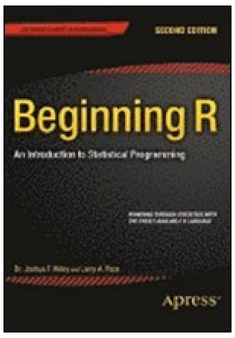

# Beginning R: An Introduction to Statistical Programming, Second Edition

by Joshua F. Wiley and Larry A. Pace

# Chapter 4: Control Structures

## Overview

In Chapter 3, you saw an example of a simple function for calculating a person's body mass index (BMI). The function used logic to determine the person's risk category. The use of logic, conditional statements, and loops are all inherent in controlling the flow of a program. Although there are many different ways to write conditional statements, and loops, too, for that matter, they all boil down to testing to see if some condition is true or false and then behaving accordingly. A program is nothing more or less than a series of instructions telling a computer what to do. As I have mentioned, R is both functional and objected-oriented, so every R program consists of function calls that operate on objects.

## 4.1 Using Logic

We have spoken frequently about the use of logic. R provides several different ways to select which statement or statements to perform based on the result of a logical test.

You saw the comparison operators in R in Table 1-2. There are also binary logical operators for Boolean expressions in R (see Table 4-1).

**Table 4-1: Binary Operators in R**

| Operator | R Expression | Explanation |
|---|---|---|
| NOT | ! | Logical negation |
| Boolean AND | & | Element-wise |
| Boolean AND | && | First element only |
| Boolean OR | \| | Element-wise |
| Boolean OR | \|\| | First element only |

As you can see, the shorter form acts in a fashion similar to the arithmetic operators, proceeding element-wise along the vector. The longer form evaluates left to right, and examines only the first element of each vector. Examples make these easier to understand. See that the shorter form of `OR` evaluates the truth of either the elements of `x` being less than the elements of `y OR` of the elements of `x` being less than 10. All of these conditions are true. But when we use the longer form, it evaluates only the first element in each vector.

Similarly, the shorter version of `AND` function evaluates to `FALSE` because on the last comparison, `x < y AND x < 10` are not both true:

```
> x <- 1:10
> y <- 11:15
> x < y | x < 10
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> x < y || x < 10
[1] TRUE
> x < y & x < 10
[1]  TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE
> x < 6 && x < 10
[1] TRUE
```

## 4.2 Flow Control

The programmer must control the flow from one part of the program to another. Flow control involves the use of loops, conditional statements and branching, and conditions that cause the program to stop doing something and do something else or to quit entirely. Virtually all computer programming languages implement these capabilities, but often in very different ways. We will discuss these concepts in the abstract briefly, but then will illustrate them with very specific examples. As you will see, with R, as with most other languages, there is often more than one way to accomplish the same purpose. Some of these ways are more efficient and effective than others, whereas others are not as efficient or effective. Our goal will be to learn how to program in such a way that we can write efficient, reproducible code. We will cover looping and conditional statements and branching before we move into a more general discussion of programming in R.

### 4.2.1 Explicit Looping

R provides three types of *`explicit`* loops. You can use one type of looping to accomplish the same purpose as the other types, but the most commonly used loop in R is the `for` loop. We also have `while` loops and `repeat` loops. Although most R coders will typically use `for` loops, we will also illustrate `while` and `repeat` loops, which could work better for some given purpose, especially when iteration through a data structure is not the primary objective.

