`* to be a separate object so that you can modify it as necessary within the braced area. For the next example, you'll use a `while` loop to iterate through an integer vector and create an identity matrix (see Section 3.3.2) with the dimension matching the current integer. This loop should stop when it reaches a number in the vector that is greater than 5 or when it reaches the end of the integer vector.

In the editor, define some initial objects, followed by the loop itself.

```
mylist <- list()
counter <- 1
mynumbers <- c(4,5,1,2,6,2,4,6,6,2)
mycondition <- mynumbers[counter]<=5
while(mycondition){
    mylist[[counter]] <- diag(mynumbers[counter])
```

```
    counter <- counter+1
    if(counter<=length(mynumbers)){
        mycondition <- mynumbers[counter]<=5
    } else {
        mycondition <- FALSE
    }
}
```

The first object, `mylist`, will store all the matrices that the loop creates. You'll use the vector `mynumbers` to provide the matrix sizes, and you'll use `counter` and `mycondition` to control the loop.

The *`loopcondition`*, `mycondition`, is initially set to `TRUE` since the first element of `mynumbers` is less than or equal to 5. Inside the loop, the first line uses double square brackets and the value of `counter` to dynamically create a new entry at that position in `mylist` (you did this earlier with named lists in Section 5.1.3). This entry is assigned an identity matrix whose size matches the corresponding element of `mynumbers`. Next, the `counter` is incremented, and now you have to update `mycondition`. Here you want to check whether `mynumbers[counter]<=5`, but you also need to check whether you've reached the end of the integer vector (otherwise, you can end up with an error by trying to retrieve an index position outside the range of `mynumbers`). So, you can use an `if` statement to first check the condition `counter<=length(mynumbers)`. If `TRUE`, then set `mycondition` to the outcome of `mynumbers[counter]<=5`. If not, this means you've reached the end of `mynumbers`, so you make sure the loop exits by setting `mycondition <- FALSE`.

Execute the loop with those predefined objects, and it will produce the `mylist` object shown here:

```
R> mylist
[[1]]
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1

[[2]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1

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

```
[[4]]
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

As expected, you have a list with four members—identity matrices of size 4×4, 5×5, 1×1, and 2×2—matching the first four elements of `mynumbers`. The loop stopped executing when it reached the fifth element of `mynumbers` (`6`) since that's greater than 5.

Exercise 10.4

a. Based on the most recent example of storing identity matrices in a list, determine, without execution, what the resulting `mylist` would look like for each of the following possible `mynumbers` vectors:

   i. `mynumbers <- c(2,2,2,2,5,2)`
   ii. `mynumbers <- 2:20`
   iii. `mynumbers <- c(10,1,10,1,2)`

   Then, confirm your answers in R (note you'll also have to reset the initial values of `mylist`, `counter`, and `mycondition` each time, just as in the text).

b. For this problem, I'll introduce the *factorial* operator. The factorial of a non-negative integer x, expressed as $x!$, refers to x multiplied by the product of all integers less than x, down to 1. Formally, it is written like this:

$$\text{“}x \text{ factorial”} = x! = x \times (x - 1) \times (x - 2) \times \ldots \times 1$$

   Note that there is a special case of *zero factorial*, which is always 1. That is:

$$0! = 1$$

   For example, to work out 3 factorial, you compute the following:

$$3 \times 2 \times 1 = 6$$

   To work out 7 factorial, you compute the following:

$$7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$$

   Write a `while` loop that computes and stores as a new object the factorial of any non-negative integer `mynum` by decrementing `mynum` by `1` at each repetition of the braced code.

Using your loop, confirm the following:

i. That the result of using `mynum <- 5` is `120`
ii. That using `mynum <- 12` yields `479001600`
iii. That having `mynum <- 0` correctly returns `1`

c. Consider the following code, where the operations in the braced area of the `while` loop have been omitted:

```
mystring <- "R fever"
index <- 1
ecount <- 0
result <- mystring
while(ecount<2 && index<=nchar(mystring)){
    # several omitted operations #
}
result
```

Your task is to complete the code in the braced area so it inspects `mystring` character by character until it reaches the second instance of the letter *e* or the end of the string, whichever comes first. The `result` object should be the entire character string if there is no second *e* or the character string made up of all the characters up to, but not including, the second *e* if there is one. For example, `mystring <- "R fever"` should provide `result` as `"R fev"`. This must be achieved by following these operations in the braces:

1. Use `substr` (Section 4.2.4) to extract the single character of `mystring` at position `index`.
2. Use a check for equality to determine whether this single-character string is either `"e"` or `"E"`. If so, increase `ecount` by `1`.
3. Next, perform a separate check to see whether `ecount` is equal to `2`. If so, use `substr` to set `result` equal to the characters between `1` and `index-1` inclusive.
4. Increment `index` by `1`.

Test your code—ensure the previous `result` for `mystring <- "R fever"`. Furthermore, confirm that using `mystring <- "beautiful"` provides `result` as `"beautiful"`; `mystring <- "ECCENTRIC"` provides `result` as `"ECC"`; `mystring <- "ElAbOrAte"` provides `result` as `"ElAbOrAt"`; and `mystring <- "eeeeek!"` provides `result` as `"e"`.

10.2.3 Implicit Looping with apply

In some situations, especially for relatively routine `for` loops (such as executing some function on each member of a list), you can avoid some of the details associated with explicit looping by using the `apply` function. The `apply`

function is the most basic form of implicit looping—it takes a function and applies it to each *margin* or dimension of an array.

For a simple illustrative example, let's say you have the following matrix:

```
R> foo <- matrix(1:12,4,3)
R> foo
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Say you want to find the sum of each row. If you call

```
R> sum(foo)
[1] 78
```

you just get the grand total of all elements, which is not what you want. Instead, you could use a `for` loop like this one:

```
R> row.totals <- rep(NA,times=nrow(foo))
R> for(i in 1:nrow(foo)){
+   row.totals[i] <- sum(foo[i,])
+ }
R> row.totals
[1] 15 18 21 24
```

This cycles through each row and stores the sum in (`row.totals` earlier). But you can use `apply` to get the same result in a more compact form. To call `apply`, you have to specify at least three arguments. The first argument, `X`, is the object you want to cycle through. The next argument, `MARGIN`, takes an integer that flags which margin of `X` to operate on (rows, columns, etc.). Finally, `FUN` provides the function you want to perform on each margin. With the following call, you get the same result as the earlier `for` loop.

```
R> row.totals2 <- apply(X=foo,MARGIN=1,FUN=sum)
R> row.totals2
[1] 15 18 21 24
```

To instruct R to sum each column instead, simply change the `MARGIN` argument to `2`.

```
R> apply(X=foo,MARGIN=2,FUN=sum)
[1] 10 26 42
```

The `MARGIN` index follows the positional order of the dimension for matrices and arrays, as discussed in Chapter 3—`1` always refers to rows, `2` to columns, `3` to layers, `4` to blocks, and so on. The operations supplied to

FUN should be appropriate for the MARGIN selected. So, if you select rows or columns with MARGIN=1 or MARGIN=2, make sure the FUN function is appropriate for vectors. Or if you have a three-dimensional array and use apply with MARGIN=3, this selects the layers of the array. Since each layer is a *matrix*, be sure to set FUN to a function appropriate for matrices. Here's an example:

```
R> bar <- array(1:18,dim=c(3,3,2))
R> bar
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

Then, make the following call:

```
R> apply(bar,3,FUN=diag)
     [,1] [,2]
[1,]    1   10
[2,]    5   14
[3,]    9   18
```

This extracts the diagonal elements of each of the matrix layers of bar. Each call to diag on a matrix returns a vector, and these vectors are returned as columns of a new matrix. The FUN argument can also be any appropriate user-defined function, and you'll look at some examples of using apply with your own functions in Chapter 11.

There are different flavors of the basic apply function. The tapply function, for example, performs operations on subsets of the object of interest, where those subsets are defined in terms of one or more factor vectors. As an example, let's return to the code from Section 8.2.3, which reads in a web-based data file on diamond pricing, sets appropriate variable names of the data frame, and displays the first five records.

```
R> diamonds <- read.table("http://www.amstat.org/publications/jse/v9n2/
                          4cdata.txt")
R> names(diamonds) <- c("Carat","Color","Clarity","Cert","Price")
R> diamonds[1:5,]
  Carat Color Clarity Cert Price
1  0.30     D     VS2  GIA  1302
```

```
2  0.30     E     VS1  GIA  1510
3  0.30     G    VVS1  GIA  1510
4  0.30     G     VS1  GIA  1260
5  0.31     D     VS1  GIA  1641
```

To add up the total value of the diamonds present but separated according to `Color`, you can use `tapply` like this:

```
R> tapply(diamonds$Price,INDEX=diamonds$Color,FUN=sum)
     D      E      F      G      H      I
113598 242349 392485 287702 302866 207001
```

This sums the relevant elements of the target vector `diamonds$Price`. The corresponding factor vector `diamonds$Color` is passed to `INDEX`, and the function of interest is specified with `FUN=sum` exactly as earlier.

Another particularly useful alternative is `lapply`, which can operate member by member on a list. In Section 10.2.1, recall you wrote a `for` loop to inspect matrices in the following list:

```
R> baz <- list(aa=c(3.4,1),bb=matrix(1:4,2,2),cc=matrix(c(T,T,F,T,F,F),3,2),
               dd="string here",ee=matrix(c("red","green","blue","yellow")))
```

Using `lapply`, you can check for matrices in the list with a single short line of code.

```
R> lapply(baz,FUN=is.matrix)
$aa
[1] FALSE

$bb
[1] TRUE

$cc
[1] TRUE

$dd
[1] FALSE

$ee
[1] TRUE
```

Note that no margin or index information is required for `lapply`; R knows to apply `FUN` to each member of the specified list. The returned value is itself a list. Another variant, `sapply`, returns the same results as `lapply` but in an array form.

```
R> sapply(baz,FUN=is.matrix)
   aa    bb    cc    dd    ee
FALSE  TRUE  TRUE FALSE  TRUE
```

Here, the result is provided as a vector. In this example, `baz` has a `names` attribute that is copied to the corresponding entries of the returned object.

Other variants of `apply` include `vapply`, which is similar to `sapply` albeit with some relatively subtle differences, and `mapply`, which can operate on multiple vectors or lists at once.

All of R's `apply` functions allow for additional arguments to be passed to `FUN`; most of them do this via an ellipsis. For example, take another look at the matrix `foo`:

```
R> apply(foo,1,sort,decreasing=TRUE)
     [,1] [,2] [,3] [,4]
[1,]    9   10   11   12
[2,]    5    6    7    8
[3,]    1    2    3    4
```

Here you've applied `sort` to each row of the matrix and supplied the additional argument `decreasing=TRUE` to sort the rows from largest to smallest.

Some programmers prefer using the suite of `apply` functions wherever possible to improve the compactness and neatness of their code. However, note that these functions generally do not offer any substantial improvement in terms of computational speed or efficiency over an explicit loop (this is particularly the case with more recent versions of R). Plus, when you're first learning the R language, explicit loops can be easier to read and follow since the operations are laid out clearly line by line.

