Part II

Programming, Monte Carlo, and Resampling

5

Programming Concepts

CONTENTS

5.1 Introduction

The code we have written so far has exclusively used functions provided in base *R* and in packages contributed by others. However, there are often situations where it can be advantageous to write our own functions. For example, we can imagine wanting to reuse some code, and rather than making copies of the code, we can create a function that performs these computations. We simply call our function when we need to carry out the computations. This approach is less error prone because we typically have less to type when we call the function than when we repeat several lines of code. It is also less error prone than copy-and-pasting lines of code. Additionally, we might want our function to handle variations in the code. To do this, we can design our function to accept arguments that control how the code is evaluated. Also, corrections and updates to our code can be made in one place (in the function), and these changes are available to all function calls. Moreover, a function typically encapsulates a task and if we give it a name that reflects this task, then our function call helps makes it clear to us and others what our code is doing. Furthermore, when we have a large task to perform, we can more easily understand our work, test our code, and track progress on the project if we divide the large task into smaller tasks that we encapsulate in separate functions.

In this chapter, we introduce the basics of writing functions. The functions that we create here perform simple tasks, such as those described in the following examples. Later, we demonstrate how to take on larger tasks and how to organize our work into multiple functions, each carrying out a separate subtask. For example, in Chapter 6, we develop a resampling technique to select a model for prediction, in Chapter 7, we design and carry out a simulation study of a complex random process, and in Chapter 14, we develop a programmatic approach to read and merge data from different sources.

Example 5-1 Calculating BMI

In Q.1-1 (page 12), we worked with artificial data from a 14-member family. These data are similar to those collected by the Centers for Disease Control (CDC) in the Behavioral Risk Factor Surveillance Survey (BRFSS). The survey respondents provide their height (inches), weight (pounds) and desired weight (pounds). Their Body Mass Index (BMI) is calculated and provided as part of the BRFSS data set. Suppose in our analysis, we are interested in the relationship between a respondent's BMI and 'desired' BMI (calculated from desired weight). To study this relationship, we can create a function for computing BMI from height and weight (or desired weight in this case). We take the trouble to write a function to do this simple calculation because we anticipate needing to compute BMI for other data and possibly for units other than pounds and inches.

■

Example 5-2 Tally Run Times in Seconds

In Q.4-2 (page 151) we extracted run times for the men's 1500 meter race from a table on a Wikipedia page. In that process, we made a simple calculation to convert the times into seconds to assist us in making plots. There are several other tables of world records on that page, including those for women and pre-IAAF (International Association of Athletics Federations) records, and if we have a function to convert times reported in minutes and seconds into seconds, then we can use our function with the times in these tables. Even though this calculation is simple, there are several good reasons to create a function for this task. We can hide the code that converts numeric times to the POSIX format in the function. If we find that we made a mistake in our code, then we can correct it in one place, i.e., in the function, and rerun our code to process the data again. Otherwise, we need to correct the mistake in multiple places, and we run the danger of making more mistakes. Additionally, the pre-IAAF times are formatted differently so we can design our function to handle these variations in format. Finally, we can later generalize our function to convert times for other races, e.g., marathons with times that include hours as well as minutes and seconds.

■

Example 5-3 Taking the Logarithm of 0

The log transformation is very useful in data analysis. For example, we often log-transform data for variance stabilization so that the data distribution is less skewed and more symmetric. Also when plotting data, we often use a log transformation to help us better understand relationships between variables (see Section 3.3.1). In many situations, the data include a few 0 values and produce errors when we take a logarithm. A common solution is to add a small positive value to all of the data. That is, we essentially compute $log(x + c)$, where c is some positive constant. Popular values for c are 1, a small value such as 0.001, or a value that depends on the minimum data value, e.g., $min(x)/100$ (where the minimum is over the positive x values). We can imagine having our own 'log' function can be useful. It can give us versatility in specifying the constant added to the data.

■

Example 5-4 Converting Liquid Measures into Tablespoons

While we primarily use *R* for data analysis and simulation studies, we sometimes use it as a calculator. For example, we can use *R* to convert liquid measures in metric units into, say, tablespoons. If we find that we often need to make these conversions, then we can write a function for this purpose where we specify the amount and units of the liquid and the function provides the equivalent number of tablespoons.

■

Example 5-5 Reading Thousands of Data Files

We often begin a data analysis by reading data stored in a file into *R* with one of the functions described in Chapter 2 and Chapter 4, such as read_delim(), readLines(), and fromJSON(). At times the data are arranged in many files, e.g., the Kiva data in Q.4-4 (page 153) are provided in 1975 files. When this happens, we need to read from all of these files and organize the results into a structure for analysis. To determine the file names, we can use a file finder, such as Spotlight, and add these to our code. However, when we have thousands of files and so thousands of file names, then using a file finder is not feasible. We want to automate this process and avoid typing or copy-and-pasting thousands file names. We can write a function for this purpose.

■

Example 5-6 Generating Fibonacci Numbers

The Fibonacci numbers are generated by setting $F_0 = 0$, $F_1 = 1$, and then using the recursive formula below for finding subsequent numbers:

$$F_n = F_{n-1} + F_{n-2}$$

The first 10 Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. For fun, we can write a function to compute F_n, for any n. Given the recursive nature of the definition, i.e., the nth number is defined in terms of the $n - 1$st and $n - 2$nd numbers, we need to build up F_n from earlier values in the sequence. We see later that there are several ways to do this, and we can carry out a computer experiment to compare the efficiency of these approaches.

■

Example 5-7 Playing Penney's Game

A simple version of Walter Penney's game [3] has two players each choose a sequence of 3 heads and tails. That is, player A selects a sequence of 3 heads and tails, shows this sequence to player B, and then, player B selects his/her sequence of 3 heads and tails. After that a (fair) coin is tossed until either player A's or player B's sequence appears. Whoever has the sequence that appears first wins. While, we can work out analytically player B's strategy for choosing a sequence given player A's sequence, we can also gain intuition by using the computer to carry out a simulation study of the game. To do this, we can create a function to play one round of the game for a given pair of strategies and then play the game many, many times for various strategies and compare the number of wins.

■

We use these examples in this chapter to demonstrate how to write functions, including issues of specification of formal arguments, control flow, return values, efficiency, and programming style guide lines.

5.2 Steps in Writing Functions

Many of the examples in Section 5.1 can be addressed with one or two lines of code. It may seem as though there is little need to write a function to carry out these simple tasks, but they offer excellent examples of how to go about writing a function, and often they can be generalized to handle more complex situations. Moreover, it is typically the case that when we have a larger problem to solve, we break the work up into smaller tasks which are carried out by functions that contain only a few lines of code.

We begin with the problem of calculating BMI from Q.5-1 (page 194) to demonstrate the process of writing a function. Briefly, these steps are: express the task in words; write code to carry out this task for a specific example; abstract the code by identifying inputs and constants and using general names and, where appropriate, default values for these; define the function and incorporate the generic code into the body of the function; and test the function with the original special case and additional test cases. Of course, as we gain expertise in function writing, we can often collapse some of these steps; we often do not need to write a concrete version of the code and instead begin by identifying the function inputs and writing a generic version of the code.

Steps in Writing Simple Functions

Explain Begin by describing in words (and possibly formulas) what you want the function to do.

Code Write code to address a specific example of the task.

Abstract Identify the inputs in your code that you want the caller to specify. Rewrite your code, if needed, to use generic variable names rather than specific quantities/variables for these inputs.

Encapsulate Wrap this generalized version of the code into a function with the inputs as formal arguments. Make sure that your function returns the desired object.

Test Check the function works as expected with your original data. Try the function on test cases that you create to check various aspects of the function.

5.2.1 Calculating BMI from Height and Weight

Explain the Task
We begin by clearly stating the task that we want the function to carry out. In our example, we want the function to compute BMI from height and weight. We can look up the formula

on the Web to find:

$$BMI = \frac{703 * weight}{height^2},$$

for weight measured in pounds and height in inches.

Code the Task
Next, we write code to carry out the task. In this code, we use data from a particular example in order to make the task concrete. For example, we can use our family data from Q.1-1 (page 12). Recall that we have the two vectors `fheight` and `fdesiredWt`. We translate the formula into a single line of code that uses these two variables with

```
703 * fdesiredWt / fheight^2
```

```
       a        b        c    ...
25.10714 19.73755 23.08594
```

It appears that our code is correct; there are no syntax errors and the results have reasonable values.