Let's start with a very simple example. We want to multiply 1 through 5 by 10 and print the results to the R console. Although this is a rather uninteresting example, it shows the syntax of a `for` loop, which is `for(names in values) expression`. Here's our code and result:

```
> x <- 1:5
> for (i in x) print (i * 10)
```

```
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
```

We defined `x` as the integers from 1 to 5, and wrote the `for` statement in such a way that the names attribute `i` iterated through the values in `x` one at a time, while the expression `print(i * 10)` printed the result of multiplying each successive value in the `x` vector by 10. Not very fancy, but it worked. Of course, the loop we just wrote was also totally unnecessary, as we could simply ask R to print the results as follows, using implicit looping (vectorization) rather than explicit looping:

```
> x <- 1:5
> x * 10
[1] 10 20 30 40 50
```

We can also use `for` loops with lists and vectors. For example, we may have a character vector such as the following, and want to loop through it:

```
> shoppingBasket <- c("apples", "bananas", "butter", "bread", "milk", "cheese")
> for(item in shoppingBasket) {
+ print(item)
+ }
[1] " apples "
[1] " bananas "
[1] " butter "
[1] " bread "
[1] " milk "
[1] " cheese "
```

Once again, while possible, this loop is also unnecessary, as we can simply print the shopping list without the need for looping. As users who come to R from languages that are not vectorized learn about vectorization and especially about a family of implicit looping functions called `apply`, such users learn that implicit looping can be accomplished across not just vectors or lists but also with matrices and even entire data frames.

The `while` loop will continue to execute until a stopping condition is met. For example, we can write a loop such as the following, which will accomplish the same results as our `for` loop. The `while` loop required a few more lines of code. Thus, it is less efficient than the `for` loop. We specified a counter and iterated the counter by 1 until the stopping condition in the `while` statement was met. Some languages begin vector indexing with 0 rather than 1, while R begins with 1. This is the reason for setting the counter to 0 initially. As you will see in Chapter 5, there are some situations in which `while` loops have potential advantages over `for` loops:

```
> count <- 0
> end <- 5
> while (count < end ) {
+   count <- count + 1
+   print(count * 10)
+ }
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
```

The `repeat` loop is similar to the `while` loop. Just as with a `while` loop, the statement or statements after the `repeat` are executed until a stopping constraint is met. However, the only way to exit a repeat loop is with a `break` statement. For this reason, it is quite possible (voice of experience speaking here) to write `repeat` loops that keep repeating infinitely if one is not careful. Here is a simple example of a `repeat` loop:

```
> total <- 0
> repeat {
+   total <- total + 1
+   print (total*10)
+   if (total == 5)
+     break
+ }
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
```

### 4.2.2 Implicit Looping

The `apply` function family in R provides users implicit looping capabilities. You should be aware that the `apply` function, contrary to the beliefs of many, does not result in speed advantages over explicit loops. In fact, if you simply type apply and press `<Enter>` to see the actual `apply` function, you will observe loops in that function definition itself. The primary advantage of such "higher-order" functions as the `apply` family is the improvement of the clarity or the objective(s) of one's code. Let's examine several of the most commonly used `apply` functions–there are additional ones that we will not cover in this basic text. The basic `apply` function applies a function over the margins of an array. The "margins" in this context are either the rows (1), the columns (2), or both (1:2). Let's delve deeper

The apply function can be used to calculate the marginal (row or column) means or sums, as shown here, but the newer `rowMeans, colMeans, rowSums`, and `colSums` functions are designed for speed. Recall the ten-word vocabulary test from the GSS we used in Chapter 2. Because the test was not administered every year, there were many missing cases. What if for some reason we wanted to do an analysis on the individual participants' scores, which are simply the total of the numbers of words the respondent defined correctly? We could use the `apply` function for this purpose. However, just to get rid of those annoying NAs, let's choose only the complete cases, that is, those in which there are no NAs (missing values). We can use the `complete.cases` function as follows:

```
> library(foreign)
> gss2012 <- read.spss("GSS2012merged_R5.sav", to.data.frame = TRUE)
> myWords <- paste0 ("word", letters [1:10])
> vocabTest <- gss2012 [ myWords ]

> vocab <- vocabTest[complete.cases(vocabTest) ,]

> #Our download of GSS2012 has CORRECT & INCORRECT. We convert those to 1 and 0 respectively.
> wordsToNum<-function(x) {
+ as.integer(x=="CORRECT")
+ }
>
> vocab<-apply(vocab,2,wordsToNum)

#apply gives back a matrix, we can turn this back into a data frame > vocab
<- as.data.frame(vocab)
> head(vocab, 3)
  worda wordb wordc wordd worde wordf wordg wordh wordi wordj
1     1     1     0     1     1     1     0     0     1     1
2     1     1     1     1     1     1     1     1     1     0
3     0     1     0     1     1     0     0     0     0     0
```