Exercise 10.5

a. Write an implicit loop that calculates the product of all the column elements of the matrix returned by the call to `apply(foo,1,sort,decreasing=TRUE)`.

b. Convert the following `for` loop to an implicit loop that does exactly the same thing:

```
matlist <- list(matrix(c(T,F,T,T),2,2),
                matrix(c("a","c","b","z","p","q"),3,2),
                matrix(1:8,2,4))
matlist
for(i in 1:length(matlist)){
    matlist[[i]] <- t(matlist[[i]])
}
matlist
```

c. In R, store the following $4 \times 4 \times 2 \times 3$ array as the object `qux`:

```
R> qux <- array(96:1,dim=c(4,4,2,3))
```

That is, it is a four-dimensional array comprised of three blocks, with each block being an array made up of two layers of 4×4 matrices. Then, do the following:

i. Write an implicit loop that obtains the diagonal elements of all second-layer matrices only to produce the following matrix:

```
     [,1] [,2] [,3]
[1,]   80   48   16
[2,]   75   43   11
[3,]   70   38    6
[4,]   65   33    1
```

ii. Write an implicit loop that will return the `dim`ensions of each of the three matrices formed by accessing the fourth column of every matrix in `qux`, regardless of layer or block, wrapped by another implicit loop that finds the row sums of that returned structure, resulting simply in the following vector:

```
[1] 12  6
```

10.3 Other Control Flow Mechanisms

To round off this chapter, you'll look at three more control flow mechanisms: `break`, `next`, and `repeat`. These mechanisms are often used in conjunction with the loops and `if` statements you've seen already.

10.3.1 Declaring break or next

Normally a `for` loop will exit only when the *`loopindex`* exhausts the *`loopvector`*, and a `while` loop will exit only when the *`loopcondition`* evaluates to `FALSE`. But you can also preemptively terminate a loop by declaring `break`.

For example, say you have a number, `foo`, that you want to divide by each element in a numeric vector `bar`.

```
R> foo <- 5
R> bar <- c(2,3,1.1,4,0,4.1,3)
```

Furthermore, let's say you want to divide `foo` by `bar` element by element but want to halt execution if one of the results evaluates to `Inf` (which will result if simply dividing by zero). To do this, you can check each iteration with the `is.finite` function (Section 6.1.1), and you can issue a `break` command to terminate the loop if it returns `FALSE`.

```
R> loop1.result <- rep(NA,length(bar))
R> loop1.result
[1] NA NA NA NA NA NA NA
R> for(i in 1:length(bar)){
+   temp <- foo/bar[i]
+   if(is.finite(temp)){
+       loop1.result[i] <- temp
+   } else {
+       break
+   }
+ }
R> loop1.result
[1] 2.500000 1.666667 4.545455 1.250000       NA       NA       NA
```

Here, the loop divides the numbers normally until it encounters `Inf` after dividing by zero (at the fifth element of `bar`). At that point, the loop ends immediately, leaving the remaining entries of `loop1.result` as they were originally set—`NA`s.

It's best to think of invoking `break` as a fairly drastic move. Often, a programmer will include it only as a safety catch that's meant to highlight or prevent unintended calculations. For more routine operations, it's best to use another method. For example, your previous loop could easily be replicated as a `while` loop or the vector-oriented `ifelse` function, rather than relying on a `break`.

Instead of `break`ing and completely ending a loop, you can use `next` to simply advance to the next iteration and continue execution. Consider the following, where using `next` avoids division by zero:

```
R> loop2.result <- rep(NA,length(bar))
R> loop2.result
[1] NA NA NA NA NA NA NA
R> for(i in 1:length(bar)){
+   if(bar[i]==0){
+       next
+   }
+   loop2.result[i] <- foo/bar[i]
+ }
R> loop2.result
[1] 2.500000 1.666667 4.545455 1.250000       NA 1.219512 1.666667
```

First, the loop checks to see whether the `i`th element of `bar` is zero. If so, `next` is declared, and as a result, R ignores any subsequent lines of code in the braced area of the loop and returns to the top, automatically advancing to the next value of the *`loopindex`*. In the current example, the loop skips the fifth entry of `bar` (leaving the original `NA` value for that place) and continues through the rest of `bar`.

Note that if you use either `break` or `next` in a nested loop, the command will apply only to the innermost loop. Only that inner loop will exit or advance to the next iteration, and any outer loops will continue as normal. For example, let's return to the nested `for` loops from Section 10.2.1 that you used to fill a matrix with multiples of two vectors. This time you'll use `next` in the inner loop to skip certain values.

```
R> loopvec1 <- 5:7
R> loopvec1
[1] 5 6 7
R> loopvec2 <- 9:6
R> loopvec2
[1] 9 8 7 6
R> baz <- matrix(NA,length(loopvec1),length(loopvec2))
R> baz
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
R> for(i in 1:length(loopvec1)){
+   for(j in 1:length(loopvec2)){
+       temp <- loopvec1[i]*loopvec2[j]
+       if(temp>=54){
+           next
+       }
+       baz[i,j] <- temp
+   }
+ }
R> baz
     [,1] [,2] [,3] [,4]
[1,]   45   40   35   30
[2,]   NA   48   42   36
[3,]   NA   NA   49   42
```

The inner loop skips to the `next` iteration if the product of the current elements is greater than or equal to 54. Note the effect applies *only to that innermost loop*—that is, only the `j` *`loopindex`* is preemptively incremented, while `i` is left untouched, and the outer loop continues normally.

Although I've been using `for` loops to illustrate `next` and `break`, they behave the same way inside `while` loops.

10.3.2 Need to repeat It?

Another option for repeating a set of operations is the `repeat` statement. The general definition is simple.

```
repeat{
    do any code in here
}
```

Notice that a `repeat` statement doesn't include any kind of *`loopindex`* or *`loopcondition`*. To stop repeating the code inside the braces, you must use a `break` declaration inside the braced area (usually within an `if` statement); without it, the braced code will repeat without end, creating an infinite loop. To avoid this, you must make sure the operations will at some point cause the loop to reach a `break`.

To see `repeat` in action, you'll use it to calculate the famous mathematical series the *Fibonacci sequence*. The Fibonacci sequence is an infinite series of integers beginning with $1, 1, 2, 3, 5, 8, 13, \ldots$ where each term in the series is determined by the sum of the two previous terms. Formally, if F_n represents the nth Fibonacci number, then you have this:

$$F_{n+1} = F_n + F_{n-1}; \qquad n = 2, 3, 4, 5, \ldots,$$

where

$$F_1 = F_2 = 1.$$

The following `repeat` statement computes and prints the Fibonacci sequence, ending when it reaches a term greater than 150:

```
R> fib.a <- 1
R> fib.b <- 1
R> repeat{
+   temp <- fib.a+fib.b
+   fib.a <- fib.b
+   fib.b <- temp
+   cat(fib.b,", ",sep="")
+   if(fib.b>150){
+       cat("BREAK NOW...\n")
+       break
+   }
+ }
2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

First, the sequence is initialized by storing the first two terms, both 1, as `fib.a` and `fib.b`. Then, the `repeat` statement is entered, and it uses `fib.a` and `fib.b` to compute the next term in the sequence, stored as `temp`. Next, `fib.a` is overwritten to be `fib.b`, and `fib.b` is overwritten to be `temp` so that the two variables move forward through the series. That is, `fib.b` becomes the newly calculated Fibonacci number, and `fib.a` becomes the second-to-last number in the series so far. Use of `cat` then prints the new value of `fib.b` to the console. Finally, a check is made to see whether the latest term is greater than 150, and if it is, `break` is declared.

When you run the code, the braced area is repeated over and over until `fib.b` reaches the first number that is greater than 150, namely, 89 + 144 = 233. Once that happens, the `if` statement condition evaluates as `TRUE`, and R runs into `break`, terminating the loop.

The `repeat` statement is not as commonly used as the standard `while` or `for` loops, but it's useful if you don't want to be bound by formally specifying the *`loopindex`* and *`loopvector`* of a `for` loop or the *`loopcondition`* of a `while` loop. However, with `repeat`, you have to take a bit more caution to prevent infinite loops.

Exercise 10.6

a. Using the same objects from Section 10.3.1,

```
foo <- 5
bar <- c(2,3,1.1,4,0,4.1,3)
```

do the following:

i. Write a `while` loop *without* using `break` (or `next`) that does exactly the same thing as the example concerning `break`. That is, produce the same vector as `loop2.result` in the text.

ii. Obtain the same result as `loop3.result`, the example concerning `next`, using an `ifelse` function instead of a loop.

b. To demonstrate `while` loops in Section 10.2.2, you used the vector

```
mynumbers <- c(4,5,1,2,6,2,4,6,6,2)
```

to progressively fill `mylist` with identity matrices whose dimensions matched the values in `mynumbers`. The loop was instructed to stop when it reached the end of the numeric vector or a number that was greater than 5.

i. Write a `for` loop using a `break` declaration that does the same thing.

ii. Write a `repeat` statement that does the same thing.

c. Suppose you have two lists, `matlist1` and `matlist2`, both filled with numeric matrices as their members. Assume that all members have finite, nonmissing values, but *do not* assume that the dimensions of the matrices are the same throughout. Write a nested pair of `for` loops that aim to create a result list, `reslist`, of all possible *matrix products* (refer to Section 3.3) of the members of the two lists according to the following guidelines:

– The `matlist1` object should be indexed/searched in the outer loop, and the `matlist2` object should be indexed/searched in the inner loop.

– You're interested only in the possible matrix premultiples (Section 3.3.5) of the members of `matlist1` with the members of `matlist2`.

- If a particular multiple isn't possible (that is, if the `ncol` of a member of `matlist1` doesn't match the `nrow` of a member of `matlist2`), then you should skip that multiplication, store the string `"not possible"` at the relevant position in `reslist`, and proceed directly to the `next` matrix multiplication.
- You can define a `counter` that is incremented at each comparison (inside the inner loop) to keep track of the current position of `reslist`.

Note, therefore, that the `length` of `reslist` will be equal to `length(matlist1)*length(matlist2)`. Now, confirm the following results:

i. If you have

```
matlist1 <- list(matrix(1:4,2,2),matrix(1:4),matrix(1:8,4,2))
matlist2 <- matlist1
```

then all members of `reslist` should be `"not possible"` apart from members `[[1]]` and `[[7]]`.

ii. If you have