Abstract the Code
The code that we have written works for a particular set of data, i.e., for `fheight` and `fdesiredWt`. We want to abstract the code to use general variables for height and weight, such as `ht` and `wt`. These variables will be the inputs to our function. Our generalized code appears as

```
703 * wt / ht^2
```

If we try to evaluate this code at the *R* prompt, we get an error because these variables are not defined in our workspace.

Encapsulate Code in a Function
We are ready to create a function, and we begin by assigning the function to a name. We do this just like we do for any assignment, i.e.,

```
calcBMI =
function(ht, wt)
{
}
```

We have created a function that takes two arguments (*ht* and *wt*) and assigned it to the variable `calcBMI`. Currently, our function doesn't do anything. We need to place the code we wrote earlier (the generic version) between the curly braces in the function definition. We do this with

```
calcBMI =
function(ht, wt)
{
  703 * wt / ht^2
}
```

We have written a function!

Although this function is very short, we typically do not define a function at the prompt. This approach can be frustrating when we make small typographical errors. Instead, we write our function in a text file (e.g., an *R* script in RStudio) and source the function into *R* with the source() function (or click on the `source` button in the RStudio menu bar).

Other approaches, include writing code in a code chunk in an *Rmd* file and running the chunk, or writing code in a text editor such as Emacs, which has an interface to *R*.

Test the Function
After we source our calcBMI() function into *R*, the next step is to test the function. We can, for example, compare the values in `fbmi` to the return value from

```
head(calcBMI(fheight, fweight))
```

```
       a        b        c        d        e        f 
25.10714 21.45386 24.40514 24.43039 18.02124 28.88625 
```

```
head(fbmi)
```

```
[1] 25.16239 21.50106 24.45884 24.48414 18.06089 28.94981
```

The values are close but not exact. This is not too surprising as there is rounding error in our use of 703 for the constant and we don't know if `fheight` and `fweight` were used in computing `fbmi` or if more precise versions of these measurements were used. Nonetheless, `fbmi` and our calculated BMI values should be close, and we can compare these two sets of values to see if they are within, say, 0.1 of each other with

```
all(abs(fbmi - calcBMI(fweight, fheight)) < 0.1 )
```

```
[1] TRUE
```

Indeed, all of our calculated BMI values are within 0.1 of the reported values in `fbmi`.

5.2.2 Generalizing Code

The process of writing a function is often iterative. After our first attempt at writing our function, we typically make the function more robust and general. That is, we iteratively refine it by testing our code with different inputs, refining code to handle bad inputs, generalizing code to handle a greater variability of inputs, and streamlining our code so it is clearer and more efficient. We repeat this process until we are satisfied with the function.

For instance, it is good to avoid hard coded constants and to make them variables either within the function or formal arguments so that the caller can specify them. As an example, notice the constant 703 in our calcBMI() function. This constant accounts for the units of measurement, i.e., that height is measured in inches and weight in pounds. If our measurements are in other units, then the constant needs to change, e.g., when height is in meters and weight in kilograms, then the constant is 1. We can generalize our function to have a formal argument for this constant. It makes sense for it to have a default value, and in this case we choose 703 for the default because we expect our measurements to be in pounds and inches. Our revised function appears below:

```
calcBMI =
function(ht, wt, unitConstant = 703)
{
  unitConstant * wt / ht^2
}
```

Notice that we make the new argument, *unitConstant*, the 3rd formal argument in our function. If we have already written code that uses this function, then the placement of the new argument and its default value of 703 means that any existing code which calls this function continues to work as expected. Further, we typically place parameters that have a default value after those that are required in the function's signature.

5.2.3 Commenting Code

Another important part of function writing is documentation. We can add comments to our code by preceding text with the # character. All text after the # to the end of the line is ignored by *R*. It is there for people to read. We typically add informative remarks to remind ourselves and others of the purpose of command(s) and the expected inputs and output. Our calcBMI() function is quite simple, except the use of the constant to change units may be a bit unclear so we add a comment explaining this:

```
calcBMI =
function(ht, wt, unitConstant = 703)
{
# use unitConstant = 1 for wt in kg and ht in m
# for others: convert to kg and m to find constant,
# e.g., for lb and in: 0.4536 kg/lb / (0.0254 m/in)^2 is 703

  unitConstant * wt / ht^2
}
```

We close this section on how to write functions with 2 more examples. We consider the tasks of converting race times into seconds (see Q.5-2 (page 194)) and developing our own logarithm function that adds a small positive quantity before taking the logarithm (see Q.5-3 (page 194)). In each case, we explicitly go through the steps to writing a function, i.e., explain, code a concrete case, abstract, encapsulate, and test.

5.2.4 Reporting Run Times in Seconds

Explain
Our task is to convert a character vector containing times into a numeric vector of times measured in seconds, e.g., "3:55.8" is converted to 235.8.

Code
Let's set up some sample data for developing our function. We can take the first few values from the 1500m world records in Section 4.4.1, i.e.,

```
testTimes = c("3:55.8", "3:54.7", "3:52.6", "3:51.0")
```

We can start with the code we developed in that section. That is, we use one of thePOSIX() functions in *R* to convert the string into a date-time object with

```
tempTime = as.POSIXlt(testTimes, format = "%M:%OS")
```

Then we convert these into numeric seconds with

```
tempTime$min * 60 + tempTime$sec
```

```
[1] 235.8 234.7 232.6 231.0
```

We now have completed the task for a specific set of values (those in `tempTimes`).

Abstract
If we are to convert this code into a function, we need the caller to provide the string of times. Let's call it *timeString* to make it clear that the input are strings. Do we need to specify any other formal arguments? As mentioned earlier, if we want to generalize this

function, then we may want the caller to supply the format of the string. It makes sense to provide a default format; we can use a value that works for our Wikipedia tables (e.g., `"%M:%OS"`). Let's name this argument *format*.

Encapsulate
We have settled on the parameters for the function so we can define our function with, e.g.,

```
timeInSec =
function(timeString, format = "%M:%OS")
{
}
```

Notice that we placed *timeString* first because it is the main argument to the function and it has no default value. We fill in the function body with the code above, first swapping the parameter names for the variables and constants in that code. This leads to the function below:

```
timeInSec =
function(timeString, format = "%M:%OS")
{
  tempTime = as.POSIXlt(timeString, format = format)
  tempTime$min * 60 + tempTime$sec
}
```

Notice that the `format` argument in our function is supplied by the caller, and we pass it into as.POSIXlt() as the value for this function's *format* argument.

Test
We test our function with

```
timeInSec(testTimes)
```

```
[1] 235.8 234.7 232.6 231.0
```

These values match those previously calculated. We leave to the exercises the task of developing additional tests for our code.

We used the *format* argument to generalize our function. Can you see any potential problems with it? If we look at the pre-IAAF times, we see that they appear as, e.g., `"4¬ :16 4/5"`. Can the as.POSIXlt() function handle this format? Also, what if the times are for a marathon? There is a Wikipedia table for marathon records and these times appear as, e.g., `"2:27:49.0"`. Does our function work properly with this format? We leave the detailed answers to these questions to the exercises.

Deciding on the Arguments to a Function

We consider the following questions when determining the formal arguments to a function:

- What data/information must the caller provide? Create arguments for these data.
- Is there a reasonable default for an argument? If so, then provide it.
- Are any of the arguments required? Typically, we place required arguments before those with default values.
- Is a variable computed from the inputs in the first few lines of code in the function?

If so, consider adding this variable to the formal arguments with the expression as a default value.

- Are there constants in the code? If yes, give them names and place near the top of the function. Consider whether it generalizes the function if they are arguments.

5.2.5 Taking the Logarithm of '0'

Explain
Our task is to take the logarithm of a numeric vector, except that our 'log' function adds a small positive value to all of the elements. That is, we compute $log(x+c)$, for some constant c. Often this constant is 1, but if the other values in the vector are small then the constant is typically a smaller value, such as 0.001. Also, this constant may depend on the data, such as 1/10 of the smallest positive value in the data.

Code
We make up some data for developing our function. Since our explanation noted a few different constants, let's create a few vectors for use with these different constants,

```
x = 0:3
y = c(0.1, 0, 1, 0.2)
z = c(0.0005, 0.1, 1, 0)
```

Our code is very simple, we want only to compute the log of these vectors (with the addition of a small constant), e.g.,

```
log(x + 1)
```

```
[1] 0.00 0.69 1.10 1.39
```

```
log(y + 0.001)
```

```
[1] -2.293 -6.908  0.001 -1.604
```

```
log(z + min(z[z > 0])/10)
```

```
[1] -7.50559 -2.30209  0.00005 -9.90349
```

Abstract
To convert this code into a function, we need the caller to provide the numeric vector, which we simply call *x*. Also, we have tried our approach for 3 different constants so it makes sense to make the constant a parameter for the function too. We call it *amt* rather than *c* because it's good programming practice to avoid using common function names for variable names. It can lead to unexpected results! It's also good practice to use names that make clear the purpose of the parameter.

