MTH 3220 R Notes 8

9 R Programming Structures

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

9.1.1 Conditional Execution Using `if()` and `else`

- If we want R to execute a statement only if a certain condition is met, we can use:

```
if()          # Used to execute a statement only if a specified con-
              # dition is met
else          # Used with if() to specify an alternative statement to
              # be executed if the condition isn't met
```

- Here's a simple example of an `if` statement:

```
y <- 5
```

```
if (y > 0) x <- 1
```

```
x
```

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

- The general form of an `if` statement is:

```
if (cond) statement
```

where `cond` is a `"logical"` expression (i.e. it evaluates to `TRUE` or `FALSE`), and `statement` is only executed if `cond` is `TRUE`. If `cond` is `FALSE`, nothing happens.

- To (conditionally) execute *more than one* statement, enclose them in curly brackets `{ }`, like this:

```
if (cond) {
  statement1
  statement2
  .
  .
  .
  statementq
}
```

- Here's an example of an `if` statement followed by `else`:

```
y <- -6
```

```
if (y > 0) x <- 1 else x <- 2
```

```
x
```

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

- The general form of an `if` statement followed by `else` is:

```
if (cond) statement1 else statement2
```

If `cond` is `TRUE`, `statement1` is executed. If it's `FALSE`, `statement2` is executed. Above, `statement1` and `statement2` could each be replaced by a *set* of statements enclosed in curly brackets `{ }`.

9.1.2 Vectorized `if` and `else`: The `ifelse()` Function

- The statements that use `if` and `else` aren't *vectorized*. More specifically, if `cond` is a `"logical"` *vector*, only the first element is used.
- The `ifelse()` function is a vectorized version of `if` followed by `else`:

```
ifelse()          # Takes a "logical" vector, and returns a vector of
                  # equal length with values depending on whether the
                  # corresponding "logical" value is TRUE or FALSE.
```

- `ifelse()` takes arguments `test`, a `"logical"` vector (usually expressed as a condition to be met by elements of another vector), `yes`, the return value when `test` is `TRUE`, and `no`, the return value when `test` is `FALSE`.
- Here's a simple example:

```
ifelse(test = c(FALSE, TRUE, FALSE), yes = "a", no = "b")
```

```
## [1] "b" "a" "b"
```

Above, the returned vector contains elements `"a"` (the value of `yes`) or `"b"` (the value of `no`) depending on whether the corresponding element of `test` is `TRUE` or `FALSE`.

- Here we classify peoples' heights as `"short"` or `"tall"`:

```
height <- c(69, 71, 67, 66, 72, 71, 61, 65, 73, 70, 68, 74)
```

```
ifelse(height > 69, yes = "tall", no = "short")
```

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

Section 9.1 Exercises

Exercise 1 Here's a variable `x`:

```
x <- 4
```

Guess what the resulting value of **z** will be after each of the following sets of commands is executed, then check your answers.

a)
```
z <- NULL
if (x > 2) z <- 1
z
```

b)
```
z <- NULL
if (x < 3) z <- 1
z
```

c)
```
z <- NULL
if (x < 3) z <- 1 else z <- 2
z
```

Exercise 2 Here's a `"character"` vector containing responses to a survey question:

```
response <- c("Agree", "Agree", "Disagree", "Agree", "Disagree", "Disagree",
              "Disagree", "Disagree", "Agree", "Disagree")
```

Guess what will be returned by the following command, then check your answer:

```
ifelse(test = response == "Agree", yes = 1, no = 0)
```

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

9.2.1 Logical Operations and Compound Logical Expressions

- ***Logical operators*** (or ***Boolean operators***) correspond to "and", "or", and "not", and are written in R as:

```
!               # "Not"
&               # "And"
|               # "Or"
```

- These operate on `"logical"` (`TRUE` or `FALSE`) expressions and return `TRUE` or `FALSE` values. They're listed above in order of operator precedence (highest to lowest).

9.2.2 Logical Operations on Scalar Logical Expressions

- `&` returns `TRUE` if both expressions are `TRUE`, and it returns `FALSE` if at least one expression is `FALSE`:

```
TRUE & TRUE

## [1] TRUE
```

```
TRUE & FALSE

## [1] FALSE
```

- `|` returns `TRUE` if at least one of the expressions is `TRUE`, and it returns `FALSE` if both expressions are `FALSE`:

```
FALSE | TRUE

## [1] TRUE
```

```
FALSE | FALSE

## [1] FALSE
```

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

```
!TRUE

## [1] FALSE

!FALSE

## [1] TRUE
```

- As an example, to test whether a variable `x` lies *between* two numbers (60 and 70), we type:

```
x <- 63
x > 60 & x < 70

## [1] TRUE
```

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

```
x < 60 | x > 70

## [1] FALSE
```

- The logical operators `&`, `|`, and `!` operate *elementwise* on `"logical"` vectors, pairing corresponding values in the two vectors. For example:

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

## [1]  TRUE FALSE FALSE
```

- As another example, here are two vectors, `Syst` and `Diast`, containing systolic and diastolic blood pressures for five people:

```
Syst

## [1] 110 119 111 113 128

Diast

## [1] 70 74 88 74 83
```

A blood pressure considered normal if the systolic level is less than 120 *and* the diastolic level is less than 80. To identify the people with normal blood pressures, we can type:

```
Syst < 120 & Diast < 80

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

or we can use `which()`:

```
which(Syst < 120 & Diast < 80)

## [1] 1 2 4
```

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

```
bp

##   Name Syst Diast
## 1  Joe  110    70
## 2 Katy  119    74
## 3 Bill  111    88
## 4  Kim  113    74
## 5 Mark  128    83
```

To extract the rows corresponding to people whose blood pressures are normal (systolic less than 120 and diastolic less than 80), we type:

```
bp[bp$Syst < 120 & bp$Diast < 80, ]

##   Name Syst Diast
## 1  Joe  110    70
## 2 Katy  119    74
## 4  Kim  113    74
```

We could also use:

```
subset(x = bp, subset = Syst < 120 & Diast < 80)
```

- Be aware that the operator precedence order for the `"logical"` operators, from highest to lowest, is `!`, `&`, then `|`. More information can be found by typing:

```
? Syntax
```

Parentheses can be used to control the order of operations – operations within parentheses are carried out first

Section 9.2 Exercises

Exercise 3 Here are two variables `x` and `y`:

```
x <- 4
y <- 7
```

Guess what the result of each of the following will be, then check your answers.

a) `x > 2 & y == 7`

b) `x < 0 | y == 7`

c) `!(x < 0)`

Exercise 4 The logical operators `&`, `|`, and `!` operate *elementwise* on `"logical"` vectors, pairing corresponding values in the two vectors. Guess what the result of each of the following will be, then check your answers.

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

b) `c(FALSE, TRUE, FALSE) | c(TRUE, TRUE, FALSE)`

c) `!c(FALSE, TRUE, FALSE)`

Exercise 5 Recall that `is.na()` will identify missing values (`NA`s) in a vector. Here's a vector `x`:

```
x <- c(1, 2, NA, 6, NA, 5)
```

a) See what happens when you type:

```
is.na(x)
```

b) Guess what the following command will return, then check your answer:

```
!is.na(x)
```

c) Guess what the following command will return, then check your answer:

```
which(!is.na(x))
```

d) We want to extract all the non-missing values from `x`. Verify that the following command does what we want:

```
x[!is.na(x)]
```

Exercise 6 Here are two vectors, `Gender` and `Age`:

```
Gender <- c("m", "f", "m", "f", "f")
Age <- c(27, 34, 55, 21, 43)
```

Guess what the following commands will return, then check your answers.

a) `Gender == "m" & Age > 40`

b) `Gender == "m" | Age > 40`

Exercise 7 Here's a data frame `x`:

```
dat <- data.frame(Gender = c("m", "f", "m", "f", "f"),
                  Age = c(27, 34, 55, 21, 43),
                  Height = c(60, 58, 65, 55, 59),
                  Weight = c(160, 129, 174, 170, 130))
dat
```

```
##   Gender Age Height Weight
## 1      m  27     60    160
## 2      f  34     58    129
## 3      m  55     65    174
## 4      f  21     55    170
## 5      f  43     59    130
```

a) We want to extract the rows of `x` corresponding to males who are over the age of 40. Verify that the following commands both do what we want.

```
dat[dat$Gender == "m" & dat$Age > 40, ]

subset(x = dat, subset = Gender == "m" & Age > 40)
```

b) Now we want to extract the rows of `x` corresponding to people who are *either* male *or* over the age of 40. Verify that the following commands both do what we want.

```
dat[dat$Gender == "m" | dat$Age > 40, ]

subset(x = dat, subset = Gender == "m" | Age > 40)
```

Exercise 8 Recall that the operator precedence of the logical operators, from highest to lowest, is `!`, `&`, and `|`. The order of operations can be controlled using parentheses – operations within parentheses are carried out first. Guess what the result of each of the following commands will be, then check your answers.

a) `TRUE | TRUE & FALSE`

b) `(TRUE | TRUE) & FALSE`

c) `9 == 9 | 5 < 6 & 3 < 2`

d) `(9 == 9 | 5 < 6) & 3 < 2`

e) `!TRUE & FALSE`

f) `!(TRUE & FALSE)`

g) `!(9 == 9) & 4 < 3`

h) `!(9 == 9 & 4 < 3)`

9.3 User-Defined Functions

- We can create our own ***user-defined function*** using:

```
function()          # Used to create user-defined functions.
return()            # Used within a function definition to terminate
                    # the function call and return a value.
```

- Here's a simple (mathematical) function $f(x)$:

$$f(x) = x^2$$

In R, we can represent this as user-defined function that takes an argument `x` and returns its square:

```
my.f <- function(x) {
  return(x^2)
}
```