```
matlist1 <- list(matrix(1:4,2,2),matrix(2:5,2,2),
                 matrix(1:16,4,2))
matlist2 <- list(matrix(1:8,2,4),matrix(10:7,2,2),
                 matrix(9:2,4,2))
```

then only the `"not possible"` members of `reslist` should be `[[3]]`, `[[6]]`, and `[[9]]`.

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`if( ){ }`	Conditional check	Section 10.1.1, p. 182
`if( ){ } else { }`	Check and alternative	Section 10.1.2, p. 185
`ifelse`	Element-wise `if-else` check	Section 10.1.3, p. 187
`switch`	Multiple `if` choices	Section 10.1.5, p. 192
`for( ){ }`	Iterative loop	Section 10.2.1, p. 196
`while( ){ }`	Conditional loop	Section 10.2.2, p. 202
`apply`	Implicit loop by margin	Section 10.2.3, p. 207
`tapply`	Implicit loop by factor	Section 10.2.3, p. 209
`lapply`	Implicit loop by member	Section 10.2.3, p. 209
`sapply`	As `lapply`, array returned	Section 10.2.3, p. 209
`break`	Exit explicit loop	Section 10.3.1, p. 211
`next`	Skip to next loop iteration	Section 10.3.1, p. 212
`repeat{ }`	Repeat code until `break`	Section 10.3.2, p. 214

11

WRITING FUNCTIONS

Defining a function allows you to reuse a chunk of code without endlessly copying and pasting. It also allows other users to access and carry out the same computations on their own data or objects. In this chapter, you'll learn about writing your own R functions. You'll learn how to define and use arguments, how to return output from a function, and how to specialize your functions in other ways.

11.1 The function Command

To define a function, use the `function` command and assign the results to an object name. Once you've done this, you can call the function using that object name just like any other built-in or contributed function in the workplace. This section will walk you through the basics of function creation and discuss some associated issues, such as returning objects and specifying arguments.

11.1.1 Function Creation

A function definition always follows this standard format:

```
functionname <- function(arg1,arg2,arg3,...){
    do any code in here when called
    return(returnobject)
}
```

The *functionname* placeholder can be any valid R object name. This name will be used to call the function. Assign to this *functionname* a call to `function`, followed by parentheses with any arguments you want the function to have. The previous pseudocode includes three argument placeholders plus an ellipsis. Of course, the number of arguments, their names, and whether to include an ellipsis all depend on the particular function you're defining. If the function does not require any arguments, simply include empty parentheses: `()`. If you do include arguments in this definition, note that they are not objects in the workspace and they do not have any type or `class` attributes associated with them—they are merely a declaration of argument names that will be required by *functionname*.

When the function is called, it runs the code in the braced area (also called the *function body* or *body code*). It can include `if` statements, loops, and even other function calls. When it encounters an internal function call during execution, R follows the search rules discussed in Chapter 9. In the braced area, you can use *arg1*, *arg2*, and *arg3*, and they are treated as objects in the function's lexical environment.

Depending on how those declared arguments are used in the body code, each argument may require a certain data type and object structure. If you're writing functions that you intend for others to use, it's important to have sound documentation to say what the function expects.

Often, the function body will include one or more calls to the `return` command. When R encounters a `return` statement during execution, the function exits, returning control to the user at the command prompt. This mechanism also allows you to pass results from operations in the function back to the user. This output is denoted in the previous example by *returnobject*, which is typically assigned a value earlier in the function body. If there is no `return` statement, the function will simply return the object created by the last executed expression (I'll discuss this feature more in Section 11.1.2).

It's time for an example. Let's take the Fibonacci sequence generator from Section 10.3.2 and turn it into a function in the editor.

```
myfib <- function(){
    fib.a <- 1
    fib.b <- 1
    cat(fib.a,", ",fib.b,", ",sep="")
    repeat{
        temp <- fib.a+fib.b
```

```
        fib.a <- fib.b
        fib.b <- temp
        cat(fib.b,", ",sep="")
        if(fib.b>150){
            cat("BREAK NOW...")
            break
        }
    }
}
```

I've named the function `myfib`, and it doesn't use or require any arguments. The body code is identical to the example in Section 10.3.2, except I've added the third line, `cat(fib.a,", ",fib.b,", ",sep="")`, to ensure the first two terms, 1 and 1, are also printed to the screen.

Before you can call `myfib` from the console, you have to send the function definition there. Highlight the code in the editor and press CTRL-R or ⌘-RETURN.

```
R> myfib <- function(){
+   fib.a <- 1
+   fib.b <- 1
+   cat(fib.a,", ",fib.b,", ",sep="")
+   repeat{
+       temp <- fib.a+fib.b
+       fib.a <- fib.b
+       fib.b <- temp
+       cat(fib.b,", ",sep="")
+       if(fib.b>150){
+           cat("BREAK NOW...")
+           break
+       }
+   }
+ }
```

This imports the function into the workspace (if you enter `ls()` at the command prompt, `"myfib"` will now appear in the list of present objects). This step is required anytime you create or modify a function and want to use it from the command prompt.

Now you can call the function from the console.

```
R> myfib()
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

It computes and prints the Fibonacci sequence up to 250, just as instructed.

Rather than printing a fixed set of terms, let's add an argument to control how many Fibonacci numbers are printed. Consider the following new function, `myfib2`, with this modification:

```
myfib2 <- function(thresh){
    fib.a <- 1
    fib.b <- 1
    cat(fib.a,", ",fib.b,", ",sep="")
    repeat{
        temp <- fib.a+fib.b
        fib.a <- fib.b
        fib.b <- temp
        cat(fib.b,", ",sep="")
        if(fib.b>thresh){
            cat("BREAK NOW...")
            break
        }
    }
}
```

This version now takes a single argument, `thresh`. In the body code, `thresh` acts as a threshold determining when to end the `repeat` procedure, halt printing, and complete the function. This means that `thresh` must be supplied as a single numeric value—supplying a character string, for example, would make no sense. After importing the definition of `myfib2` into the console, note the same results as given by the original `myfib` when `thresh=150`.

```
R> myfib2(thresh=150)
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, BREAK NOW...
```

But now you can print the sequence to any limit you want (this time using positional matching to specify the argument).

```
R> myfib2(1000000)
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811,
514229, 832040, 1346269, BREAK NOW...
```

If you want to use the results of a function in future operations (rather than just printing output to the console), you need to return content to the user. Continuing with the current example, here's a Fibonacci function that stores the sequence in a vector and returns it:

```
myfib3 <- function(thresh){
    fibseq <- c(1,1)
    counter <- 2
    repeat{
```

```
        fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
        counter <- counter+1
        if(fibseq[counter]>thresh){
            break
        }
    }
    return(fibseq)
}
```

First you create the vector `fibseq` and assign it the first two terms of the sequence. This vector will ultimately become the *`returnobject`*. You also create a `counter` initialized to `2` to keep track of the current position in `fibseq`. Then the function enters a `repeat` statement, which overwrites `fibseq` with `c(fibseq,fibseq[counter-1]+fibseq[counter])`. That expression constructs a new `fibseq` by appending the sum of the most recent two terms to the contents of what is already stored in `fibseq`. For example, with `counter` starting at `2`, the first run of this line will sum `fibseq[1]` and `fibseq[2]`, appending the result as a third entry onto the original `fibseq`.

Next, `counter` is incremented, and the condition is checked. If the most recent value of `fibseq[counter]` is not greater than `thresh`, the loop repeats. If it is greater, the loop breaks, and you reach the final line of `myfib3`. Calling `return` ends the function and passes out the specified *`returnobject`* (in this case, the final contents of `fibseq`). After importing `myfib3`, consider the following code:

```
R> myfib3(150)
 [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
R> foo <- myfib3(10000)
R> foo
 [1]     1     1     2     3     5     8    13    21    34    55    89   144
[13]   233   377   610   987  1597  2584  4181  6765 10946
R> bar <- foo[1:5]
R> bar
[1] 1 1 2 3 5
```

Here, the first line calls `myfib3` with `thresh` assigned `150`. The output is still printed to the screen, but this isn't the result of the `cat` command as it was earlier; it is the *`returnobject`*. You can assign this *`returnobject`* to a variable, such as `foo`, and `foo` is now just another R object in the global environment that you can manipulate. For example, you use it to create `bar` with a simple vector subset. This would not have been possible with either `myfib` or `myfib2`.

11.1.2 To return or Not to return?

If there's no `return` statement inside a function, the function will end when the last line in the body code has been run, at which point it will return the most recently assigned or created object in the function. If nothing is created, such as in `myfib` and `myfib2` from earlier, the function returns

NULL. To demonstrate this point, consider the following two dummy functions in the editor:

```
dummy1 <- function(){
    aa <- 2.5
    bb <- "string me along"
    cc <- "string 'em up"
    dd <- 4:8
}

dummy2 <- function(){
    aa <- 2.5
    bb <- "string me along"
    cc <- "string 'em up"
    dd <- 4:8
    return(dd)
}
```

The first function, dummy1, simply assigns four different objects in its lexical environment (not the global environment) and does not explicitly return anything. On the other hand, dummy2 creates the same four objects and explicitly returns the last one, dd. If you import and run the two functions, both provide the same return object.

```
R> foo <- dummy1()
R> foo
[1] 4 5 6 7 8
R> bar <- dummy2()
R> bar
[1] 4 5 6 7 8
```

A function will end as soon as it evaluates a return command, without executing any remaining code in the function body. To emphasize this, consider one more version of the dummy function:

```
dummy3 <- function(){
    aa <- 2.5
    bb <- "string me along"
    return(aa)
    cc <- "string 'em up"
    dd <- 4:8
    return(bb)
}
```

Here, dummy3 has two calls to return: one in the middle and one at the end. But when you import and execute the function, it returns only one value.

```
R> baz <- dummy3()
R> baz
[1] 2.5
```

Executing `dummy3` returns only the object `aa` because only the first instance of `return` is executed and the function exits immediately at that point. In the current definition of `dummy3`, the last three lines (the assignment of `cc` and `dd` and the `return` of `bb`) will never be executed.

Using `return` adds another function call to your code, so it has some computational expense. Because of this, some argue that `return` statements should be avoided unless absolutely necessary. But the additional computational cost of the call to `return` is small enough to be negligible for most purposes. Plus, `return` statements can make code more readable, making it easier to see where the author of a function intends it to complete and precisely what is intended to be supplied as output. I'll use `return` throughout the remainder of this work.

Exercise 11.1

a. Write another Fibonacci sequence function, naming it `myfib4`. This function should provide an option to perform either the operations available in `myfib2`, where the sequence is simply printed to the console, or the operations in `myfib3`, where a vector of the sequence is formally returned. Your function should take two arguments: the first, `thresh`, should govern the limit of the sequence (just as in `myfib2` or `myfib3`); the second, `printme`, should be a logical value. If `TRUE`, then `myfib4` should just print; if `FALSE`, then `myfib4` should return a vector. Confirm the correct results arise from the following calls:
   - `myfib4(thresh=150,printme=TRUE)`
   - `myfib4(1000000,T)`
   - `myfib4(150,FALSE)`
   - `myfib4(1000000,printme=F)`

b. In Exercise 10.4 in Section 10.2.2, you were tasked with writing a `while` loop to perform integer factorial calculations.
   i. Using your factorial `while` loop (or writing one if you didn't do so earlier), write your own R function, `myfac`, to compute the factorial of an integer argument `int` (you may assume `int` will always be supplied as a non-negative integer). Perform a quick test of the function by computing 5 factorial, which is 120; 12 factorial, which is 479,001,600; and 0 factorial, which is 1.

ii. Write another version of your factorial function, naming it `myfac2`. This time, you may still assume `int` will be supplied as an integer but not that it will be non-negative. If negative, the function should return `NaN`. Test `myfac2` on the same three values as previously, but also try using `int=-6`.

11.2 Arguments

Arguments are an essential part of most R functions. In this section, you'll consider how R evaluates arguments. You'll also see how to write functions that have default argument values, how to make functions handle missing argument values, and how to pass extra arguments into an internal function call with ellipses.