Encapsulate
We define our function with, e.g.,

```
logPlus =
function(x, amt = 1)
{
}
```

As in earlier examples, we place *x* first because it is the main argument to the function and has no default value. We use 1 for the default value of *amt*, but it can be reasonable to use another default value. For example, if we want the default to depend on *x*, then our function definition appears as, e.g.,

```
logPlus =
function(x, amt = min(1, min(x[x > 0]) / 10))
{
}
```

In this function signature, we set the default value of *amt* as the smaller of 1 and 1/10th the minimum of the positive data values. Notice that we have used the input from another parameter (*x*) to compute the default value of *amt*. We might be inclined to think this would cause an error, but it does not. See Section 5.4 for details on how functions are invoked in *R*, i.e., how the parameter values are used to create variables available within the function's workspace. We complete our function by simply adding one line of code to the body of the function, i.e.,

```
logPlus =
function(x, amt = 1)
{
  log(x + amt)
}
```

Test
We test our function and compare the results to our earlier trials with

```
logPlus(x)
```

```
[1] 0.00 0.69 1.10 1.39
```

Recall that the `x` used in the function call above is a variable in our global environment, and it is different from the `x` in the body of our function. We discuss this in greater detail in Section 5.4. Also, the return value from `logPlus(x)` matches the previous calculation, as do the calls: `logPlus(y, 0.001)` and `logPlus(z, min(z[z > 0])/10)`.

5.3 Defining a Function

We have seen several examples of function definitions in Section 5.2. More formally, the syntax for defining a function is:

```
functionName =
function(formal arguments)
{
  statements
}
```

The keyword `function` tells *R* that we want to create a function, and the formal arguments (aka parameters) to the function are supplied in the parenthesis as a comma-separated list of names. If a parameter has a default value, then it is supplied in this list. For example, with `function(ht, wt, unitConstant = 703)`, we have specified 3 formal arguments *ht*, *wt*, and *unitConstant* and assigned *unitConstant* a default value of 703. We assign the function to a name in order to call it, e.g., we call calcBMI() with `calcBMI(fheight, fweight)`.

We have adopted the convention of placing the `function` keyword and the opening curly brace on their own lines. This is not required by *R* and others use different conventions. For example, the following syntax is equally valid:

```
functionName = function(formal arguments) {
  statements
}
```

Also, if our function contains only one line of code, we can drop the curly braces entirely and define our function as

```
functionName = function(formal arguments) one statement
```

We discuss additional style guidelines for coding in Section 5.8.

Defining a Function

The general format for a function is:

```
myFunc =
function(arg1, arg2, arg3 = 17)
{
  body
}
```

- The keyword `function` tells *R* that we want to create a function.
- The formal arguments (i.e., parameters) of the function are provided between the parentheses as a comma-separated list of parameter names. If a parameter has a default value, then it is supplied here. In this example, there are 3 parameters, called *arg1*, *arg2* and *arg3*; the first 2 parameters are required and the 3rd has a default value.
- We assign the function to a variable, in this case `myFunc` so we can use this name when we call the function. We need not assign the function a name, in which case it is an anonymous function. Anonymous functions are useful when they are supplied as inputs to other functions, and we do not expect to use them in other settings.
- The body of the function appears between the opening and closing curly braces. The body consists of the expressions needed to carry out the function's task. The function returns only one object when it completes execution.

A very common style of writing and managing functions is to create them in a text or script file. Then, when we want to test them, we use the source() function (or the Source button in the RStudio environment) to read and evaluate the commands in that file. This call to source() defines the functions as regular variables in our workspace that are ready for us to use and check. If we need to modify the function to fix a bug or make it more general, then we edit the code in the file and re-source the file. We can define many functions in one file and we can include code that is not part of a function. For example, we can set up tests which are placed in the file after the function definitions; then, when we source in the file, the functions are re-defined and after that our tests are run. There are other ways to source our functions into the global environment, including using code chunks in *Rmd* files and text editors such as Emas that interface to *R*.

5.3.1 Anonymous Functions

We saw in Section 5.2 that the keyword `function` in the function definition tells *R* that we want to create a function, and we assign that function to a variable in order to call it. That is, we use the regular assignment operator to give our function a name. However, we do not have to assign the function to a variable. In this case, we have an anonymous function. Typically we use anonymous functions when the functions is used only in one particular context. We give an example with applying a function over the elements of a list.

Example 5-8 Counting Years of Operation for Weather Stations

In Section 4.2.3, we explored data on rainfall in the Colorado Front Range. The data are organized in a list, `FrontRangeWeather`, that includes a list called `days` of numeric vectors of dates of the recorded precipitation for each weather station. The weather stations are in operation for varying amounts of time. Suppose that we are interested in the number of years that each weather station is in operation. Note, if a station is in operation for 10 days in, say, 1967 then we count that as 1 year.

The weather station data is in the file *FrontRangeWeather.rda*. After we load the *rda* file into our *R* session, we can find the years of operation for, say, the 18th weather station in the list with, e.g.,

```
length(unique(floor(FrontRangeWeather$days[[18]])))
```

```
[1] 8
```

To find the number of years of operation for each of the 56 weather stations, we can use sapply(). However, unlike earlier applications of sapply() and lapply(), there is no single function in *R* to apply to each vector in `days` so we supply our own function. Our function needs to compute the length of the unique values in a vector of dates after these dates have been reduced to integers. Our function for this task is

```
countYears =
function(x)
{
  length(unique(floor(x)))
}
```

Here countYears() encapsulates the code for the 3 nested function calls we performed earlier for the 18th weather station. We apply countYears() to each element of `days` in `FrontRangeWeather` with

```
sapply(FrontRangeWeather$days, countYears)
```

The return value is a numeric vector with 56 values.

Since we don't imagine using this function in other situations, we don't need to name it. Instead, we can pass an anonymous function into sapply() with

```
sapply(FrontRangeWeather$days,
       function(x) length(unique(floor(x))))
```

```
st050183 st050263 st050712 st050843 st050945 ...
      53       34       20       54       19
```

Notice that we have dropped the curly braces in our function definition because our function contains only one expression. Typically, anonymous functions are quite short.

Finally, we point out that we can carry out these 3 operations in sequence and so avoid creating a function. To do this, we use intermediate variables to store the results, e.g.,

```
years = sapply(FrontRangeWeather$days, floor)
uniqueYrs = sapply(years, unique)
sapply(uniqueYrs, length)
```

This approach is neither as clear or efficient as the approach that uses anonymous functions. It is less efficient because we have called sapply() 3 times and created extra copies of the `days` data, and it is less clear because this simple operation is split into 3 separate steps. However, this may be a natural approach to take when first developing our code.

■

5.4 Calling a Function

5.4.1 Argument Matching by Name and Position

When we call a function, *R* maps the inputs in the function call to the formal argument names in the function's signature and puts them into a call frame with variables corresponding to these formal argument names. One can think of this process in the following way. *R* first creates a list (or table) of variables with one variable name for each of the formal arguments, and default values are provided as values for each formal argument that has one. After that, *R* maps the inputs specified in the function call to these variables in the call frame and updates their values. *R* first matches arguments by name and then by position.

An example will help concretize this process. In our calcBMI() example, our function signature is

```
function (ht, wt, unitConstant = 703)
```

When we call calcBMI(), *R* creates a call frame with variables named `ht`, `wt`, and `unitConstant`. *R* then assigns the value 703 to `unitConstant`. At this point, *R* then maps the inputs specified in the function call to the variables in the call frame. Suppose we call calcBMI() with:

```
calcBMI(1.4, 44.2, unitConstant = 1)
```

R first matches named arguments. In this case, *R* recognizes the name `unitConstant` and assigns the value 1 to `unitConstant` in the call frame table. Now, since there are no other named arguments in the function call, *R* matches arguments by position. The value 1.4 is assigned to the first unmatched formal argument, which is *ht*. The next step is to map 44.2 to the next unmatched formal argument, which is *wt*. These steps for creating the call frame are displayed in Table 5.1. Now that the call frame is set up, *R* begins executing the code in the body of the function.

