Basic Course on R: Programming Structures 1

Karl Brand* and Elizabeth Ribble†

28 Oct - 1 Nov 2019

Contents

*brandk@gmail.com
†emcclel3@msudenver.edu

1 `if()`, `else`, and `ifelse()` Statements

1.1 Conditional Execution Using `if()` and (Optionally) `else`

- Sometimes we'll want R to execute a statement only if a certain condition is met. This can be accomplished via the `if()` and (optionally) `else` statements:

```
if()    # Used to execute a statement only if the given condition
        # is met
else    # Used to specify an alternative statement to be executed
        # if the condition given in if() isn't met
```

- Such ***conditional execution*** commands have the forms:

```
if (condition) {
  statement1
}
```

and

```
if (condition) {
  statement1
  } else {
    statement2
  }
```

- In both cases above, if `condition` is `TRUE`, `statement1` is executed. If `condition` is `FALSE`, then in the first case nothing happens. In the second case, `statement2` is executed.
- Here's a simple example:

```
x <- 5
if (x < 10) {
  y <- 0
}
y

## [1] 0
```

- Here's another:

```
if (x >= 10) {
  y <- 1
  } else {
    y <- 0
}
y

## [1] 0
```

- There's actually an easier way to accomplish the above task:

```
y <- if(x >= 10) 1 else 0
y

## [1] 0
```

- When using such *conditional assignment* statements, in the absence of `else`, `if()` returns `NULL` if `condition` isn't met. So

```
y <- if(condition) 1
```

is equivalent to

```
y <- if(condition) 1 else NULL
```

- In the next example, `return()` is used to terminate a function call and return a value that depends on whether or not a condition is met:

```
mySign <- function(x) {
    if(x < 0) {
      return("Negative")
    } else {
        return("Non-negative")
    }
}
```

We get:

```
mySign(13)

## [1] "Non-negative"
```

1.2 Using `if()` and `else` with a Sequence of Statements

- `if()` and `else` can be used to conditionally execute whole *sequences* of statements, which we enclose in curly brackets { }. The general form of an `if()` command is:

```
if (condition) {
  statement11
  statement12
      .
      .
      .
  statement1q
}
```

which could be followed by `else` and another sequence of statements (in curly brackets) to be executed if `condition` isn't met.

1.3 Vectorized if-else: The `ifelse()` Function

- Sometimes we'll need to create a vector whose values depend on whether or not the values in another vector satisfies some condition. We use:

```
ifelse()    # Returns a vector whose values depend on whether or
            # not a given condition is met by the elements of
            # another vector
```

- `ifelse()` takes argument `test`, the condition to be met, `yes` the return value (or vector of values) when `test` is `TRUE`, and `no`, the return values (or vector of values) when `test` is `FALSE`.

- Here we convert the values in `ht` to "short" or "tall":

```
ht <- c(69, 71, 67, 66, 72, 71, 61, 65, 73, 70, 68, 74)
htCategory <- ifelse(ht > 69, yes = "tall", no = "short")
htCategory
```

```
##  [1] "short" "tall"  "short" "short" "tall"  "tall"  "short" "short"
##  [9] "tall"  "tall"  "short" "tall"
```

2 Looping

- ***Loops*** are used to ***iterate*** (repeat) an R statement (or set of statements). They're implemented in three ways, `for()`, `while()`, and `repeat`:

```
for()       # Repeat a set of statements a specified number of
            # times
while()     # Repeat a set of statements as long as a specified
            # condition is met
repeat      # Repeat a set of statements until a break command is
            # encountered
```

- Two other commands, `break` and `next`, are used, respectively, to terminate a loop's iterations and to skip ahead to the next iteration:

```
break       # Terminate a loop's iterations
next        # Skip ahead to the next iteration
```

- Here's an example in which each of the three loop types, `for()`, `while()`, and `repeat`, are used to perform a simple task, namely printing the numbers $1^2, 2^2, ..., 5^2$ to the screen:

```
for(i in 1:5) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

```
i <- 1
while(i <= 5) {
  print(i^2)
  i <- i + 1
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

```
i <- 1
repeat {
  print(i^2)
  i <- i + 1
  if(i > 5) break
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

2.1 `for()` Loops

- `for()` loops are used when we know in advance how many iterations the loop should perform.
- The general form of a `for()` loop is:

```
for(i in sequence) {
  statement1
  statement2
      .
      .
      .
  statementq
}
```

where `sequence` is a vector, `i` (whose name you're free to change) assumes the values in `sequence` one after another, each time triggering another iteration of the loop during which `statements 1` through `q` are executed. The `statements` usually involve the variable `i`.

- Here's an example. Suppose we have the data frame describing someone's coin collection:

```
coins <- data.frame(Coin = c("penny", "quarter", "nickel",
                             "quarter", "dime", "penny"),
                    Year = c(1943, 1905, 1889, 1960, 1937, 1900),
                    Mint = c("Den", "SF", "Phil", "Den", "SF",
                             "Den"),
                    Condition = c("good", "fair", "excellent",
                                  "good", "poor", "good"),
                    Value = c(12.00, 55.00, 300.00, 40.00, 18.00,
                              28.00),
                    Price = c(15.00, 45.00, 375.00, 25.00, 20.00,
                              20.00))
coins
```

