Chapter 8

User-Written Functions

The previous chapter outlined functions that are built into the R language. But you will likely encounter applications in which an appropriate function is not available. R gives you the capability to write your own functions for such applications. This chapter gives the syntax and some examples of writing simple R functions. More sophisticated functions can be written after covering the programming topics in Chapters 22–24. When a user-written function is saved at the end of an R session, it can be used in subsequent R sessions. An important purpose of user-written R functions is to decompose complex programming tasks into smaller pieces, thus decreasing the chances of an error.

The syntax for defining a user-written function named `FunctionName` is

```
FunctionName = function(argument1, argument2, ...)  {  }
```

where the curly braces contain the R commands associated with the function. If the entire function can be written on a single line, the curly braces are not necessary; the R command is simply written after the argument list. Most functions, however, require multiple lines, so there is typically a left-hand curly brace on the first line of the function and a right-hand curly brace on the last line of the function. You choose the name of the function and the name of the arguments using the usual naming rules for objects, for example these names are case sensitive. Typing the name of the function (without the arguments) at the R command prompt allows you to see its source code. The arguments to the function are contained in parentheses and are separated by commas. You have the option of establishing *default* argument values, which are assumed if certain arguments are omitted in a call to the function. It is acceptable for a function to have no arguments. There are certain function names that should be avoided because they perform other roles in R, such as `c`, `q`, `if`, `while`, etc. As was the case with object names, it is helpful to choose function names that adequately and succinctly reflect the purpose of the function.

R's style of using the assignment operator (= or <-) to define a function is unlike most programming languages. One upside to this approach is that a user-written function is an object, just like a vector, matrix, or array. Typing `ls()` or `objects()` displays the names of the user-written functions along with all of the other objects.

The syntax for calling (invoking) a user-written function is exactly the same as calling a function that is built into R. To call a function named `FunctionName`, use

```
FunctionName(argument1, argument2, ...  )
```

The topics considered in this chapter are (a) elementary functions, (b) scoping, the rules that govern the way that R looks up the value of an object, and (c) variable number of arguments.

8.1 Elementary functions

As an elementary first example of a user-written function, consider a function that cubes its argument. We will name this function `cube`. There is currently no such function with that name in R, but, to be on the safe side, begin by typing `cube` to see if there is an existing object or function named `cube`.

```
> cube                          # check for a built-in function named cube
Error: object 'cube' not found
```

Seeing that the name `cube` is "available," we can begin writing the function.

```
> cube = function(x) x ^ 3      # write the function
```

This function is so short that the curly braces were not necessary. R silently accepts the new function. Next, type the name of the function to see that it has been successfully included.

```
> cube                          # display the code
function(x) x ^ 3
```

All looks good so far. The next step is to test the function.

```
> cube(2)                       # 2 ^ 3
[1] 8
```

Success! Now try a bigger argument.

```
> cube(10)                      # 10 ^ 3
[1] 1000
```

It seems as though `cube` will work for constants, although it would be prudent to test it with some negative values and some non-integers. Next, try a vector as an argument in `cube`.

```
> cube(1:10)                    # first ten perfect cubes
 [1]    1    8   27   64  125  216  343  512  729 1000
```

Our `cube` function works in exactly the same fashion as, for example, the `sqrt` function. It works in an element-wise fashion. Next, try placing a matrix into `cube`.

```
> cube(matrix(1:8, 2, 4))       # cube each element in a matrix
     [,1] [,2] [,3] [,4]
[1,]    1   27  125  343
[2,]    8   64  216  512
```

Each element of the matrix is cubed as desired. Notice in this case that reversing the order of the calls to `cube` and `matrix` in the R command results in the same 2×4 matrix.

```
> matrix(cube(1:8), 2, 4)       # matrix of the cubes
     [,1] [,2] [,3] [,4]
[1,]    1   27  125  343
[2,]    8   64  216  512
```

Finally, using an array as an argument in `cube` also works.