As another example, consider the following call to matrix(),

TABLE 5.1: Call Frame Set Up:
Signature: calcBMI = function(ht, wt, unitConstant = 703)
Call: calcBMI(1.4, 44.2, unitConstant = 1)

Initial Variable	**Set Up Value**	**Match Variable**	**Name Value**	**Match Variable**	**Position Value**
ht		ht		ht	`1.4`
wt		wt		wt	`44.2`
unitConstant	`703`	unitConstant	`1`	unitConstant	`1`

```
matrix(, 2, 3)
```

```
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
```

This creates a 2 by 3 matrix with an `NA` value in each element. Let's consider how the arguments in the function call are matched to the formal arguments of the matrix() function, which are:

```
function (data = NA, nrow = 1, ncol = 1,
           byrow = FALSE, dimnames = NULL)
```

First, all 5 formal arguments have default values and the call frame is set up with these variables and values. Next, we handle all the named arguments in the function call. In this case, there are no named arguments, so we skip to the next stage of matching by position. The first argument is intentionally missing (i.e., we have not provided an argument before the first comma in `( , 2, 3)`). *R* considers this to be the first formal argument (*data*) and leaves its default value in the call frame. Then *R* processes the value 2. This is assigned to the next formal argument, which is *nrow*. Lastly, the 3 is matched to the next (3rd) formal argument, which is *ncol*. The call frame now has the values `NA`, 2, 3, `FALSE`, `NULL` for the variables `data`, `nrow`, `ncol`, `byrow`, and `dimnames`, respectively.

The Call Frame in a Function Call

When we call a function, *R* maps the inputs in the function call to the formal argument names in the function's signature and puts them into a call frame with variables corresponding to these formal argument names. The sequence of steps go as follows:

- Create a frame of variables with one variable name for each of the formal arguments.
- Assign the default value, for each formal argument that has one, to the corresponding variable in the frame.
- Map the inputs specified by name in the function call to the variables in the call frame. Names in the function call can match partially, provided they uniquely identify a formal argument of the function. If a name doesn't match and the function has a ... argument then this variable and value are added to the call frame.
- Map the remaining inputs provided in the function call by position.

TABLE 5.2: Call Frame Set Up:
plotRatio = function(r, k = 1, date = seq(along = r), ...)
plotRatio(1:10, 2, col = "red", main = "A Title")

Initial	**Set Up**	**Match**	**Name**	**Match**	**Position**
Var	**Value**	**Var**	**Value**	**Var**	**Value**
r		r		r	`1:10`
k	`1`	k	`1`	k	`2`
date	`seq(along = r)`	date	`seq(along = r)`	date	`seq(along = r)`
...		col	`"red"`	col	`"red"`
		main	`"A Title"`	main	`"A Title"`

When an expression is provided as the default value for an argument, this expression is not evaluated until code that references this variable is evaluated. Also, if a variable is referenced in the code that is not in the call frame and has not been created by earlier statements in the function, then *R* looks for this variable in the function's parent environment.

5.4.2 The ... parameter

The ... mechanism is a special kind of formal argument. If *R* cannot match an argument by name or by position and the function has ... in its signature, then the argument is added to the call frame. This is a mechanism by which we can have an arbitrary number of arguments for the call frame. For example, with c(), the signature is

```
function (..., recursive = FALSE)
```

The ... argument allows us to concatenate an arbitrary number of variables together. While ... does allow us to have any number of arguments in functions like c(), save(), sum(), and so on, this argument also has another purpose. It allows us to write top-level functions that use ... to take arguments which our function then passes on to lower-level functions by including ... in that function's call.

For example, consider the plotRatio() function defined below

```
plotRatio =
function(r, k = 1, date = seq(along = r), ...)
{
  plot(date, r, type = "l", ...)
  abline(h = c(mean(r), mean(r) + k * sd(r),
               mean(r) - k * sd(r)))
}
```

Notice that we allow the caller to pass additional arguments to plot() via .... The caller can use this mechanism to specify a title, axis labels, etc. For example, the following function call, `plotRatio(1:10, col = "red", main = "A Title")` passes the *col* and *main* arguments via the ... to plot() (see Table 5.2).

5.4.3 Lazy Evaluation

The call frame for

```
plotRatio(1:10, 2, col = "red", main = "A Title")
```

that is displayed in Table 5.2 shows the expressions `1:10` and `seq(along = r)` as the values for `r` and `date`, respectively. We might think the value for `r` should be the integer vector `1 2 3 ... 10`, rather than the expression `1:10`. The reason for this is that *R* does not evaluate these expressions and assign their value to the associated variable until *R* evaluates an expression that refers to this variable. This is called lazy evaluation.

As an example, consider the `logPlus` function that we wrote in Section 5.2.5. We reproduce it below for reference

```
logPlus = function(x, amt = 1) log(x + amt)
```

We modify this function in 2 ways to show how lazy evaluation works. We change the default value for *amt* to `min(x[x > 0])`. We also add code to check to see if any values in *x* are negative, and if they are, then we replace *x* with its absolute value. The revised function appears as

```
myLog =
function(x, amt = min(x[x > 0]))
{
  if (any(x < 0)) x = abs(x)
  log(x + amt)
}
```

When we call the function with `myLog(0:3)`, the call frame has the expression `0:3` associated with *x* and the expression `min(x[x > 0])` associated with *amt*. The expression `0:3` is evaluated the first time *R* evaluates an expression that contains `x`. This is in `any(x < 0)`. The expression `min(x[x > 0])` is not evaluated until the second line of code, i.e., when `amt` is used in `log(x + amt)`. At this point, *R* assigns 1 to `amt` and our function returns the logarithm of the values `1 2 3 4`, i.e.,

```
[1] 0.00 0.69 1.10 1.39
```

We can confirm that our function behaves as expected by comparing the return value with `log(1:4)`.

What happens when we call the function with `myLog(c(-2, -1, 0, 4, 5))`? The smallest positive value of the input is 4. Does our function return the logarithm of `6 5 4 8 9`? Let's try it.

```
myLog(c(-2, -1, 0, 4, 5))
```

```
[1] 1.10 0.69 0.00 1.61 1.79
```

```
log(c(6, 5, 4, 8, 9))
```

```
[1] 1.8 1.6 1.4 2.1 2.2
```

These values do not match! Why? The reason has to do with lazy evaluation.

When we call `myLog(c(-2, -1, 0, 4, 5))`, the call frame contains expressions for the values of `x` and `amt`. *R* does not evaluate these expressions until it needs to. With the first line of code (the `if` statement), the expression for the value of `x` in the call frame is evaluated and assigned to `x` so `x` is now `2 1 0 4 5`. The variable `amt` appears in the second line of code, so the expression that defines `amt` is not evaluated until we take the logarithm of `x` and at this point, `x` no longer has negative values. This means that `amt = min(x[x > 0])` assigns 1 to `amt` and the return value of our function call is the logarithm of `3 2 1 5 6`. When we think about it, lazy evaluation has provided a reasonable value for `amt`. If the expression `min(x[x > 0])` was evaluated earlier, then the amount may not be what we want. For example, consider `logPlus(c(-0.1, 1))`. Do we want `amt` to be `1` or `0.1`? The latter seems preferable.

5.4.4 The Search Path

In Section 1.13.2, we introduced the search path that *R* follows to find variables when we refer to them in our code. We saw there that *R* chains together a collection of places in which to search for variables and functions. When we type a command at the prompt, *R* begins to look for variables in the global environment. If a variable is not found, then *R* looks in the parent environment and works its way along the chain of environments until it finds the variable. If it doesn't find the variable in any environment then we receive an error. What happens when we call a function and an expression in the function uses a variable?

When we call a function, *R* sets up a call frame for the function. As noted in Section 5.4 and Section 5.4.3, *R* populates the call frame with the function's arguments. Additionally, as *R* executes the expressions in the function, if new variables are created, then these are added to the call frame. The parent of this call frame is the environment in which the function was defined, which is typically the global environment. When our code refers to a variable, *R* searches for it in the call frame, and if it's not there then *R* searches for the variable in the parent environment and along the search path.

To provide an example, we return to our simple function calcBMI() from Section 5.2.1 and define 2 alternative functions. We provide the original function for reference,

```
calcBMI =
function(ht, wt, unitConstant = 703)
{
  unitConstant * wt / ht^2
}
```

Our first variant of calcBMI() is

```
calcBMI.alt1 =
function(ht, wt, unitConstant = 703)
{
  convert(wt / ht^2)
}
```

We also define convert() with

```
convert = function(x) x * unitConstant
```

Notice that convert() simply multiplies its input by `unitConstant`. Our second version appears as

```
calcBMI.alt2 =
function(ht, wt, unitConstant = 703)
{
  convert = function(x) x * unitConstant
  convert(wt / ht^2)
}
```

Notice that the function definitions for all 3 versions of calcBMI() are identical. Also notice that the body of calcBMI.alt2() contains a function definition for a convert() function.