```
##      Coin Year Mint Condition Value Price
## 1   penny 1943  Den      good    12    15
## 2 quarter 1905   SF      fair    55    45
## 3  nickel 1889 Phil excellent   300   375
## 4 quarter 1960  Den      good    40    25
## 5    dime 1937   SF      poor    18    20
## 6   penny 1900  Den      good    28    20
```

- If we type:

```
colMeans(coins)
```

```
## Error in colMeans(coins):  'x' must be numeric
```

  we get an error message because some of the columns are non-numeric.

- We can compute the means of the numeric columns by looping over the columns, each time checking whether it's numeric before computing it's mean:

```
means <- NULL
for(i in 1:ncol(coins)) {
  if (is.numeric(coins[ , i])) {
    means <- c(means, mean(coins[ , i]))
    }
}
```

The result is:

```
means

## [1] 1922.33333   75.50000   83.33333
```

2.2 `while()` Loops

- `while()` loops are used when we want the loop to iterate until some condition is no longer met.
- The general form of a `while()` loop is:

```
while(condition) {
  statement1
  statement2
      .
      .
      .
  statementq
}
```

where `condition` is a logical (`TRUE` or `FALSE`) expression involving a variable whose value changes over the course of the loop iterations.

- Prior to each iteration, R checks whether `condition` is `TRUE` or `FALSE`. If it's `TRUE`, the iteration proceeds, otherwise the iterations are terminated.

2.3 `repeat` Loops

- A `repeat` loop iterates a set of statements until a `break` statement is encountered. The general form is of a `repeat` loop is:

```
repeat {
  statement1
  statement2
      .
      .
      .
  statementq
}
```

where at least one of the `statements` should be of the form

```
if(condition) break
```

where `condition` is a logical (`TRUE` or `FALSE`) expression which may be updated during the loop's iterations.

2.3.1 Terminating an "Endless" Loop

- Once in a while we (mistakenly) write a loop that has no way of stopping, for example:

```
i <- 1
while(i <= 5) {
  print("I Cannot Stop by Myself")
}
```

- To terminate the iterations hit the **Escape** key or select **Terminate R...** in RStudio's **Session** pulldown menu.

2.4 Looping Over List Elements

- In the next example, we loop over the elements of a list, printing a list element and recording it's length during each iteration:

```
myList <- list(
      w = c(4, 4, 5, 5, 6, 6),
      x = c("a", "b", "c"),
      y = c(5, 10, 15),
      z = c("r", "s", "t", "u", "v")
)
lengths <- NULL
for(i in myList) {
  print(i)
  lengths <- c(lengths, length(i))
}
```

```
## [1] 4 4 5 5 6 6
## [1] "a" "b" "c"
## [1]  5 10 15
## [1] "r" "s" "t" "u" "v"
```

```
lengths

## [1] 6 3 3 5
```

3 The Logical Operations "And", "Or", and "Not"

3.1 Logical Operations and Compound Logical Expressions

- R has ***logical operators*** (or ***Boolean operators***) corresponding to "and" and "or". They're used to combine two logical expressions together to form a single ***compound*** logical expression. Another logical operator corresponding to "not" is used to negate a logical expression. These are written in R as:

```
&&              # "And" for logical scalars
||              # "Or" for logical scalars
!               # "Not" (for logical scalars or vectors)
&               # "And" for logical vectors
|               # "Or" for logical vectors
```

- These operate on logical (`TRUE` or `FALSE`) expressions and return `TRUE` or `FALSE`.
- The distinction between `&&` and `&`, and between `||` and `|` is this:
  - `&&` and `||` operate on logical *scalars* (single `TRUE` or `FALSE` values).
  - `&&` and `||` are the preferred operators to use in `if()` statements.
  - `&` and `|` operate elementwise on logical *vectors*.
  - `&` and `|` are the preferred operators to use in `ifelse()` statements and in square brackets `[ ]` when extracting subsets from vectors or data frames.

3.2 Logical Operations on Scalar Logical Expressions

- `&&` returns `TRUE` if both of the expressions are `TRUE` and it returns `FALSE` otherwise:

```
TRUE && TRUE

## [1] TRUE

TRUE && FALSE

## [1] FALSE
```

- `||` returns `TRUE` if one or both of the expressions are `TRUE` and it returns `FALSE` otherwise:

```
FALSE || TRUE

## [1] TRUE

FALSE || FALSE

## [1] FALSE
```

- As a practical example, if we want to test whether a variable `x` lies *between* two numbers, say 60 and 70, we type:

```
x <- 75
x > 60 && x < 70

## [1] FALSE
```

  and to test whether it lies *outside* the range 60 to 70, we type:

```
x < 60 || x > 70

## [1] TRUE
```

- Here's an example of using `&&` in an `if()` statement:

```
x <- 3
y <- 5
if(x < 10 && y < 10) {
  print("Both less than 10")
} else {
  print("Not both less than 10")
}