11.2.1 Lazy Evaluation

An important concept related to handling arguments in many high-level programming languages is *lazy evaluation*. Generally, this refers to the fact that expressions are evaluated only when they are needed. This applies to arguments in the sense that they are accessed and used only at the point they appear in the function body.

Let's see exactly how R functions recognize and use arguments during execution. As a working example to be used throughout this section, you'll write a function to search through a specified list for matrix objects and attempt to post-multiply each with another matrix specified as a second argument. The function will store and return the result in a new list. If no matrices are in the supplied list or if no appropriate matrices (given the dimensions of the multiplying matrix) are present, the function should return a character string informing the user of these facts. You can assume that if there are matrices in the specified list, they will be numeric. Consider the following function, which I'll call `multiples1`:

```
multiples1 <- function(x,mat,str1,str2){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
```

```
            result[[counter]] <- temp%*%mat
        }
    }

    if(counter==0){
        return(str2)
    } else {
        return(result)
    }
}
```

This function takes four arguments; none has a default assigned. The target list to search is intended to be supplied to `x`; the post-multiplying matrix is supplied to `mat`; and two other arguments, `str1` and `str2`, take character strings to return if `x` has no suitable members.

Inside the body code, a vector called `matrix.flags` is created with the `sapply` implicit looping function. This applies the function `is.matrix` to the list argument `x`. The result is a logical vector of equal length as `x`, with `TRUE` elements where the corresponding member of `x` is in fact a matrix. If there are no matrices in `x`, the function hits a `return` statement, which exits the function and outputs the argument `str1`.

If the function did not exit at that point, this means there are indeed matrices in `x`. The next step is to retrieve the matrix member indexes by applying `which` to `matrix.flags`. A `counter` is initialized to `0` to keep track of how many successful matrix multiplications are carried out, and an empty list (`result`) is created to store any results.

Next, you enter a `for` loop. For each member of `indexes`, the loop stores the matrix member at that position as `temp` and checks to see whether it's possible to perform post-multiplication of `temp` by the argument `mat` (to perform the operation, `ncol(temp)` must equal `nrow(mat)`). If the matrices are compatible, `counter` is incremented, and this position of `result` is filled with the relevant calculation. If `FALSE`, nothing is done. The indexer, `i`, then takes on the next value of `indexes` and repeats until completion.

The final procedure in `multiples1` checks whether the `for` loop actually found any compatible matrix products. If no compatibility existed, the braced `if` statement code inside the `for` loop would never have been executed, and the `counter` would remain set to zero. So, if `counter` is still equal to zero upon completion of the loop, the function simply returns the `str2` argument. Otherwise, appropriate results will have been computed, and `multiples1` returns the `result` list, which would have at least one member.

It's time to import and then test the function. You'll use the following three list objects:

```
R> foo <- list(matrix(1:4,2,2),"not a matrix",
               "definitely not a matrix",matrix(1:8,2,4),matrix(1:8,4,2))
R> bar <- list(1:4,"not a matrix",c(F,T,T,T),"??")
R> baz <- list(1:4,"not a matrix",c(F,T,T,T),"??",matrix(1:8,2,4))
```

You'll set the argument mat to the 2×2 identity matrix (post-multiplying by this matrix will simply return the original matrix), and you'll pass in appropriate string messages for str1 and str2. Here's how the function works on foo:

```
R> multiples1(x=foo,mat=diag(2),str1="no matrices in 'x'",
              str2="matrices in 'x' but none of appropriate dimensions given
              'mat'")
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

The function has returned result with the two compatible matrices of foo (members [[1]] and [[5]]). Now let's try it on bar using the same arguments.

```
R> multiples1(x=bar,mat=diag(2),str1="no matrices in 'x'",
              str2="matrices in 'x' but none of appropriate dimensions given
              'mat'")
[1] "no matrices in 'x'"
```

This time, the value of str1 has been returned. The initial check identified that there are no matrices in the list supplied to x, so the function has exited before the for loop. Finally, let's try baz.

```
R> multiples1(x=baz,mat=diag(2),str1="no matrices in 'x'",
              str2="matrices in 'x' but none of appropriate dimensions given
              'mat'")
[1] "matrices in 'x' but none of appropriate dimensions given 'mat'"
```

Here the value of str2 was returned. Though there is a matrix in baz and the for loop in the body code of multiples1 has been executed, the matrix is not compatible for post-multiplication by mat.

Notice that the string arguments str1 and str2 are used only when the argument x does not contain a matrix with the appropriate dimensions. When you applied multiples1 to x=foo, for example, there was no need to use str1 or str2. In part to address that type of situation, R follows behavior known as *lazy evaluation*, which dictates that argument values are sought only at the moment they are required during execution. In this function, str1

and str2 are required only when the input list doesn't have suitable matrices, so you can lazily ignore providing values for these arguments when x=foo.

```
R> multiples1(x=foo,mat=diag(2))
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

This returns the same results as before with no problem whatsoever. Attempting this with bar, on the other hand, doesn't work.

```
R> multiples1(x=bar,mat=diag(2))
Error in multiples1(x = bar, mat = diag(2)) :
  argument "str1" is missing, with no default
```

Here we are quite rightly chastised by R because it requires the value for str1. It informs us that the value is missing and there is no default.

11.2.2 Setting Defaults

The previous example shows one case where it's useful to set default values for certain arguments. Default argument values are also sensible in many other situations, such as when the function has a large number of arguments or when arguments have natural values that are used more often than not. Let's write a new version of the multiples1 function from Section 11.2.1, multiples2, which now includes default values for str1 and str2.

```
multiples2 <- function(x,mat,str1="no valid matrices",str2=str1){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
```

```
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }

    if(counter==0){
        return(str2)
    } else {
        return(result)
    }
}
```

Here, you have given `str1` a default value of `"no valid matrices"` by assigning the string value in the formal definition of the arguments. You've also set a default for `str2` by assigning `str1` to it. If you import and execute this function again on the three lists, you no longer need to explicitly provide values for those arguments.

```
R> multiples2(foo,mat=diag(2))
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

R> multiples2(bar,mat=diag(2))
[1] "no valid matrices"
R> multiples2(baz,mat=diag(2))
[1] "no valid matrices"
```

You can now call the function, whatever the outcome, without being required to specify every argument in full. If you don't want to use the default arguments, you can still specify different values for those arguments when calling the function, in which case the defaults are overwritten.

11.2.3 Checking for Missing Arguments

The `missing` function takes an argument tag and returns a single logical value of `TRUE` if the specified argument isn't found. You can use `missing` to avoid the error you saw in an earlier call to `multiples1`, when `str1` was required but not supplied.

In some situations, the `missing` function can be particularly useful in the body code. Consider another modification of the example function:

```
multiples3 <- function(x,mat,str1,str2){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        if(missing(str1)){
            return("'str1' was missing, so this is the message")
        } else {
            return(str1)
        }
    }

    indexes <- which(matrix.flags)
    counter <- 0
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }

    if(counter==0){
        if(missing(str2)){
            return("'str2' was missing, so this is the message")
        } else {
            return(str2)
        }
    } else {
        return(result)
    }
}
```

The only differences between this version and `multiples1` are in the first and last `if` statements. The first `if` statement checks whether there are no matrices in `x`, in which case it returns a string message. In `multiples1`, that message was always `str1`, but now you use another `if` statement with `missing(str1)` to see whether the `str1` argument actually has a value first. If not, the function returns another character string saying that `str1` was `missing`. A similar alternative is defined for `str2`. Here it is once more importing the function and using `foo`, `bar`, and `baz`:

```
R> multiples3(foo,diag(2))
[[1]]
     [,1] [,2]
```

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

[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

R> multiples3(bar,diag(2))
[1] "'str1' was missing, so this is the message"
R> multiples3(baz,diag(2))
[1] "'str2' was missing, so this is the message"
```

Using `missing` this way permits arguments to be left unspecified in a given function call. It is primarily used when it's difficult to choose a default value for a certain argument, yet the function still needs to handle cases when that argument is not supplied. In the current example, it makes more sense to define defaults for `str1` and `str2`, as you did for `multiples2`, and avoid the extra code required to implement `missing`.

11.2.4 Dealing with Ellipses

In Section 9.2.5, I introduced the ellipsis, also called dot-dot-dot notation. The ellipsis allows extra arguments to be passed in without needing to be explicitly defined in the argument list, and these arguments can then be passed to another function call within the code body. When included in a function definition, the ellipsis is usually (but not always) placed in the last position because it represents a variable number of arguments.

Building on the `myfib3` function from Section 11.1.1, let's use the ellipsis to write a function that can plot the specified Fibonacci numbers.

```
myfibplot <- function(thresh,plotit=TRUE,...){
    fibseq <- c(1,1)
    counter <- 2
    repeat{
        fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
        counter <- counter+1
        if(fibseq[counter]>thresh){
            break
        }
    }

    if(plotit){
        plot(1:length(fibseq),fibseq,...)
    } else {
        return(fibseq)
```

```
    }
}
```

In this function, an `if` statement checks to see whether the `plotit` argument is `TRUE` (which is the default value). If so, then you call `plot`, passing in `1:length(fibseq)` for the *x*-axis coordinates and the Fibonacci numbers themselves for the *y*-axis. After these coordinates, you also pass the ellipsis directly into `plot`. In this case, the ellipsis represents any additional arguments a user might pass in to control the execution of `plot`.

Importing `myfibplot` and executing the following line, the plot in Figure 11-1 pops up in a graphics device.

```
R> myfibplot(150)
```

Here you used positional matching to assign `150` to `thresh`, leaving the default value for `plotit`. The ellipsis is empty in this call.

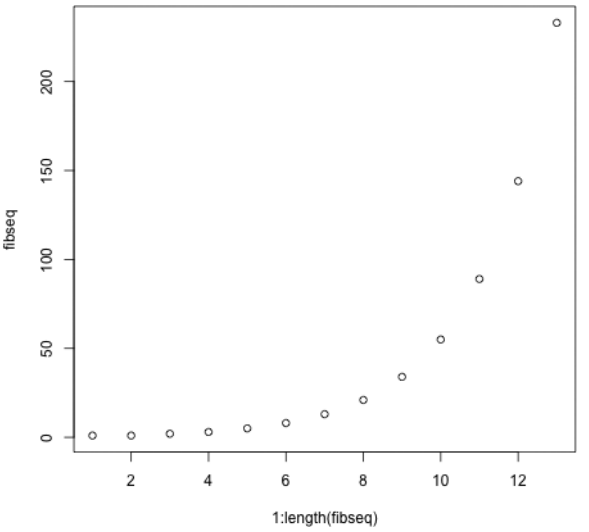


Figure 11-1: The default plot produced by a call to `myfibplot`*, with* `thresh=150`

Since you didn't specify otherwise, R has simply followed the default behavior of `plot`. You can spruce things up by specifying more plotting options. The following line produces the plot in Figure 11-2:

```
R> myfibplot(150,type="b",pch=4,lty=2,main="Terms of the Fibonacci sequence",
             ylab="Fibonacci number",xlab="Term (n)")
```

Here the ellipsis allows you to pass arguments to `plot` through the call to `myfibplot`, even though the particular graphical parameters are not explicitly defined arguments of `myfibplot`.

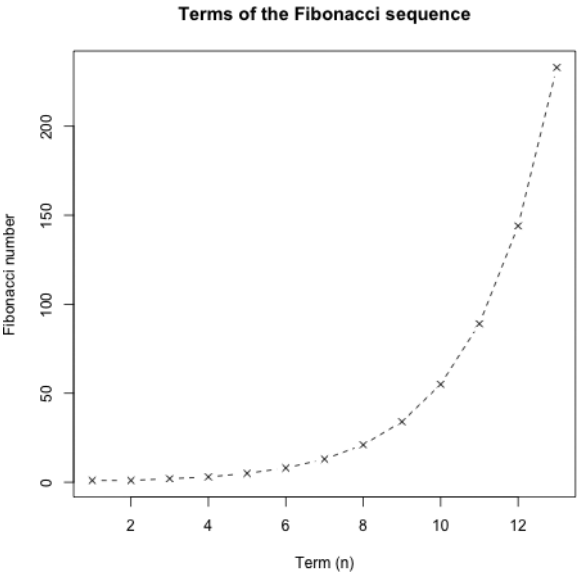


Figure 11-2: A plot produced by a call to `myfibplot` *with graphical parameters passed in using the ellipses*

Ellipses can be convenient, but they require care. The ambiguous `...` can represent any number of mysterious arguments. Good function documentation is key to indicate the appropriate usage.

If you want to unpack the arguments passed in through an ellipsis, you can use the `list` function to convert those arguments into a list. Here's an example:

```
unpackme <- function(...){
    x <- list(...)
    cat("Here is ... in its entirety as a list:\n")
    print(x)
    cat("\nThe names of ... are:",names(x),"\n\n")
    cat("\nThe classes of ... are:",sapply(x,class))
}
```

This dummy function simply takes an ellipsis and converts it to a list with `x <- list(...)`. This subsequently allows the object `x` to be treated the same way as any other list. In this case, you can summarize the object by providing its `names` and `class` attributes. Here's a sample run:

```
R> unpackme(aa=matrix(1:4,2,2),bb=TRUE,cc=c("two","strings"),
            dd=factor(c(1,1,2,1)))
Here is ... in its entirety as a list:
$aa
     [,1] [,2]
```

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

$bb
[1] TRUE

$cc
[1] "two"     "strings"

$dd
[1] 1 1 2 1
Levels: 1 2


The names of ... are: aa bb cc dd


The classes of ... are: matrix logical character factor
```

Four tagged arguments, `aa`, `bb`, `cc`, and `dd`, are provided as the contents of the ellipsis, and they are explicitly identified within `unpackme` by using the simple `list(...)` operation. This construction can be useful for identifying or extracting specific arguments supplied through `...` in a given call.

Exercise 11.2

a. Accruing annual compound interest is a common financial benefit for investors. Given a principal investment amount P, an interest rate per annum i (expressed as a percentage), and a frequency of interest paid per year t, the final amount F after y years is given as follows:

$$F = P\left(1 + \frac{i}{100t}\right)^{ty}$$

Write a function that can compute F as per the following notes:

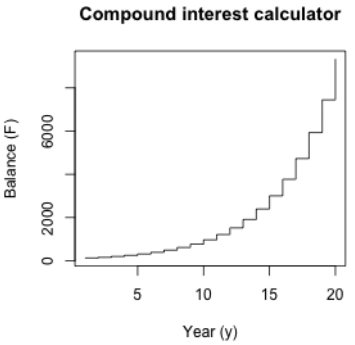
- Arguments must be present for P, i, t, and y. The argument for t should have a default value of 12.
- Another argument giving a logical value that determines whether to `plot` the amount F at each integer time should be included. For example, if `plotit=TRUE` (the default) and y is 5 years, the plot should show the amount F at $y = 1, 2, 3, 4, 5$.
- If this function is plotted, the plot should always be a step-plot, so `plot` should always be called with `type="s"`.

- If `plotit=FALSE`, the final amount F should be returned as a numeric vector corresponding to the same integer times, as shown earlier.
- An ellipsis should also be included to control other details of plotting, if it takes place.

Now, using your function, do the following:

i. Work out the final amount after a 10-year investment of a principal of $5000, at an interest rate of 4.4 percent per annum compounded monthly.

ii. Re-create the following step-plot, which shows the result of $100 invested at 22.9 percent per annum, compounded monthly, for 20 years:



iii. Perform another calculation based on the same parameters as in (ii), but this time, assume the interest is compounded annually. Return and store the results as a numeric vector. Then, use `lines` to add a second step-line, corresponding to this annually accrued amount, to the plot created previously. Use a different color or line type and make use of the `legend` function so the two lines can be differentiated.

b. A *quadratic equation* in the variable x is often expressed in the following form:

$$k_1 x^2 + k_2 x + k_3 = 0$$

Here, k_1, k_2, and k_3 are constants. Given values for these constants, you can attempt to find up to two *real roots*—values of x that satisfy the equation. Write a function that takes k_1, k_2, and k_3 as arguments and finds and returns any solutions (as a numeric vector) in such a situation. This is achieved as follows:

- Evaluate $k_2^2 - 4k_1k_3$. If this is negative, there are no solutions, and an appropriate message should be printed to the console.
- If $k_2^2 - 4k_1k_3$ is zero, then there is one solution, computed by $-k_2/2k_1$.

- If $k_2^2 - 4k_1k_3$ is positive, then there are two solutions, given by $(-k_2 - (k_2^2 - 4k_1k_3)^{0.5})/2k_1$ and $(-k_2 + (k_2^2 - 4k_1k_3)^{0.5})/2k_1$.
- No default values are needed for the three arguments, but the function should check to see whether any are missing. If so, an appropriate character string message should be returned to the user, informing the user that the calculations are not possible.

Now, test your function.

i. Confirm the following:
   * $2x^2 - x - 5$ has roots 1.850781 and −1.350781.
   * $x^2 + x + 1$ has no real roots.
ii. Attempt to find solutions to the following quadratic equations:
   * $1.3x^2 - 8x - 3.13$
   * $2.25x^2 - 3x + 1$
   * $1.4x^2 - 2.2x - 5.1$
   * $-5x^2 + 10.11x - 9.9$
iii. Test your programmed response in the function if one of the arguments is missing.

11.3 Specialized Functions

In this section, you'll look at three kinds of specialized user-defined R functions. First, you'll look at helper functions, which are designed to be called multiple times by another function (and they can even be defined inside the body of a parent function). Next, you'll look at disposable functions, which can be directly defined as an argument to another function call. Finally, you'll look at recursive functions, which call themselves.

11.3.1 Internally and Externally Defined Helper Functions

It is common for R functions to call other functions from within their body code. A *helper function* is a general term used to describe functions written and used specifically to facilitate the computations carried out by another function. Helper functions are a good way to improve the readability of complicated functions.

A helper function can be either defined internally (within another function definition) or externally (within the global environment). In this section, you'll see an example of each.

Building on the `multiples2` function from Section 11.2.2, here's a new version that splits the functionality over two separate functions:

```
multiples_helper_ext <- function(x,matrix.flags,mat){
    indexes <- which(matrix.flags)
    counter <- 0
```

```
    result <- list()
    for(i in indexes){
        temp <- x[[i]]
        if(ncol(temp)==nrow(mat)){
            counter <- counter+1
            result[[counter]] <- temp%*%mat
        }
    }
    return(list(result,counter))
}

multiples4 <- function(x,mat,str1="no valid matrices",str2=str1){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
    }

    helper.call <- multiples_helper_ext(x,matrix.flags,mat)
    result <- helper.call[[1]]
    counter <- helper.call[[2]]

    if(counter==0){
        return(str2)
    } else {
        return(result)
    }
}
```

If you import and execute this code on the sample lists from earlier, it behaves the same way as the previous version. All you've done here is moved the matrix-checking loop to an external function. The `multiples4` function now calls a helper function named `multiples_helper_ext`. Once the code in `multiples4` makes sure that there are in fact matrices in the list `x` to be checked, it calls `multiples_helper_ext` to perform the required loop.

In the previous example, the helper function is defined externally. That is, it exists in the global environment for any other function to call, making it easier to reuse. But if the helper function is intended to be used for only one particular function, it makes more sense to define the helper function internally, within the lexical environment of the function that calls it. The fifth version of the matrix multiplication function does just that, shifting the definition to within the body code.

```
multiples5 <- function(x,mat,str1="no valid matrices",str2=str1){
    matrix.flags <- sapply(x,FUN=is.matrix)

    if(!any(matrix.flags)){
        return(str1)
```

```
    }

    multiples_helper_int <- function(x,matrix.flags,mat){
        indexes <- which(matrix.flags)
        counter <- 0
        result <- list()
        for(i in indexes){
            temp <- x[[i]]
            if(ncol(temp)==nrow(mat)){
                counter <- counter+1
                result[[counter]] <- temp%*%mat
            }
        }
        return(list(result,counter))
    }

    helper.call <- multiples_helper_int(x,matrix.flags,mat)
    result <- helper.call[[1]]
    counter <- helper.call[[2]]

    if(counter==0){
        return(str2)
    } else {
        return(result)
    }
}
```

Now the helper function `multiples_helper_int` is defined within `multiples5`. That means it's visible only within the lexical environment as opposed to residing in the global environment like `multiples_helper_ext`. It makes sense to internally define a helper function when (a) it's used only by a single parent function, and (b) it's called multiple times within the parent function. (Of course, `multiples5` satisfies only (a), and it's provided here just for the sake of illustration.)

11.3.2 Disposable Functions

Often, you may need a function that performs a simple, one-line task. For example, when you use `apply`, you'll typically just want to pass in a short, simple function as an argument. That's where *disposable* (or *anonymous*) functions come in—they allow you to define a function intended for use in a single instance without explicitly creating a new object in your global environment.

Say you have a numeric matrix whose columns you want to repeat twice and then sort.

```
R> foo <- matrix(c(2,3,3,4,2,4,7,3,3,6,7,2),3,4)
R> foo
```

```
     [,1] [,2] [,3] [,4]
[1,]    2    4    7    6
[2,]    3    2    3    7
[3,]    3    4    3    2
```

This is a perfect task for `apply`, which can apply a function to each column of the matrix. This function simply has to take in a vector, repeat it, and sort the result. Rather than define that short function separately, you can define a disposable function right in the argument of `apply`.

```
R> apply(foo,MARGIN=2,FUN=function(x){sort(rep(x,2))})
     [,1] [,2] [,3] [,4]
[1,]    2    2    3    2
[2,]    2    2    3    2
[3,]    3    4    3    6
[4,]    3    4    3    6
[5,]    3    4    7    7
[6,]    3    4    7    7
```

The function is defined in the standard format directly in the call to `apply`. This function is defined, called, and then immediately forgotten once `apply` is complete. It is disposable in the sense that it exists only for the one instance where it is actually used.

Using the `function` command this way is a shortcut more than anything else; plus, it avoids the unnecessary creation and storage of a function object in the global environment.

11.3.3 Recursive Functions

Recursion is when a function calls itself. This technique isn't commonly used in statistical analyses, but it pays to be aware of it. This section will briefly illustrate what it means for a function to call itself.