Before we explain the differences between our 2 variants, we call them a few times. We first call calcBMI.alt1() with

```
calcBMI.alt1(66, 130)
```

```
Error in convert(wt/ht^2) : object 'unitConstant' not found
```

This variant produces an error. However, when we call the other version of the function with the same inputs, i.e., `calcBMI.alt2(66, 130)` we receive

```
[1] 21
```

This seems somewhat curious. If we call convert() with `convert(130 / 66^2)` then we get the same error as above. Now let's define `unitConstant` at the prompt with

```
unitConstant = 17
```

Then we find

```
calcBMI.alt1(66, 130)
```

```
[1] 0.51
```

Now, calcBMI.alt1() now longer gives an error, but it gets the wrong answer. However, `calcBMI.alt2(66, 130)` still returns the correct answer of `21`.

What is going on here? It has to do with the search path and where convert() and `unitConstant` are defined. Both calcBMI.alt1() and calcBMI.alt2() are defined in *.GlobalEnv*. Also, the convert() function that calcBMI.alt1() calls is defined in *.GlobalEnv* and convert() function that calcBMI.alt2() calls is defined in calcBMI.alt2()'s call frame. We can confirm this with the find() function. That is, `find("calcBMI.alt1")`, `find("calcBMI.alt2")` and `find("convert")` all return `".GlobalEnv"`

When we call calcBMI.alt1(), *R* sets up a call frame and places `ht`, `wt` and `unitConstant` in it. Then, when convert() is called with `convert(wt / ht^2)`, *R* sets up another call frame. This call frame contains `x`, but it does not contain `unitConstant` because it is not an argument to convert() and it is not created in the body of convert(). When *R* evaluates the expression `x * unitConstant`, *R* searches for `unitConstant`. Since `unitConstant` is not in the call frame for convert(), *R* looks in the parent environment to the function call, which is .GlobalEnv because convert() was defined there. The first time we called calcBMI.alt1(), *R* does not find `unitConstant` in the call frame, it's parent environment, or any of the environments in the search path. This is why we get an error. The second time we call calcBMI.alt1(), *R* finds `unitConstant` in the global environment; it has a value of 17 and the function returns `0.51`.

On the other hand, when we call calcBMI.alt2(), *R* sets up the call frame with `ht`, `wt` and `unitConstant`, just as for calcBMI.alt1(). However, when *R* executes the first expression in calcBMI.alt2(), the convert() function is added to the call frame. Next, *R* evaluates the 2nd expression, i.e., `convert(wt / ht^2)`, and sets up a call frame for this function call. Since convert() was defined in the first call frame, this is its parent environment. As before, *R* does not find `unitConstant` in call frame for `convert(wt / ht^2)`, but it does find it in the parent environment, i.e., the call frame for the `calcBMI.alt2(66, 130)`. This is why we get the answer `21`. This return value does not change when we assign `unitConstant` to 17 in the global environment because *R* finds `unitConstant` before needing to look in .GlobalEnv.

This example may seem silly, but when we collect code that we have been developing into a function, it is easy to forget some of the variables that need to be defined in the code or added to the function's signature. When we test our code, it works because these variables are found in the *R* session (i.e., in globalenv()). However, our function may no longer work in a new *R* session because a variable can't be found, or worse, the function gives the wrong results because the global variables have been changed. For these reasons, it's poor programming practice to write functions that depend on variables in the global environment. To help us avoid this, we can use the findGlobals() function in the `codetools` package [4], which identifies globals in our functions. For example,

```
library(codetools)
findGlobals(calcBMI.alt2)
```

```
[1] "{" "*" "/" "^" "="
```

The findGlobals() function looks for globally defined variables and functions. We see here that for calcBMI.alt2() these are only functions in base *R*. On the other hand, `findGlobals(calcBMI.alt1)` returns

```
[1] "{"        "/"        "^"        "convert"
```

This indicates that convert() is defined in the global environment, and when we check it, we find

```
[1] "*"             "unitConstant"
```

That is, convert() references `unitConstant`, which as we discovered already, is not defined in the function's call frame.

5.4.5 Partial Matching

R has a mechanism that allows us to abbreviate argument names when they unambiguously identify the particular argument. For example, rather than calling matrix() with `matrix¬(1:10, ncol = 5)`, we can call it with `matrix(1:10, nc = 5)` because *nc* uniquely matches the first 2 letters of *ncol* among the arguments to the function. This style of abbreviated argument names is called partial matching and can make code much harder to read and confusing. It can also lead to some very subtle and frustrating bugs so we suggest avoiding using it.

5.5 Exiting a Function

A simple way for a function to exit and hand control back to the caller is with an explicit return() function call. This return() can be called with no arguments or with a single value which can be any *R* object. In many functions, we often want to return the last computed value. *R* makes it easy for us to do this by always returning the result from the last expression evaluated, i.e., we don't necessarily need to use return(). For example, there is no return() function in our calcBMI() function so the result from the evaluation of the last (and only) line of code in this function is returned, i.e., the result is `( unitConstant * wt / ht^2`. For purposes of clarity, we can add a return to calcBMI() with

```
calcBMI =
  function(wt, ht, unitConstant = 703)
{
# unitConstant = 1 if wt in kg and ht in m
# for other units: determine constant by converting to kg and m
# e.g., for lb and in: 703 = 0.4536 kg/lb / (0.0254 m/in)^2
  return(unitConstant * wt / ht^2)
}
```

Now, we have made the return value explicit.

If we want to return two or more objects, then we must put them together into 1 object and return that single object. For example, suppose in calcBMI(), we want to return both the BMI vector and the constant that was used in creating it. To do this, we can create a named list with these two elements and then return that, e.g.,

```
return(list(bmi = (wt * unitConstant / ht^2),
            unit = unitConstant))
```

Although this is a list with two named elements, we are returning a single *R* object.

Explicitly calling return() allows us to exit from a function within loops and conditional statements. We show examples of this in Section 5.6. If we don't want to return anything, then we can call return() with no argument. Note, however, that this actually returns `NULL`. Also, if no expression is evaluated then `NULL` is returned.

5.6 Conditionally Invoking Code

In the examples we have worked on in the chapter so far, we have evaluated each statement in the function body sequentially i.e., one expression after another in order from top to bottom. Control flow structures allow us to selectively evaluate statements, repeat the evaluation of statements, and stop or suspend evaluation of code. One primary flow control statement is the `if-else` statement (and the related ifelse() and switch() functions). We cover these first, and then introduce mechanisms for looping, which augment the suite of apply functions introduced in Chapter 4. Although we typically control the flow of code in functions, conditional statements and loops are not restricted to function bodies.

The `if-else` Statement

The `if-else` statement allows us to perform computations conditionally. The basic format is:

```
if (condition) {
  statements-A
} else {
  statements-B
}
```

The condition in parentheses evaluates to TRUE or FALSE (i.e., a logical vector of length 1), and the statements in the first set of curly braces (the A set) are executed if the condition is TRUE, otherwise the B set of statements in the curly braces following the else are evaluated. The else construct in the if-else statement is optional, and multiple if-else statements can be nested. The following example demonstrates a simple use for an if-else statement.