## [1] "Both less than 10"
```

- The negation operator, `!`, returns "the opposite" of a logical expression:

```
!TRUE
```

```
## [1] FALSE
```

```
!FALSE
```

```
## [1] TRUE
```

```
!(5 < 6)
```

```
## [1] FALSE
```

- Pay attention to the operator precedence for `&&`, `||`, and `!`. It can be found by typing:

```
?Syntax
```

  but parentheses can be used to control the order of operations.

- If we try to apply `&&` or `||` to *vectors*, R only applies it to their first elements:

```
c(TRUE, FALSE, TRUE) && c(TRUE, TRUE, FALSE)
```

```
## [1] TRUE
```

3.3 Logical Operations on Logical Vectors

- To apply the operations "and" and "or" elementwise on two logical vectors, use `&` and `|`. For example:

```
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)
```

```
## [1]  TRUE FALSE FALSE
```

- `&` and `|` are useful in `ifelse()` statements. (Recall that `ifelse()` operates elementwise on vectors.). For example, consider the systolic and diastolic blood pressure readings:

```
systolic <- c(110, 119, 111, 113, 128)
diastolic <- c(70, 74, 88, 74, 83)
```

A blood pressure is classified as normal if the systolic level is less than 120 *and* the diastolic level is less than 80:

```
classification <- ifelse(systolic < 120 & diastolic < 80,
                         yes = "Normal",
                         no = "Abnormal")
classification

## [1] "Normal"   "Normal"   "Abnormal" "Normal"   "Abnormal"
```

- In the next example, we use `&` in square brackets `[ ]` to extract rows from a data frame:

```
bpData <- data.frame(
  name = c("Joe", "Katy", "Bill", "Kim", "Mark"),
  systolic = c(110, 119, 111, 113, 128),
  diastolic = c(70, 74, 88, 74, 83))
bpData

##   name systolic diastolic
## 1  Joe      110        70
## 2 Katy      119        74
## 3 Bill      111        88
## 4  Kim      113        74
## 5 Mark      128        83
```

```
attach(bpData)
bpData[systolic < 120 & diastolic < 80, ]

##   name systolic diastolic
## 1  Joe      110        70
## 2 Katy      119        74
## 4  Kim      113        74

detach(bpData)
```

4 Writing A Function

- To write your own function, you need to use the function `function()`, specify argument(s) that will be used in the function (with our without defaults), and in curly brackets specify what the function should do (referring to the argument(s)), including whether something is returned. The general form is:

```
myFun <- function(arg1, arg2, ...) {
  ## expressions that use the arguments
  ## the last command is what you want the function to return
}
```

  Each line inside the function is an object assignment, a function call, a subsetting, a conditional statement, an if/else statement, a for loop, etc. - basically anything you have now learned how to do in R that you want the function to do! If you do specify arguments, you should use them.

  To have the function output something, you must return the object (either the last command is just the object name or you can use `return()`). If you have multiple objects to return, I recommend returning everything as a list e.g. put this in the last line: `return(list(object1, object2))`.

- Here are a few examples of functions with no default arguments; note the different outputs:

```
do1 <- function(x, y){
  z <- x + y
  x
  z
}
do1(x = 1, y = 3) ## note that x is not returned

## [1] 4

do2 <- function(x, y){
  z <- x + y
  return(x)
  z
}
do2(x = 1, y = 3) ## note that z is returned

## [1] 1
```

```r
do3 <- function(x, y){
  z <- x + y
  return(list(x, z))
}
do3(x = 1, y = 3) ## x and z are returned as a single list

## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
```

- Here is an example of a function with default arguments that returns a vector:

```r
do4 <- function(x, y = 2){
  z1 <- x + y
  z2 <- x * y
  c(z1, z2)
}
do4(x = 1) ## uses y = 2

## [1] 3 2

do4(x = 1, y = 3) ## overwrites default value of y

## [1] 4 3
```

- Here is an example of a function that takes no arguments:

```r
do5 <- function(){
  sum(2, 4, 6)
  print("Hello World!")
  return(mean(1, 3, 5))
}
do5()

## [1] "Hello World!"
## [1] 1
```

- Recall that you can create functions with variable arguments using `...`. For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

```
meanOfAll <- function(...) {
  return(mean(c(...)))
}
```

The command

```
meanOfAll(usSales, europeSales, otherSales)
```

would combine the three vectors and take the mean of all the data.

5 The `source()` Function

- `source()` is a nice function for reading in big chunks of R code, e.g. a set of functions that you want to use every time you start a new R session.

```
source()       # Read R commands from a text file.
```

- For example, suppose we have the following commands saved in a text file '**C:\myRcode.txt**':

myFun < − function(message) {
    print(message)
}

myFun("Hello World")

We can execute those commands using `source()` by running:

```
source("myRcode.txt")
```

```
## [1] "Hello World"
```

6 Document License