Suppose you want to write a function that takes a single positive integer argument `n` and returns the corresponding *n*th term of the Fibonacci sequence (where $n = 1$ and $n = 2$ correspond to the initial two terms 1 and 1, respectively). Earlier you built up the Fibonacci sequence in an *iterative* fashion by using a loop. To write a recursive function, instead of using a loop to repeat an operation, the function calls itself multiple times. Consider the following function:

```
myfibrec <- function(n){
    if(n==1||n==2){
        return(1)
    } else {
        return(myfibrec(n-1)+myfibrec(n-2))
    }
}
```

The recursive `myfibrec` checks a single `if` statement that defines a *stopping condition*. If either `1` or `2` is supplied to the function (requesting the first or second Fibonacci number), then `myfibrec` directly returns `1`. Otherwise, the function returns the sum of `myfibrec(n-1)` and `myfibrec(n-2)`. That means if you call `myfibrec` with `n` greater than `2`, the function generates two more calls to `myfibrec`, using `n-1` and `n-2`. The recursion continues until it reaches a call for the `1`st and `2`nd terms, triggering the stopping condition, which simply returns `1`. Here's a sample call that retrieves the fifth Fibonacci number:

```
R> myfibrec(5)
[1] 5
```

Figure 11-3 shows the structure of this recursive call.

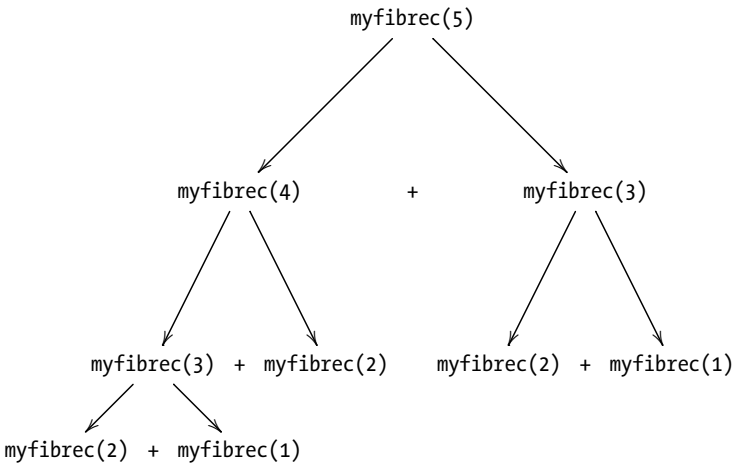


Figure 11-3: A visualization of the recursive calls made to `myfibrec` *with* `n=5`

Note that an accessible stopping rule is critical to any recursive function. Without one, recursion will continue indefinitely. For example, the current definition of `myfibrec` works as long as the user supplies a positive integer for the argument `n`. But if `n` is negative, the stopping rule condition will never be satisfied, and the function will recur indefinitely (though R has some automated safeguards to help prevent this).

Recursion is a powerful approach, especially when you don't know ahead of time how many times a function needs be called to complete a task. For many sort and search algorithms, recursion provides the speediest and most efficient solution. But in simpler cases, such as the Fibonacci example here, the recursive approach often requires more computational expense than an iterative looping approach. For beginners, I recommended sticking with explicit loops unless recursion is strictly required.

Exercise 11.3

a. Given a list whose members are character string vectors of varying lengths, use a disposable function with `lapply` to paste an exclamation mark onto the end of each element of each member, with an empty string as the separation character (note that the default behavior of `paste` when applied to character vectors is to perform the concatenation on each element). Execute your line of code on the list given by the following:

```
foo <- list("a",c("b","c","d","e"),"f",c("g","h","i"))
```

b. Write a recursive version of a function implementing the non-negative integer factorial operator (see Exercise 10.4 in Section 10.2.2 for details of the factorial operator). The stopping rule should return the value `1` if the supplied integer is `0`. Confirm that your function produces the same results as earlier.
   i. 5 factorial is 120.
   ii. 120 factorial is 479,001,600.
   iii. 0 factorial is 1.

c. For this problem, I'll introduce the geometric mean. The *geometric mean* is a particular measure of centrality, different from the more common arithmetic mean. Given n observations denoted $x_1, x_2, \ldots, x_n$, their geometric mean $\bar{g}$ is computed as follows:

$$\bar{g} = (x_1 \times x_2 \times \ldots \times x_n)^{1/n} = \left(\prod_{i=1}^{n} x_i \right)^{1/n}$$

For example, to find the geometric mean of the data 4.3, 2.1, 2.2, 3.1, calculate the following:

$$\bar{g} = (4.3 \times 2.1 \times 2.2 \times 3.1)^{1/4} = 61.5846^{0.25} = 2.8$$

(This is rounded to 1 d.p.)

Write a function named `geolist` that can search through a specified argument list and compute the geometric means of each member per the following guidelines:

- Your function must define and use an internal helper function that returns the geometric mean of a vector argument.
- Assume the list can only have numeric vectors or numeric matrices as its members. Your function should contain an appropriate loop to inspect each member in turn.
- If the member is a vector, compute the geometric mean of that vector, overwriting the member with the result, which should be a single number.
- If the member is a matrix, use an implicit loop to compute the geometric mean *of each row* of the matrix, overwriting the member with the results.

– The final list should be returned to the user.

Now, as a quick test, check whether your function matches the following two calls:

i.

```
R> foo <- list(1:3,matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),4,2),
               matrix(c(3.3,3.2,2.8,2.1,4.6,4.5,3.1,9.4),2,4))
R> geolist(foo)
[[1]]
[1] 1.817121

[[2]]
[1] 3.896152 3.794733 2.946184 4.442972

[[3]]
[1] 3.388035 4.106080
```

ii.

```
R> bar <- list(1:9,matrix(1:9,1,9),matrix(1:9,9,1),matrix(1:9,3,3))
R> geolist(bar)
[[1]]
[1] 4.147166

[[2]]
[1] 4.147166

[[3]]
[1] 1 2 3 4 5 6 7 8 9

[[4]]
[1] 3.036589 4.308869 5.451362
```

Important Code in This Chapter

Function/operator	Brief description	First occurrence
`function`	Function creation	Section 11.1.1, p. 218
`return`	Function return objects	Section 11.1.1, p. 220
`missing`	Argument check	Section 11.2.3, p. 229
`...`	Ellipsis (as argument)	Section 11.2.4, p. 230

12

EXCEPTIONS, TIMINGS, AND VISIBILITY

Now that you've seen how to write your own functions in R, let's examine some common function augmentations and behaviors. In this chapter, you'll learn how to make your functions throw an error or warning when they receive unexpected input. You'll also see some simple ways to measure completion time and check progress for computationally expensive functions. Finally, you'll see how R masks functions when two have the same name but reside in different packages.

12.1 Exception Handling

When there's an unexpected problem during execution of a function, R will notify you with either a *warning* or an *error*. In this section, I'll demonstrate how to build these constructs into your own functions where appropriate.

I'll also show how to *try* a calculation to check whether it's possible (that is, to see whether it'll even work) before throwing an error.

12.1.1 Formal Notifications: Errors and Warnings

In Chapter 11, you made your functions print a string (for example, `"no valid matrices"`) when they couldn't perform certain operations. Warnings and errors are a more formal mechanism to send these types of messages and handle subsequent operations. An error forces the function to immediately terminate at the point it occurs. A warning is less severe. It indicates that the function is being run in an atypical way but tries to work around the issue and continue executing. In R, you can issue warnings with the `warning` command, and you can throw errors with the `stop` command. The following two functions show an example of each:

```
warn_test <- function(x){
    if(x<=0){
        warning("'x' is less than or equal to 0 but setting it to 1 and
                continuing")
        x <- 1
    }
    return(5/x)
}

error_test <- function(x){
    if(x<=0){
        stop("'x' is less than or equal to 0... TERMINATE")
    }
    return(5/x)
}
```

Both `warn_test` and `error_test` divide 5 by the argument `x`. They also both expect `x` to be positive. In `warn_test`, if `x` is nonpositive, the function issues a warning, and `x` is overwritten to be `1`. In `error_test`, on the other hand, if `x` is nonpositive, the function throws an error and terminates immediately. The two commands `warning` and `stop` are used with a character string argument, which becomes the message printed to the console.

You can see these notifications by importing and calling the functions as follows:

```
R> warn_test(0)
[1] 5
Warning message:
In warn_test(0) :
  'x' is less than or equal to 0 but setting it to 1 and continuing
R> error_test(0)
Error in error_test(0) : 'x' is less than or equal to 0... TERMINATE
```

Notice that `warn_test` has continued to execute and returned the value `5`—the result of `5/1` after setting `x` to `1`. The call to `error_test` did not return anything because R exited the function at the `stop` command.

Warnings are useful when there is a logical way for a function to try to save itself even when it doesn't get the input it expects. For example, in Section 10.1.3, R issued a warning when you supplied a logical vector of elements to the `if` statement. Remember that the `if` statement expects a single logical value, but rather than quit when a logical vector is provided instead, it continues execution using just the first logical entry in the supplied vector. That said, sometimes it's more appropriate to actually throw an error and stop execution altogether.

Let's go back to `myfibrec` from Section 11.3.3. This function expects a positive integer (the position of the Fibonacci number it should return). Suppose you assume that if the user supplies a negative integer, the user actually means the positive version of that term. You can add a warning to handle this situation. Meanwhile, if the user enters 0, which doesn't correspond to any position in the Fibonacci series, the code will throw an error. Consider these modifications:

```
myfibrec2 <- function(n){
    if(n<0){
        warning("Assuming you meant 'n' to be positive -- doing that instead")
        n <- n*-1
    } else if(n==0){
        stop("'n' is uninterpretable at 0")
    }

    if(n==1||n==2){
        return(1)
    } else {
        return(myfibrec2(n-1)+myfibrec2(n-2))
    }
}
```

In `myfibrec2`, you now check whether `n` is negative or zero. If it's negative, the function issues a warning and continues executing after swapping the argument's sign. If `n` is zero, an error halts execution with a corresponding message. Here you can see the responses for a few different arguments:

```
R> myfibrec2(6)
[1] 8
R> myfibrec2(-3)
[1] 2
Warning message:
In myfibrec2(-3) :
  Assuming you meant 'n' to be positive -- doing that instead
R> myfibrec2(0)
Error in myfibrec2(0) : 'n' is uninterpretable at 0
```

Note that the call to `myfibrec2(-3)` has returned the third Fibonacci number.

Broadly speaking, both errors and warnings signal that something has gone wrong. If you're using a certain function or running chunks of code and you encounter these kinds of messages, you should look carefully at what has been run and what may have occurred to spark them.

NOTE *Identifying and repairing erroneous code is referred to as* debugging, *for which there are various strategies. See* The Art of Debugging *by Matloff & Salzman (2008), for example. One of the most basic strategies simply involves including* `print` *or* `cat` *commands to inspect various quantities as they are calculated during live execution. R does have some more sophisticated debugging tools; if you're interested, check out the excellent discussion on them provided in Chapter 13 of Matloff (2011). Regardless, as you gain more experience in R, understanding error messages or locating potential problems in code before they arise becomes easier and easier, a benefit you get partly because of R's interpretative style.*

12.1.2 Trying Expressions and Catching Errors

When a function terminates from an error, it also terminates any parent functions. For example, if function A calls function B and function B halts because of an error, this halts execution of A at the same point. To avoid this severe consequence, you can use a `try` statement to attempt a function call and check whether it produces an error.