Example 5-9 Generalizing the BMI Calculator

Suppose that we want to simplify our calcBMI() function so that the caller need only specify whether or not the units of measurements are metric or not. This way, they do not need to know the constant multiplier in the formula, $c * wt/ht^2$. If the units are English, we assume weight is measured in pounds and height in inches, and if metric, the units are kilograms and meters. Our new function definition has a *metric* argument with a default value of FALSE and we no longer need the *unitConstant* argument. Now, we check the value of *metric* and set *unitConstant* to either 703 (when FALSE) or 1 (when TRUE). Our revised function appears as

```
calcBMI =
  function(ht, wt, metric = FALSE)
{
  if (metric) {
    unitConstant = 1
  } else {
    unitConstant = 703
  unitConstant * wt / ht^2
}
```

We call this revised function with, e.g.,

```
calcBMI(1.73, 72, TRUE)
```

```
[1] 24.06
```

```
calcBMI(68, 160)
```

```
[1] 24.33
```

A more advance function definition keeps *unitConstant* as a formal argument and sets its value based on the *metric* argument. That is, consider the function signature,

```
function(ht, wt, metric = FALSE,
         unitConstant = if (metric) 1 else 703)
```

Here we have placed an if-else statement in the function's signautre. A somewhat unusual feature of the if-else statement is that it is an expression and so returns a value. We have assigned the result of the if-else statement as the default value of *unitConstant*. This value is 1 or 703 and is assigned to *unitConstant* (recall from Section 5.4 that this assignment takes place when *unitConstant* is first used in the function's code). Also note that since the code blocks in the if-else statement each contain one expression, we can drop the curly braces. This approach of using both the *metric* and the *unitConstant* parameters has the advantage of allowing the caller to specify a unitConstant other than 1 and 703 so we maintain the flexibility of the first version of the function and the simplicity of the second version where we don't need to know the specific value for the constant for standard calculations of BMI. In this case, the function body is the same as for the first function, i.e.,

```
calcBMI =
function(ht, wt, metric = FALSE,
         unitConstant = if (metric) 1 else 703)
{
  unitConstant * wt / ht^2
}
```

We confirm that the function still returns the same values as with the previous modification, i.e.,

```
calcBMI(1.73, 72, TRUE)
```

```
[1] 24.06
```

```
calcBMI(68, 160)
```

```
[1] 24.33
```

However, we can also call this function by supplying a value for *unitConstant* with

```
calcBMI(1.73, 72, unitConstant = 1)
```

```
[1] 24.06
```

```
calcBMI(68, 160, unitConstant = 703)
```

```
[1] 24.33
```

In these examples, the value of *metric* is not checked because *unitConstant* is supplied. As a further example, if weight is in pounds and height is in meters, then we can supply a constant appropriate for these units, i.e., `calcBMI(1.727, 160, unitConstant = 0.4535)`. The result is 24.33; this is the same as `calcBMI(68, 160)` because 1.727 meters is 68 inches and 0.4535 is the constant needed when height is measured in meters and weight in pounds.

■

In the following sections, we explore the use of `if-else` statements in greater detail.

5.6.1 Conditionally Modifying Inputs to log(0)

We revisit the problem that we addressed in Section 5.2.5 of using a log transformation when some values are 0 . The logPlus() function we created there had two inputs *x*, a vector of the values to be log-transformed, and *amt*, the amount to add to *x* before taking the logarithm. Suppose that we want our function to add this small amount to *x* only if 0s are present in the data. This means that we want our function to perform different computations, depending on the data.

We can modify our function to conditionally compute `log(x + amt)` when `x` has 0s and `log(x)` when there are no 0s. The condition that we check can be expressed as, e.g., `any(x == 0)`, which is `TRUE` if `x` contains one or more 0s and `FALSE` otherwise. Our modified function appears as

```
logPlus =
function(x, amt = 1)
{
  if (any(x == 0)) {
    y = log(x + amt)
  } else {
    y = log(x)
  }
  return(y)
}
```

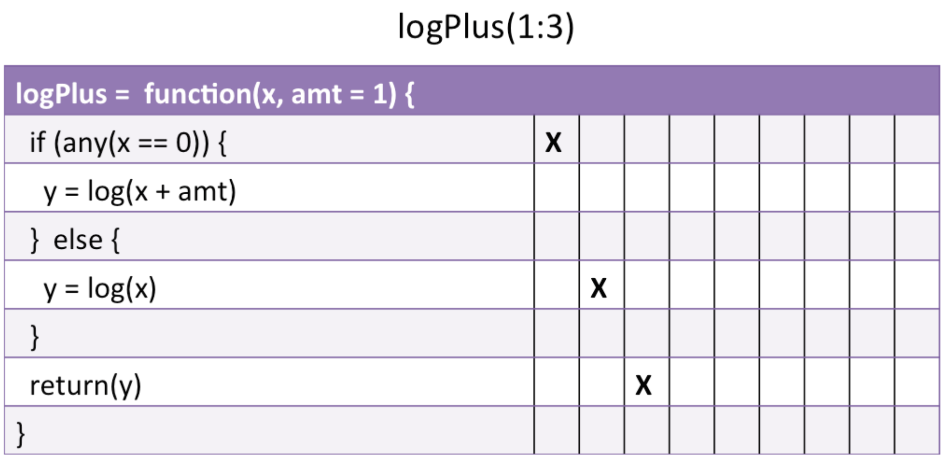
Notice that we have created a variable `y` to hold the transformed values and explicitly return `y` in the last statement of the function.

The flow diagram below provides the sequence of evaluation of the expressions. Each vertical column to the right of the code represents a step in the sequence. When we call `logPlus(0:3)`, the first expression evaluated is the condition in the `if` construct. Since this condition evaluates to `TRUE` the second statement evaluated is `y = log(x + amt)`. Then, the `else` construct is skipped, and the third statement evaluated is the return.

logPlus(0:3)

logPlus = function(x, amt = 1) {									
if (any(x == 0)) {	X								
y = log(x + amt)		X							
} else {									
y = log(x)									
}									
return(y)			X						
}									

On the other hand, when we call the function with `logPlus(1:3)`, then we have a different code flow, as shown in the diagram below. The first statement evaluated is the condition in the `if` construct, and since this is `FALSE`, the statement in the `if` block is skipped. In this invocation, the second statement evaluated is in the `else` block, and lastly, the call to return() is made.



logPlus(1:3)

logPlus = function(x, amt = 1) {									
if (any(x == 0)) {	X								
y = log(x + amt)									
} else {									
y = log(x)		X							
}									
return(y)			X						
}									

The else construct is not always needed and so is optional. To demonstrate, suppose that we want to throw an exception if the function is passed any negative values. Then we can modify logPlus() to check if x has any negative values and, if so, exit the function without taking the logarithm of the data. We augment our function with an additional if statement to handle this situation, e.g.,

```
logPlus =
function(x, amt = 1)
{
  if (any(x < 0)) {
    stop("Negative values in x")
  }
  if (any(x == 0)) {
    y = log(x + amt)
  } else {
    y = log(x)
  }
  return(y)
}
```

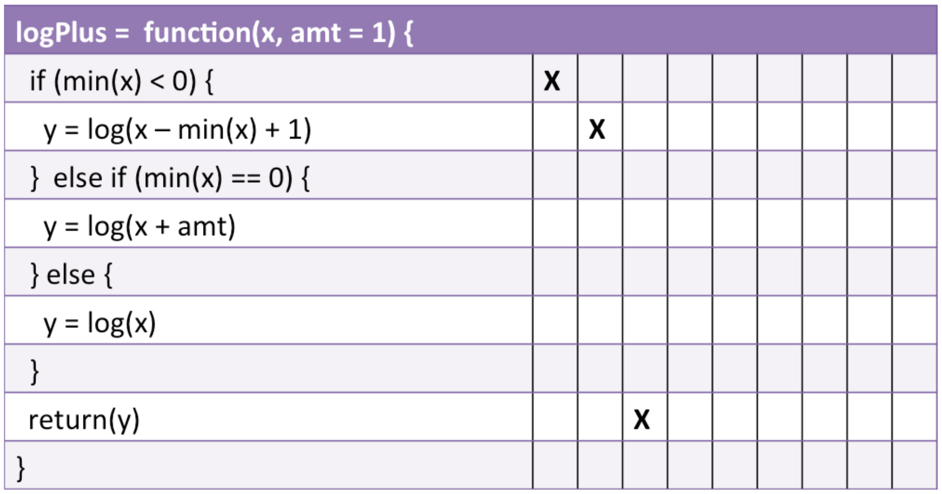
When the stop() function is called, execution is halted, the function is exited and the text supplied to stop() is printed to the console. We do not need to include an else here because the subsequent expressions are evaluated only when all data values are nonnegative.

