2

NUMERICS, ARITHMETIC, ASSIGNMENT, AND VECTORS

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 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 enter 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 9.1.2.

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 numeric 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 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 instance, in this example, `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` again, 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 new values. 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

Vectors are so useful because they allow R to carry out operations on multiple elements simultaneously with speed and efficiency. 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, in increments of 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. 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

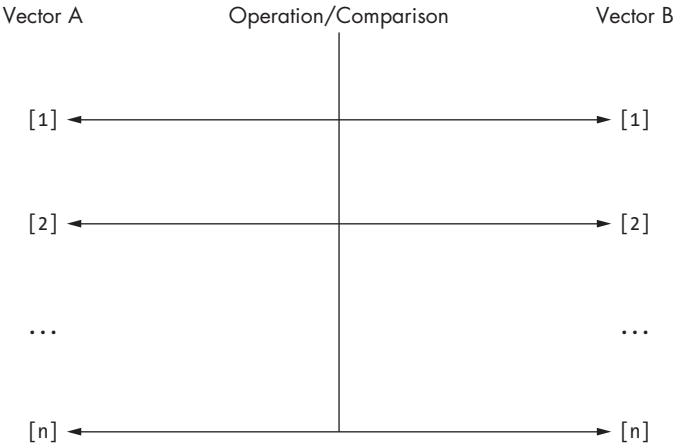

Figure 2-1: A conceptual diagram of the element-wise behavior of a comparison or operation carried out on two vectors of equal length in R. Note that the operation is performed by matching up the element positions.

and positive. You could explicitly multiply `foo` by `c(1,-1,1,-1,1,-1)`, but 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.

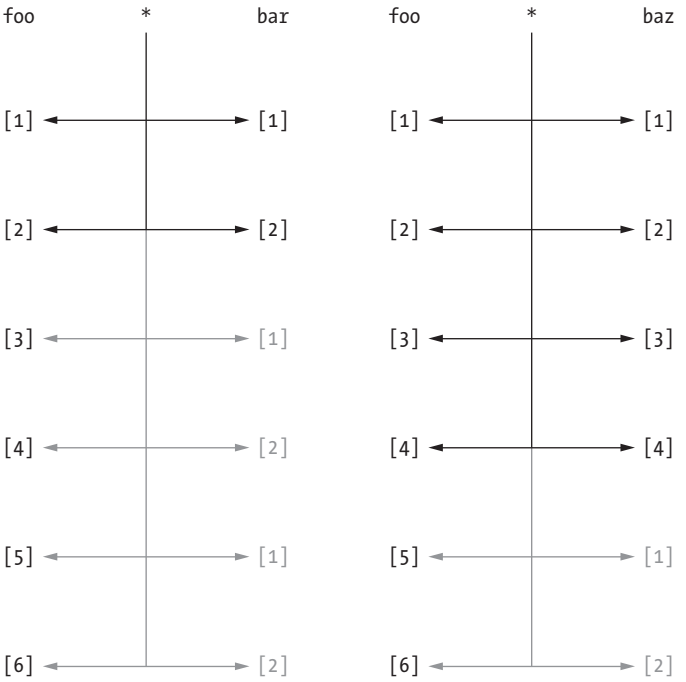

Figure 2-2: An element-wise operation on two vectors of differing lengths. Left: `foo` *multiplied by* `bar`*; lengths are evenly divisible. Right:* `foo` *multiplied by* `baz`*; lengths are not evenly divisible, and a warning is issued.*

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. 17
`sqrt`	Square root	Section 2.1.1, p. 18
`log`	Logarithm	Section 2.1.2, p. 19
`exp`	Exponential	Section 2.1.2, p. 19
`<-, =`	Object assignment	Section 2.2, p. 21
`c`	Vector creation	Section 2.3.1, p. 23
`:, seq`	Sequence creation	Section 2.3.2, p. 24
`rep`	Value/vector repetition	Section 2.3.2, p. 25
`sort`	Vector sorting	Section 2.3.2, p. 26
`length`	Determine vector length	Section 2.3.2, p. 27
`[ ]`	Vector subsetting/extraction	Section 2.3.3, p. 28
`sum`	Sum all vector elements	Section 2.3.4, p. 36
`prod`	Multiply all vector elements	Section 2.3.4, p. 36