From the entire GSS dataset, there are 2,494 individuals who responded to the vocabulary words. Let's compare the apply function with the `colSums` function by using the `rbenchmark` package, which is a convenient wrapper around `system.time`. By default, it will perform 100 replications of the expression being tested, which should be sufficient for our purposes of comparing the speed of the two functions. The difference in elapsed times is impressive, with the `colSums` function producing almost a fivefold improvement. Sometimes a significant speed improvement is as simple as using a newer function designed for speed. There are also other ways to achieve speed increases, as we will discuss later:

```
> install.packages("rbenchmark")
> library(rbenchmark)
> benchmark(apply(vocab, 2, sum))
                  test replications elapsed relative user.self sys.self user.child sys.child
1 apply(vocab, 2, sum)          100    0.43        1      0.42        0         NA        NA
> benchmark(colSums(vocab))
                  test replications elapsed relative user.self sys.self user.child sys.child
1 colSums(vocab)                100    0.09        1       0.1        0         NA        NA
```

The sapply function applies a function to the elements in a list but returns the results in a vector, matrix, or a list. If the argument `simplify = FALSE, sapply` will return its results in a list, just as does the lapply function. If simplify = TRUE (which is the default), then `sapply` will return a simplified form of the results if that is possible. If the results are all single values, `sapply` will return a vector. If the results are all of the same length, `sapply` will return a matrix with a column for each element in the list to which the function was applied. Let's illustrate these two functions again, each of which avoids explicit looping. We will use an example similar to our previous one, in which we have three numeric vectors of different lengths combined into a list and then use the `sapply` and `lapply` functions to find the means for all three vectors:

```
> x1
[1] 55 47 51 54 53 47 45 44 46 50
> x2
[1] 56 57 63 69 60 57 68 57 58 56 67 56
> x3
[1] 68 63 60 71 76 67 67 78 69 69 66 63 78 72
> tests <- list (x1 , x2 , x3)
> tests
[[1]]
[1] 55 47 51 54 53 47 45 44 46 50

[[2]]
```

```
[1] 56 57 63 69 60 57 68 57 58 56 67 56

[[3]]
[1] 68 63 60 71 76 67 67 78 69 69 66 63 78 72
> sapply (tests , mean )
[1] 49.20000 60.33333 69.07143
> lapply (tests , mean )
[[1]]
[1] 49.2

[[2]]
[1] 60.33333

[[3]]
[1] 69.07143
```

Another very useful function is the `tapply` function, which applies a function to each cell of a ragged array. To demonstrate this function, assume that we have 20 students who have each studied either alone or in a study group, and each of whom was taught a different study method. The scores for the 20 students on a quiz may look like the following, with the factors as shown:

```
> QuizScores
   score approach group
1     80  method1 alone
2     80  method2 alone
3     76  method1 alone
4     79  method2 alone
5     80  method1 alone
6     81  method2 alone
7     83  method1 alone
8     72  method2 alone
9     83  method1 alone
10    77  method2 alone
11    73  method1 group
12    69  method2 group
13    66  method1 group
14    67  method2 group
15    70  method1 group
16    72  method2 group
17    76  method1 group
18    71  method2 group
19    71  method1 group
20    72  method2 group
```

We can use the tapply function to summarize the scores by the method used and by whether the student studied alone or in a group. We can also combine the two factors and find the means for each of the cells of a two-way table. To make our code a little easier to understand, let's attach the QuizScores data frame so that we can refer to the scores and the two factors directly. Without doing any statistical analyses, it appears that studying alone using method 1 may be a good strategy for maximizing one's quiz score:

```
> attach(QuizScores)
> tapply(score, approach, mean)
method1 method2
   75.8    74.0
> tapply(score, group, mean)
alone group
 79.1  70.7
> tapply (score, list(approach, group), mean)
         alone group
method1   80.4  71.2
method2   77.8  70.2
```