We continue this example to demonstrate how to check multiple conditions in nested if-else statements. Let's suppose that instead of stopping the function when we find negative values in x, we shift all of the data by $1 - min(x)$ and then take logs. Now we have 3 possible return values, and our code appears as

```
logPlus =
function(x, amt = 1)
{
  if (min(x) < 0) {
    y = log(x - min(x) + 1)
  } else if (min(x) == 0) {
    y = log(x + amt)
  } else {
    y = log(x)
  }
  return(y)
}
```

Below is an example of the sequence of evaluation of the expressions when we call this version of logPlus() with `-1:3` as input.

logPlus(-1:3)

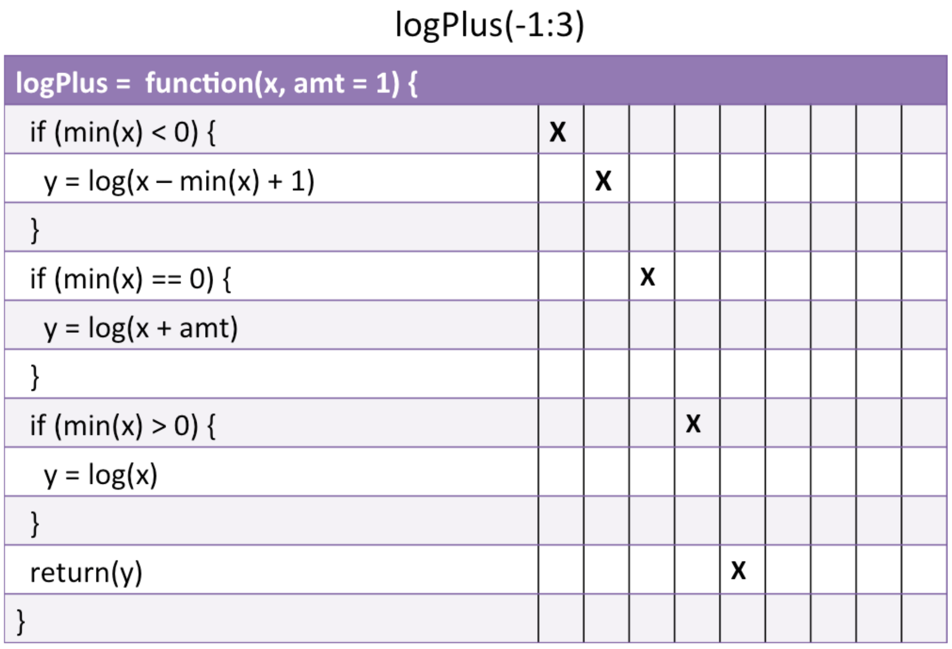


logPlus = function(x, amt = 1) {									
if (min(x) < 0) {	X								
y = log(x – min(x) + 1)		X							
} else if (min(x) == 0) {									
y = log(x + amt)									
} else {									
y = log(x)									
}									
return(y)			X						
}									

Lastly, we can modify this code to eliminate all of the `else`s, e.g.,

```
logPlus =
function(x, amt = 1)
{
  if (min(x) < 0) {
    y = log(x - min(x) + 1)
  }
  if (min(x) == 0) {
    y = log(x + amt)
  }
  if (min(x) > 0) {
    y = log(x)
  }
  return(y)
}
```

This version of the function correctly computes the 'logarithm' of the data, but it always checks all three conditions. To see this, compare the flow diagrams for this version of the function to the previous version (again when we call the function with `-1:3` as input). Note that this version evaluates 2 extra `if` conditions.



logPlus(-1:3)

logPlus = function(x, amt = 1) {									
if (min(x) < 0) {	X								
y = log(x – min(x) + 1)		X							
}									
if (min(x) == 0) {			X						
y = log(x + amt)									
}									
if (min(x) > 0) {				X					
y = log(x)									
}									
return(y)					X				
}									

This latest version of the function, the one without any `else`s, is less efficient than the version with the nested `if-else` statements because the code must always check all three conditions. Another inefficiency comes from the computation of `min(x)` repeatedly. We can address this re-computation of `min(x)` by assigning it to a variable at the beginning of the function. We can also return from the function in multiple places to avoid checking all 3 conditions. The issue of efficiency is covered in more detail in [?]. We incorporate the computation of the minimum once and the use of the return() function in the `if` code blocks to obtain the following revision of our function:

```
logPlus =
function(x, amt = 1)
{
  m = min(x)

  if (m < 0) return(log(x - m + 1))
  if (m == 0) return(log(x + amt))
  return(log(x))
}
```

Note that the curly braces have been eliminated because these blocks of code contain only one statement, and the condition and associated expression are collapsed onto one line of code. See Section 5.8 for a further discussion of the style guidelines for writing `if-else` statements.

The conditions in an `if-else` statement are single logical values so we often have to map a logical vector of length n to one of length 1 for our condition. The functions all() and any() are often used for this purpose. We already have seen examples of any() in a condition, i.e.,

```
if (any(x < 0)) {
  stop("Negative values in x")
}
```

A similar condition that uses all() is

```
if(all(x > 0)) {
  y = log(x)
}
```

The ifelse() function

The ifelse() function is essentially an element-wise version of `if-else`. As just mentioned, the condition in an `if-else` statement must be a single logical value. If we want to iterate over several elements in a logical vector and do something for each depending on whether it is `TRUE` or `FALSE`, then we can use the ifelse() function as a convenience function. For a simple example, consider again the issue of taking the logarithm of a vector of numeric values. Suppose we want to return `NA` for any negative values. The ifelse() function assists us here, i.e.,

```
log(ifelse(x >= 0,  x, NA))
```

```
[1]    NA -1.33  0.36  0.98 -0.33 -0.57
```

We have suppressed the warning and the first value is `NA` rather than `NaN`.

The first argument of the ifelse() function is a logical vector. The other two inputs should be vectors of the same length as this logical vector. That is, we have 3 vectors of the same length. If the i-th element in the condition vector is `TRUE`, then the i-th element of the 2nd vector (the *yes* argument) is assigned to the i-th element of the result. Otherwise, the i-th element of the result is taken from the 3rd vector, i.e., the *no* argument. Notice that in our example, the 3rd input vector is of length 1 so this value is recycled to create a vector of 6 `NA`s to match the length of the logical vector.

Alternatively, we can carry out this vectorized version of the `if-else` with subsetting, e.g.,

```
condition = x >= 0
ans = numeric(length(condition))
ans[condition] = log(x[condition])
ans[!condition] = NA
ans
```

```
[1]    NA -1.33  0.36  0.98 -0.33 -0.57
```

Here, we created a logical vector (`condition`) from the expression: `x >= 0`. Then, we create a vector to hold the results (`ans`). This vector is the same length as our condition and by default consists of 0s. We populate `ans` with values from `log(x)` or with `NA`, depending on the corresponding values in `condition`. That is, the 1st element of `ans` is set to the 1st element of `log(x)` when the 1st element of `condition` is `TRUE`, the 2nd element of `ans` is set to the 2nd element of `log(x)` when the 2nd element of `condition` is `TRUE`, and so on. Similarly, in the next statement, we use `!condition` to place `NA`s in all elements in `ans` where `x` is negative. More generally, the ifelse() function is equivalent to

```
 ans = numeric(length(test))
 ans[test] = yes[test]
 ans[!test] = no[!test]
```

Here *test*, *yes* and *no* are equivalent to the 3 vectors of inputs to ifelse(). Lastly, we note that the result from ifelse() can be a matrix or array (but of course these are still vectors).

5.6.2 Converting Liquid Measures into Tbs. Using switch()

Recall from Q.5-4 (page 195) that our goal is to write a function to convert liquid measures into tablespoons. Specifically, the caller supplies the amount and units of the liquid, and the function returns the equivalent number of tablespoons. We have enough information to specify our function signature as

```
convertLiquid = function(x, units = "ml")
```

Here, we require the amount, which is specified in x and we provide a default value ("ml") for the units via the *units* argument.

We can carry out the conversion via multiple nested if-else statements, i.e.,

```
if (units == "ml") {
  y = x * 0.067628
} else if (units == "l") {
  y = x * 67.628
} else if (units == "oz") {
  y = x * 2
} else if (units == "tsp") {
  y = x / 3
} else if (units == "tbs") {
  y = x
} else {
  warning(paste("Unrecognized units: ", units, "\n"))
  y = NA
}
```

Notice that in the final else block we issue a warning that our function does not recognize the units specified by the caller. We provide a value for the conversion that is NA and do not stop the execution.

We can encapsulate this code into our convertLiquid() function, but instead, we use the more concise switch() function to check the various options for the input units of measurement. We do this with

```
convertLiquid = function(x, units = "ml") {
  units = tolower(units)
  y = switch(units,
             milliliter = , ml = x * 0.067628,
             liter = , l = x * 67.628,
             ounce = , oz = x * 2,
             tablespoon = , tbs = x,
             teaspoon = , tsp = x / 3,
             NA)
  if (any(is.na(y)))
    warning(paste("Unrecognized units: ", units, "\n"))
  return(y)
}
```

We have made our convertLiquid() function more robust by converting *units* to lower case before checking its value. We also made the function more robust by adding more options to switch() that handle the case when the caller supplies the unabbreviated name for the units. For these options, we do not supply an expression, which means the next option that has an expression is evaluated so we have paired *milliliter* with *ml*, *liter* with *l*, and so on.