In R, you can use the `try` command to catch an error and specify alternative operations, thereby providing the opportunity to avoid the instant termination of all relevant processes should an error be thrown.

For example, if you call the `myfibrec2` function from earlier and pass it 0, the function throws an error and terminates. But watch what happens when you pass that function call as the first argument to `try`:

```
R> attempt1 <- try(myfibrec2(0),silent=TRUE)
```

What's happened to the error? In fact, the error has still occurred, but `try` has suppressed the printing of an error message to the console (that's because you set `silent=TRUE`). Furthermore, error information is now stored in `attempt1` as an object of class `"try-error"`.

```
R> attempt1
[1] "Error in myfibrec2(0) : 'n' is uninterpretable at 0\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in myfibrec2(0): 'n' is uninterpretable at 0>
```

Catching an error this way can be a huge benefit, especially when a function produces the error in the body code of another function. Using `try`, you can handle the error without terminating that parent function.

Meanwhile, if you pass a function to try and it doesn't throw an error, then try has no effect, and you simply get the normal return value.

```
R> attempt2 <- try(myfibrec2(6),silent=TRUE)
R> attempt2
[1] 8
```

Here, you executed myfibrec2 with a valid argument, n=6. Since this call doesn't result in an error, the result passed to attempt2 is the normal return value from myfibrec2, in this case 8.

Let's see a more complete example of how you could use try in a larger function. The following myfibvector function takes a vector of indexes as the argument nvec and provides the corresponding terms from the Fibonacci sequence:

```
myfibvector <- function(nvec){
    nterms <- length(nvec)
    result <- rep(0,nterms)
    for(i in 1:nterms){
        result[i] <- myfibrec2(nvec[i])
    }
    return(result)
}
```

This function uses a for loop to work through nvec element by element, computing the corresponding Fibonacci number with the earlier function, myfibrec2. As long as all the values in nvec are nonzero, myfibvector works just fine. For example, the following call obtains the first, the second, the tenth, and the eighth Fibonacci number:

```
R> foo <- myfibvector(nvec=c(1,2,10,8))
R> foo
[1]  1  1 55 21
```

Suppose, however, there's a mistake and one of the entries in nvec ends up being zero.

```
R> bar <- myfibvector(nvec=c(3,2,7,0,9,13))
Error in myfibrec2(nvec[i]) : 'n' is uninterpretable at 0
```

The internal call to myfibrec2 has thrown an error when it's called on n=0, and this has terminated execution of myfibvector. Nothing is returned, and the entire call has failed.

You can prevent this outright failure by using try to check each call to myfibrec2 and have it catch any errors. The following function, myfibvectorTRY, does just that.

```
myfibvectorTRY <- function(nvec){
    nterms <- length(nvec)
    result <- rep(0,nterms)
    for(i in 1:nterms){
        attempt <- try(myfibrec2(nvec[i]),silent=T)
        if(class(attempt)=="try-error"){
            result[i] <- NA
        } else {
            result[i] <- attempt
        }
    }
    return(result)
}
```

Here, within the `for` loop, you use `attempt` to store the result of trying each call to `myfibrec2`. Then, you inspect `attempt`. If this object's class is `"try-error"`, that means `myfibrec2` produced an error, and you fill the corresponding slot in the `result` vector with `NA`. Otherwise, `attempt` will represent a valid return value from `myfibrec2`, so you place it in the corresponding slot of the `result` vector. Now if you import and call `myfibvectorTRY` on the same `nvec`, you see a complete set of results.

```
R> baz <- myfibvectorTRY(nvec=c(3,2,7,0,9,13))
R> baz
[1]   2   1  13  NA  34 233
```

The error that would have otherwise terminated everything was silently caught, and the alternative response in this situation, `NA`, was inserted into the `result` vector.

NOTE *The `try` command is a simplification of R's more complex `tryCatch` function, which is beyond the scope of this book, but it provides more precise control over how you test and execute chunks of code.*

In all the `try` calls I've shown so far, I've set the `silent` argument to `TRUE`, which stops any error messages from being printed. If you leave `silent` set to `FALSE` (the default value), the error message will be printed, but the error will still be caught without terminating execution.

Note that setting `silent=TRUE` only suppresses error messages, not warnings. Observe the following:

```
R> attempt3 <- try(myfibrec2(-3),silent=TRUE)
Warning message:
In myfibrec2(-3) :
  Assuming you meant 'n' to be positive -- doing that instead
R> attempt3
[1] 2
```

Although `silent` was `TRUE`, the warning (for negative values of `n` in this example) is still issued and printed. Warnings are treated separately from errors in this type of situation, as they should be—they can highlight other unforeseen issues with your code during execution. If you are absolutely sure you don't want to see any warnings, you can use `suppressWarnings`.

```
R> attempt4 <- suppressWarnings(myfibrec2(-3))
R> attempt4
[1] 2
```

The `suppressWarnings` function should be used only if you are certain that every warning in a given call can be safely ignored and you want to keep the output tidy.

Exercise 12.1

a. In Exercise 11.3 (b), your task was to write a recursive R function to compute integer factorials, given some supplied non-negative integer `x`. Now, modify your function so that it throws an error (with an appropriate message) if `x` is negative. Test your new function responses by using the following:
   i. `x` as `5`
   ii. `x` as `8`
   iii. `x` as `-8`

b. The idea of *matrix inversion*, briefly discussed in Section 3.3.6, is possible only for certain square matrices (those with an equal number of columns as rows). The `solve` function can compute these inversions. Here's an example:

```
R> solve(matrix(1:4,2,2))
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Note that `solve` throws an error if the supplied matrix cannot be inverted. With this in mind, write an R function that attempts to invert each matrix in a list, according to the following guidelines:

- The function should take four arguments.
  * The list `x` whose members are to be tested for matrix inversion
  * A value `noninv` to fill in results where a given matrix member of `x` cannot be inverted, defaulting to `NA`
  * A character string `nonmat` to be the result if a given member of `x` is not a matrix, defaulting to `"not a matrix"`

    - A logical value `silent`, defaulting to `TRUE`, to be passed to `try` in the body code
- The function should first check whether `x` is in fact a list. If not, it should throw an error with an appropriate message.
- Then, the function should ensure that `x` has at least one member. If not, it should throw an error with an appropriate message.
- Next, the function should check whether `nonmat` is a character string. If not, it should try to coerce it to a character string using an appropriate "as-dot" function (see Section 6.2.4), and it should issue an appropriate warning.
- After these checks, a loop should search each member `i` of the list `x`.
    - If member `i` is a matrix, attempt to invert it with `try`. If it's invertible without error, overwrite member `i` of `x` with the result. If an error is caught, then member `i` of `x` should be overwritten with the value of `noninv`.
    - If member `i` is not a matrix, then member `i` of `x` should be overwritten with the value of `nonmat`.
- Finally, the modified list `x` should be returned.

Now, test your function using the following argument values to make sure it responds as expected:

i. `x` as

```
list(1:4,matrix(1:4,1,4),matrix(1:4,4,1),matrix(1:4,2,2))
```

and all other arguments at default.

ii. `x` as in (i), `noninv` as `Inf`, `nonmat` as `666`, `silent` at default.

iii. All as in (ii), but now with `silent=FALSE`.

iv. `x` as

```
list(diag(9),matrix(c(0.2,0.4,0.2,0.1,0.1,0.2),3,3),
     rbind(c(5,5,1,2),c(2,2,1,8),c(6,1,5,5),c(1,0,2,0)),
     matrix(1:6,2,3),cbind(c(3,5),c(6,5)),as.vector(diag(2)))
```

and `noninv` as `"unsuitable matrix"`; all other values at default.

Finally, test the error messages by attempting calls to your function with the following:

v. `x` as `"hello"`

vi. `x` as `list()`

12.2 Progress and Timing

R is often used for lengthy numerical exercises, such as simulation or random variate generation. For these complex, time-consuming operations, it's often useful to keep track of progress or see how long a certain task took to complete. For example, you may want to compare the speed of two different programming approaches to a given problem. In this section, you'll look at ways to time code execution and show its progress.

12.2.1 Textual Progress Bars: Are We There Yet?

A *progress bar* shows how far along R is as it executes a set of operations. To show how this works, you need to run code that takes a while to execute, which you'll do by making R *sleep*. The `Sys.sleep` command makes R pause for a specified amount of time, in seconds, before continuing.

```
R> Sys.sleep(3)
```

If you run this code, R will pause for three seconds before you can continue using the console. Sleeping will be used in this section as a surrogate for the delay caused by computationally expensive operations, which is where progress bars are most useful.

To use `Sys.sleep` in a more common fashion, consider the following:

```
sleep_test <- function(n){
    result <- 0
    for(i in 1:n){
        result <- result + 1
        Sys.sleep(0.5)
    }
    return(result)
}
```

The `sleep_test` function is basic—it takes a positive integer `n` and adds `1` to the `result` value for `n` iterations. At each iteration, you also tell the loop to sleep for a half second. Because of that sleep command, executing the following code takes about four seconds to return a result:

```
R> sleep_test(8)
[1] 8
```

Now, say you want to track the progress of this type of function as it executes. You can implement a textual progress bar with three steps: initialize the bar object with `txtProgressBar`, update the bar with `setTxtProgressBar`, and terminate the bar with `close`. The next function, `prog_test`, modifies `sleep_test` to include the following three commands.

```
prog_test <- function(n){
    result <- 0
    progbar <- txtProgressBar(min=0,max=n,style=1,char="=")
    for(i in 1:n){
        result <- result + 1
        Sys.sleep(0.5)
        setTxtProgressBar(progbar,value=i)
    }
    close(progbar)
    return(result)
}
```

Before the `for` loop, you create an object named `progbar` by calling `txtProgressBar` with four arguments. The `min` and `max` arguments are numeric values that define the limits of the bar. In this case, you set `max=n`, which matches the number of iterations of the impending `for` loop. The `style` argument (integer, either `1`, `2`, or `3`) and the `char` argument (character string, usually a single character) govern the appearance of the bar.

Once this object is created, you have to instruct the bar to actually progress during execution with a call to `setTxtProgressBar`. You pass in the bar object to update (`progbar`) and the `value` it should update to (in this case, `i`). Once complete (after exiting the loop), the progress bar must be terminated with a call to `close`, passing in the bar object of interest. Import and execute `prog_test`, and you'll see a line of `"="` drawn in steps as the loop completes.

```
R> prog_test(8)
==================================================================
[1] 8
```

The width of the bar is, by default, determined by the width of the R console pane upon execution of the `txtProgressBar` command. You can also customize the bar a bit by changing the `style` and `char` arguments. Choosing `style=3`, for example, shows the bar as well as a "percent completed" counter. Some packages offer more elaborate options too, such as pop-up widgets, but the textual version is the simplest and most universally compatible version across different systems.

12.2.2 Measuring Completion Time: How Long Did It Take?

If you want to know how long a computation takes to complete, you can use the `Sys.time` command. This command outputs an object that details current date and time information based on your system.