It's worth mentioning here that the `aggregate` function can be used to apply a function such as the mean to a combination of factors as well. For the quiz data, we can get a convenient summary by using `aggregate` as follows. We supply a "formula"–in this case, the score by the two factors–identify the data frame, and then identify the function. We get a slightly differently formatted result from that of tapply, although the results are identical:

```
> aggregate ( score ~ approach + group, data = QuizScores, mean)
  approach group score
1  method1 alone  80.4
2  method2 alone  77.8
3  method1 group  71.2
4  method2 group  70.2
```

## 4.3 If, If-Else, and ifelse( ) Statements

In addition to looping, we can use various forms of `if` statements to control the flow of our programs. The `if` statement is the simplest. `If` evaluates simply as:

```
if (condition) expression
```

The expression may be a single line of code, or it may be multiple statements (what the R documentation refers to as a compound statement) enclosed in braces. If the condition evaluates to TRUE, the expression is performed, and if the conditions evaluates to FALSE, the expression is not performed.

We can also use an if-else statement, which evaluates as follows. Expression1 is performed if the condition is `TRUE`, and expression2 is performed if the condition is `FALSE:`

```
if (condition) expression1 else expression2
```

There is also an `ifelse` function, which takes the form `ifelse(test, yes, no)`. The test is the logical condition, "yes" is what is returned if the logical condition is TRUE and "no" is what is returned when the logical condition is false. As usual, examples make the use of these statements clearer.

An `if` statement leads to the evaluation of the expression if the condition is TRUE. Otherwise, the expression is not evaluated. For example, say we are conducting a hypothesis test with an alpha level of .05. We will reject the null hypothesis if the p-value is equal to or lower than .05, and will not reject the null hypothesis if the p-value is greater than .05. We can use an if statement to specify rejection of the null hypothesis, Note that in the second case, the condition is false because .051 is greater than .05, so nothing happens. We can make our function a little more flexible by specifying a default alpha level, which we can then change as necessary:

```
> rejectNull <- function(pValue, alpha = .05) {
+ if (pValue <= alpha) print("Reject Null")
+ }
> rejectNull(.05)
[1] "Reject Null"
> rejectNull (.051)
>
```

Adding an `else` condition covers both bases:

```
> rejectNull <- function(pValue, alpha = .05) {
+   if (pValue <= alpha) print("Reject Null")
+   else print ("Do not Reject")
+ }
> rejectNull (.07)
[1] "Do not Reject"
> rejectNull(.05)
[1] "Reject Null"
> rejectNull(.05, .01)
[1] "Do not Reject"
```

The `ifelse` function is quite interesting. For `if` and `else` statements, the condition must result in a single logical condition being `TRUE` or `FALSE`. If you supply a vector, only the first element will be evaluated. But the `ifelse` function works on vectors. Say we create a sequence of integers from $-5$ to $+5$ and then attempt to take the square roots of the elements of the vector. This will return `NaNs` for all the negative numbers and will give us a warning. We can avoid the warning, extract the square roots of the non-negative numbers, and define the square roots of the negative numbers as `NA` all at once by using `ifelse:`

```
> x <- -5:5
> x
[1] -5 -4 -3 -2 -1  0  1  2  3  4  5
> sqrt (x)
[1]       NaN      NaN      NaN     NaN     NaN 0.000000 1.000000 1.414214
[9]  1.732051 2.000000 2.236068
Warning message:
In sqrt(x) : NaNs produced

> sqrt(ifelse(x >= 0, x, NA))
[1]       NA       NA       NA       NA      NA 0.000000 1.000000 1.414214
[9] 1.732051 2.000000 2.236068
```