The switch() function
The switch() function is another type of branching statement. This function lets us use the value of a variable to identify which one of many different alternative expressions to evaluate. It can be more convenient than multiple `if-else` statements, when we can identify the expression to evaluate by the value of the condition itself. More specifically, switch() works in 2 distinct ways depending on whether the expression provided in the first argument evaluates to a character string or number. When it's a string, switch() then looks for a named argument that matches that value, e.g., when a string is "ml" then the expression assigned to the argument *ml* is evaluated and returned. If none of the argument names match and there is an unnamed argument, then its value is returned. In our example above, we return `NA` for the conversion amount when none of the argument names match. When a value for an argument is not supplied, e.g., `liter = ,` then the next non-missing element is evaluated.

5.7 Iterative Evaluation of Code

Much of what is handled using loops in other languages can be more efficiently carried out in *R* with vectorized computations or using one of the apply functions. However, there are situations where we need looping, such as with algorithms that require recursion. There are two main looping constructs in *R*: the `for` and the `while` loops. The `for` loop is more common so we introduce it first.

For Loops
The basic format of a `for` loop is:

```
for (var in vector) {
  statements
}
```

Here, the variable `var` is set to the 1st value in `vector` and the code block between the curly braces is evaluated. Then, `var` is set to the 2nd value in `vector` and the code block is evaluated again, and so on for each element in `vector`. The `vector` often contains integers, but it can be any valid type. If `vector` is a variable, any changes to it within the loop do not affect the values of `var` in the looping. Also, if `vector` has length 0, then the loop is skipped.

5.7.1 Reading Thousands of Files of Data With a `for` Loop

Recall from Q.4-4 (page 153) that our goal is to read many files into *R*. There are so many files that we cannot possibly type the file names at the console, or in a script. Instead, our goal is to programmatically acquire the file names and read the file contents. After downloading and unzipping the data, we found that all of the files are within one directory (folder) called *loans*. Our first step is to capture all of the file names in a character vector; we can do this with the list.files() function in *R*. Suppose the folder name is a string in `dirName`. Then we obtain the names of all of the files with

```
fileNames = list.files(dirName, full.names = TRUE)
length(fileNames)
```

```
[1] 1975
```

The *full.names* argument indicates that we want the full path names from the top level directory to the file. We have nearly 2000 file names in `fileNames`.

To read any particular file, we can simply call, say, fromJson() and pass the name of one of the files in our character vector, e.g.,

```
x = fromJSON(fileNames[7])
```

We want to generalize this code so that we read all the files. What kind of data structure should we use to hold all of these files? We can use a list. That is, we can loop over the file names, reading in each file and adding it to a list, e.g.,

```
rawData = vector("list", length = length(fileNames))
for (i in 1:length(fileNames)) {
  rawData[[i]] = fromJSON(fileNames[i])
}
```

Notice that our first step is to create a list structure that is large enough to hold all of the raw data. Then, our `for` loop reads each file, one at a time, and puts the result in the appropriate element of `rawData`.

We mentioned that the index in the `for` loop need not be integer values. For example, we can use the file names themselves as our index, i.e.,

```
rawData = vector("list", length = length(fileNames))
names(rawData) = fileNames
for (oneName in fileNames) {
  rawData[[oneName]] = fromJSON(oneName)
}
```

Here, we use the file names as names for the elements in our list. The `for` loop then cycles through these names, i.e., `oneName` takes on the first file name, this file is read into *R* and assigned to the element in the list of that name, then `oneName` takes on the 2nd file name, this file is read into *R* and assigned to the corresponding element in the list, and so on.

Hopefully, the thought has crossed your mind that this seems very much like what lapply() does. Recall that the apply functions such as lapply() applies a function to each element of a list. We can also supply a character vector to lapply(), e.g.,

```
rawData = lapply(fileNames, fromJSON)
```

This one line of code is simpler and focusses on the essence of the computation, i.e., to apply fromJSON() to each element of `fileNames`. Notice that with this approach, we do not need to create `rawData` in advance. This approach to looping with lapply() often can be faster than using the `for` loop.

Those familiar with for loops in other languages often lean toward using the `for`-loop construct rather than a vector calculation. For example, rather than compute BMI with `bmi = 703 * wt / ht^2`, we can use a for loop, e.g.

```
n = length(ht)
bmi = vector("numeric", length = n)
for (i in 1:n) {
  bmi[i] = 703 * wt[i] / ht[i]^2
}
```

We advise strongly against this approach. Ignoring the vectorized nature of *R* can significantly impact the speed of execution and should be avoided.

The `for` loop is a more appropriate construct, when one iteration of the code block depends on a previous iteration. We provide an example in the next section.

5.7.2 Generating Fibonacci Numbers

We saw in Q.5-6 (page 195) that Fibonacci numbers can be defined recursively, for $n > 1$, by

$$F_n = F_{n-1} + F_{n-2}$$

Additionally, $F_0 = 0$ and $F_1 = 1$.

One way to generate, say, F_6 is to begin with F_0 and F_1, add them together to get F_2, and continue adding values together until we reach F_6. For example, we can begin by creating a numeric vector to hold the 7 Fibonacci numbers (recall that we start with F_0) and setting the first two values with

```
fibNums = vector("numeric", length = 7)
fibNums[1]= 0
fibNums[2] = 1
```

Then, we compute each subsequent Fibonacci number from the previous two with

```
fibNums[3] = fibNums[1] + fibNums[2]
fibNums[4] = fibNums[2] + fibNums[3]
fibNums[5] = fibNums[3] + fibNums[4]
fibNums[6] = fibNums[4] + fibNums[5]
fibNums[7] = fibNums[5] + fibNums[6]
fibNums[7]
```

```
[1] 8
```

We can replace each of the computations for the 2nd through 7th Fibonacci numbers with a `for` loop, e.g.,

```
for (i in 2:6) {
  fibNums[i + 1] = fibNums[i] + fibNums[i - 1]
}
```

Notice that the `vector` in our loop consists of the integers $2, 3, \ldots, 6$ and the indexing variable `i` is used to populate the 3rd through 7th elements of `fibNums`. This offset is because of 0-based counting in the series, i.e., the first element in `fibNums` is F_0.

Now that we have a vector of the 1st 7 Fibonacci numbers, we can abstract this code so that it works for arbitrary Fibonacci numbers and encapsulate it into a function. We leave this as an exercise.

As an alternative approach, notice that we find a particular Fibonacci number from the previous 2 numbers. That is, if `fib.1.bak` is the previous Fibonacci number and `fib.2.bak` is the number before that, then the current Fibonacci number is `fib = fib.1.bak + fib.2.bak`. For example, to find F_2, we start with the first 2 Fibonacci numbers, which in this case are the previous 2 Fibonacci numbers, and add them together, e.g.,

```
fib.2.bak = 0
fib.1.bak = 1
fib = fib.1.bak + fib.2.bak
fib
```

```
[1] 1
```

Can we use `fib.1.bak`, `fib.2.bak`, and `fib` to compute the next Fibonacci number? We can update `fib.1.bak` and `fib.2.bak` as follows:

```
fib.2.bak = fib.1.bak
fib.1.bak = fib
```

Then, we can compute the next Fibonacci number with

```
fib = fib.1.bak + fib.2.bak
fib
```

```
[1] 2
```

Notice that we are careful with the order of the re-assignments of `fib.1.bak` and `fib.2.bak`. If we switch the order, then both `fib.1.bak` and `fib.2.bak` have the same value (`fib`), and we do not get the correct result.

If we repeat these 3 steps, we eventually get the desired number in the sequence. That is,

```
fib.2.bak = fib.1.bak
fib.1.bak = fib
fib = fib.1.bak + fib.2.bak
fib.2.bak = fib.1.bak
fib.1.bak = fib
fib = fib.1.bak + fib.2.bak
fib.2.bak = fib.1.bak
fib.1.bak = fib
fib = fib.1.bak + fib.2.bak
fib
```

```
[1] 8
```

This is the same value that we obtained earlier, but we did not keep all of the intermediate Fibonacci numbers.

Again, we can write a `for` loop to carry out this calculation, rather than repeat these 3 lines of code the required number of times. Once the initial 2 values of the Fibonacci sequence are defined, (i.e., `fib.1.bak = 1` and `fib.2.bak = 0`), then we can compute any number with the following loop:

```
for (i in 2:n) {
  fib = fib.1.bak + fib.2.bak
  fib.2.bak = fib.1.bak
  fib.1.bak = fib
}
```