```
R> Sys.time()
[1] "2014-01-24 11:44:39 NZDT"
```

You can store objects like these before and after some code and then compare them to see how much time has passed. Write this in the editor:

```
t1 <- Sys.time()
Sys.sleep(3)
t2 <- Sys.time()
t2-t1
```

Now highlight all four lines and execute them in the console.

```
R> t1 <- Sys.time()
R> Sys.sleep(3)
R> t2 <- Sys.time()
R> t2-t1
Time difference of 3.012889 secs
```

By executing this entire code block together, you get an easy measure of the total completion time in a nicely formatted string printed to the console. Note that there's a small time cost for interpreting and invoking any commands, in addition to the three seconds you tell R to sleep. This time will vary between computers.

If you need more detailed timing reports, there are more sophisticated tools. You can replace the `Sys.time()` calls with the `proc.time()` calls in the previous example to receive not just the total elapsed "wall clock" time but also computer-related CPU timings (see the definitions in the help file `?proc.time`). To time a single expression, you can also use the `system.time` function (which uses the same detail of output as `proc.time`). Furthermore, *benchmarking* tools (formal/systematic comparisons of different approaches) for timing your code are also available; see, for example, the `rbenchmark` package (Kusnierczyk 2012). However, for everyday use, the time-object differencing approach used here is easy to interpret and provides a good indication of the computational expense.

Exercise 12.2

a. Modify `prog_test` to include an ellipsis in its argument list, intended to take values for the additional arguments in `txtProgressBar`; name the new function `prog_test_fancy`. Time how long it takes a call to `prog_test_fancy` from Section 12.2.1 to execute. Set `50` as `n`, instruct the progress bar (through the ellipsis) to use `style=3`, and set the bar character to be `"r"`.

b. In Section 12.1.2, you defined a function named `myfibvectorTRY` (which itself calls `myfibrec2` from Section 12.1.1) to return multiple terms from the Fibonacci sequence based on a supplied "term vector" `nvec`. Write a new version of `myfibvectorTRY` that

includes a progress bar of `style=3` and a character of your choosing that increments at each pass of the internal `for` loop. Then, do the following:

i. Use your new function to reproduce the results from the text where `nvec=c(3,2,7,0,9,13)`.

ii. Time how long it takes to use your new function to return the first 35 terms of the Fibonacci sequence. What do you notice, and what does this say about your recursive Fibonacci functions?

c. Remain with the Fibonacci sequence. Write a stand-alone `for` loop that can compute, and store in a vector, the same first 35 terms as in (b)(ii). Time it. Which approach would you prefer?

12.3 Masking

With the plethora of built-in and contributed data and functionality available for R, it is virtually inevitable that at some point you will come across objects, usually functions, that share the same name in distinctly different loaded packages.

So, what happens in those instances? For example, say you define a function with the same name as a function in an R package that you have already loaded. R responds by *masking* one of the objects—this protects objects from overwriting or blocking one another. In this section, you'll look at the two most common masking situations in R.

12.3.1 Function and Object Distinction

When two functions or objects in different environments have the same name, the object that comes earlier in the search path will mask the other one. That is, when the object is sought, R will use the object or function it finds first, and you'll need extra syntax to access the other, masked version. Remember, you can see the current search path by executing `search()`.

```
R> search()
 [1] ".GlobalEnv"        "tools:RGUI"        "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

To see a simple example of masking, you'll define a function with the same name as a function in the base package: `sum`. Here's how `sum` works normally, adding up all the elements in the vector `foo`:

```
R> foo <- c(4,1.5,3)
R> sum(foo)
[1] 8.5
```

Now, suppose you were to write the following function:

```
sum <- function(x){
    result <- 0
    for(i in 1:length(x)){
        result <- result + x[i]^2
    }
    return(result)
}
```

This version of `sum` takes in a vector `x` and uses a `for` loop to square each element before summing them and returning the result. This can be imported into the R console without any problem, but clearly, it doesn't offer the same functionality as the (original) built-in version of `sum`. Now, after importing the previous code, if you make a call to `sum`, your version is used.

```
R> sum(foo)
[1] 27.25
```

This happens because the user-defined function is stored in the global environment (`.GlobalEnv`), which always comes first in the search path. R's built-in function is part of the `base` package, which comes at the end of the search path. In this case, your user-defined function is masking the original version.

Now, if you want R to run the `base` version of `sum`, you have to include the name of its package in the call, with a double colon.

```
R> base::sum(foo)
[1] 8.5
```

This tells R to use the version in `base`, even though there's another version of the function in the global environment.

To avoid any confusion, let's remove the `sum` function from the global environment.

```
R> rm(sum)
```

When you load a package, R will notify you if any objects in the package clash with other objects that are accessible in the present session. To illustrate this, I'll make use of two contributed packages: the `car` package (you saw this earlier in Exercise 8.1 (b)) and the `spatstat` package (you'll use this

in Part V). After ensuring these two packages are installed, when I load them in the following order, I see this message:

```
R> library("spatstat")

spatstat 1.40-0       (nickname: 'Do The Maths')
For an introduction to spatstat, type 'beginner'
R> library("car")

Attaching package: 'car'

The following object is masked from 'package:spatstat':

    ellipse
```

This indicates that the two packages each have an object with the same name—`ellipse`. R has automatically notified you that this object is being masked. Note that the functionality of both `car` and `spatstat` remains completely available; it's just that the `ellipse` objects require some distinction should they be needed. Using `ellipse` at the prompt will access `car`'s object since that package was loaded more recently. To use `spatstat`'s version, you must type `spatstat::ellipse`. These rules also apply to accessing the respective help files.

A similar notification occurs when you load a package with an object that's masked by a global environment object (a global environment object will always take precedence over a package object). To see an example, you can load the `MASS` package (Venables & Ripley 2002), which is included with R but isn't automatically loaded. Continuing in the current R session, create the following object:

```
R> cats <- "meow"
```

Now, suppose you need to load `MASS`.

```
R> library("MASS")

Attaching package: 'MASS'

The following object is masked _by_ '.GlobalEnv':

    cats

The following object is masked from 'package:spatstat':

    area
```

Upon loading the package, you're informed that the cats object you've just created is masking an object of the same name in MASS. (As you can see with ?MASS::cats, this object is a data frame with weight measurements of household felines.) Furthermore, it appears MASS also shares an object name with spatstat—area. The same kind of "package masking" message as shown earlier is also displayed for that particular item.

You can unmount loaded packages from the search path. With the packages loaded in this discussion, my current search path looks like this:

```
R> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:car"
 [4] "package:spatstat"  "tools:RGUI"        "package:stats"
 [7] "package:graphics"  "package:grDevices" "package:utils"
[10] "package:datasets"  "package:methods"   "Autoloads"
[13] "package:base"
```

Now, suppose you don't need car anymore. You can remove it with the detach function.

```
R> detach("package:car")
R> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:spatstat"
 [4] "tools:RGUI"        "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "package:methods"   "Autoloads"         "package:base"
```

This removes the elected package from the path. Now, the functionality of car is no longer immediately available, and spatstat's ellipsis function is no longer masked.

NOTE *As contributed packages get updated by their maintainers, they may include new objects that spark new maskings or remove/rename objects that previously caused maskings (when compared with other contributed packages). The specific maskings illustrated here among car, spatstat, and MASS occur at the time of this writing with the versions available and may change in the future.*

12.3.2 Data Frame Variable Distinction

There is one other common situation in which you'll be explicitly notified of masking: when you add a data frame to the search path. Let's see how this works. Continuing in the current workspace, define the following data frame:

```
R> foo <- data.frame(surname=c("a","b","c","d"),
                     sex=c(0,1,1,0),height=c(170,168,181,180),
                     stringsAsFactors=F)
R> foo
  surname sex height
```

```
1       a   0    170
2       b   1    168
3       c   1    181
4       d   0    180
```

The data frame `foo` has three column variables: `person`, `sex`, and `height`. To access a column of this data frame, normally you need to use the `$` operator and type something like `foo$surname`. However, you can *attach* a data frame directly to your search path, which makes it easier to access a variable.

```
R> attach(foo)
R> search()
 [1] ".GlobalEnv"        "foo"               "package:MASS"
 [4] "package:spatstat"  "tools:RGUI"        "package:stats"
 [7] "package:graphics"  "package:grDevices" "package:utils"
[10] "package:datasets"  "package:methods"   "Autoloads"
[13] "package:base"
```

Now the `surname` variable is directly accessible.

```
R> surname
[1] "a" "b" "c" "d"
```

This saves you from having to type `foo$` every time you want to access a variable, which can be a handy shortcut if your analysis deals exclusively with one static, unchanging data frame. However, if you forget about your attached objects, they can cause problems later, especially if you continue to mount more objects onto the search path in the same session. For example, say you enter another data frame.

```
R> bar <- data.frame(surname=c("e","f","g","h"),
                     sex=c(1,0,1,0),weight=c(55,70,87,79),
                     stringsAsFactors=F)
R> bar
  surname sex weight
1       e   1     55
2       f   0     70
3       g   1     87
4       h   0     79
```

Then add it to the search path too.

```
R> attach(bar)
The following objects are masked from foo:

    sex, surname
```

The notification tells you that the bar object now precedes foo in the search path.

```
R> search()
 [1] ".GlobalEnv"        "bar"               "foo"
 [4] "package:MASS"      "package:spatstat"  "tools:RGUI"
 [7] "package:stats"     "package:graphics"  "package:grDevices"
[10] "package:utils"     "package:datasets"  "package:methods"
[13] "Autoloads"         "package:base"
```

As a result, any direct use of either sex or surname will now access bar's contents, not foo's. Meanwhile, the unmasked variable height from foo is still directly accessible.

```
R> height
[1] 170 168 181 180
```

This is a pretty simple example, but it highlights the potential for confusion when data frames, lists, or other objects are added to the search path. Mounting objects this way can quickly become difficult to track, especially for large data sets with many different variables. For this reason, it's best to avoid attaching objects this way as a general guideline.

Note that detach can be used to remove objects from the search path, in a similar way as you saw with packages. In this case, you can simply type the object name itself.

```
R> detach(foo)
R> search()
 [1] ".GlobalEnv"        "bar"               "package:MASS"
 [4] "package:spatstat"  "tools:RGUI"        "package:stats"
 [7] "package:graphics"  "package:grDevices" "package:utils"
[10] "package:datasets"  "package:methods"   "Autoloads"
[13] "package:base"
```

Important Code in This Chapter

Function/operator	Brief description	First occurrence
warning	Issue warning	Section 12.1.1, p. 244
stop	Throw error	Section 12.1.1, p. 244
try	Attempt error catch	Section 12.1.2, p. 246
Sys.sleep	Sleep (pause) execution	Section 12.2.1, p. 251
txtProgressBar	Initialize progress bar	Section 12.2.1, p. 251
setTxtProgressBar	Increment progress bar	Section 12.2.1, p. 251
close	Close progress bar	Section 12.2.1, p. 251
Sys.time	Get local system time	Section 12.2.2, p. 252
detach	Remove library/object from path	Section 12.3.1, p. 257
attach	Attach object to search path	Section 12.3.2, p. 258