We use user-defined functions just as we would built-in functions:

```
my.f(x = 2)

## [1] 4
```

We can look at the function definition by typing its name on the command line:

```
my.f

## function(x) {
##   return(x^2)
## }
```

- The general format for creating a *user-defined function* is:

```
my.fun <- function(arg1, arg2, ..., argk) {
     statement1
     statement2
          .
```

```
        .
        .
    statementq
    return(value)              # The value to be returned.  We could
}                              # also just write value (without return()).
```

Above,

- `arg1`, `arg2`, ..., `argk` are argument names (that we're free to choose) for k (formal) arguments.
- `statement1`, `statement2`, ..., `statementq` are a set of q statements (which may involve `arg1`, `arg2`, ..., `argk`).
- `value` is a value (or expression that gives a value) to be returned by the function.

- As another example, here's a (mathematical) function that computes the average absolute value of two numbers x and y:

$$f(x, y) = \frac{|x| + |y|}{2},$$

We can write this function in R as:

```
AvgAbsVal <- function(x, y) {
  avg <- (abs(x) + abs(y)) / 2
  return(avg)
}
```

We can now call `AvgAbsVal()`, passing it values via the arguments `x` and `y`:

```
AvgAbsVal(x = -4, y = 2)

## [1] 3
```

- In fact, we don't need to use `return()`. In the absence of a `return()` statement, R returns the value of any expression that appears by itself as the last line of the function definition:

```
AvgAbsVal <- function(x, y) {
  avg <- (abs(x) + abs(y)) / 2
  avg
}
```

- If the body of the function consists of just a `value` (or expression), we can omit the curly brackets `{ }` as long as we write the entire function definition on a single line:

```
my.f <- function(x) x^2
```

9.3.1 Formal Arguments and Actual Arguments

- A function's arguments are sometimes referred to as ***formal*** arguments. Values passed to the function are referred to as ***actual*** arguments.

For example, below, `x` is a *formal* argument, and the value `5` is an *actual* argument:

```
g <- function(x) {
  return(x + 1)
}
```

```
g(x = 5)              # x is a formal argument, 5 is an actual argument

## [1] 6
```

9.3.2 Specifying Default Values for a Function's Arguments

- We can specify default values for one or more of a function's arguments by specifying `arg = expr` in the function definition:

```
my.fun <- function(arg1, arg2 = expr2, ..., argk = exprk) {
     statement1
     statement2
          .
          .
          .
     statementq
     value
  }
```

If values aren't passed for arguments that have default values during a function call, the default values are used.

- For example, below we define `AvgAbsVal()` so that the default value for `y` is 0:

```
AvgAbsVal <- function(x, y = 0) {
  avg <- (abs(x) + abs(y)) / 2
  avg
}
```

Now, if a value for `y` isn't passed to `AvgAbsVal()`, it uses 0:

```
AvgAbsVal(x = -120)

## [1] 60
```

9.3.3 Variable Number of Arguments Using "..."

- Functions can be written to take a *variable number* of arguments. The argument name `...` in the function definition will match any number of arguments.

  Within the body of the function, we can refer to `...` as if it was the name of a variable.

- For example, here's a function that returns the mean of all the values in *an arbitrary number of vectors*:

```
mean.of.all <- function(...) {
    overall.mean <- mean(c(...))
    return(overall.mean)
}
```

If `us.sales`, `europe.sales`, and `other.sales` were numeric vectors, the command

```
mean.of.all(us.sales, europe.sales, other.sales)
```

would combine them and take the mean of the combined data. The effect of `c(...)` is as if `c(us.sales, europe.sales, other.sales)` were called with the same three vectors that were passed as arguments to `mean.of.all()`.

- Many of R's built-in functions take a variable number of arguments. For example look at the help files for `list()` and `c()` by typing:

```
? list
? c
```

9.3.4 Printing Warning or Error Messages Using `warning()` or `stop()`

- The following functions are useful for terminating a function call or just printing a warning message:

```
stop()          # Terminate a function call and print an error message.
warning()       # Print a warning message (without terminating the
                # function call).
```

- `stop()` and `warning()` are usually used in `if()` statements within function definitions.
- `stop()` terminates a function call (without returning a value) and prints an error message. Here's an example:

```
my.ratio <- function(x, y) {
    if (y == 0) stop("Cannot divide by 0")
    x/y
}
```

An attempt to pass the value `0` for `y` now results in the following:

```
my.ratio(x = 3, y = 0)

## Error in my.ratio(x = 3, y = 0):  Cannot divide by 0
```

(Note that the last line, `x/y`, was never encountered during the call to `my.ratio()`.)

- `warning()` just prints a warning message to the screen without terminating the function call. Here's an example:

```
my.ratio <- function(x, y) {
    if (y == 0) warning("Attempt made to divide by 0")
    x/y
}
```

Now when we pass the value `0` for `y`, the function call isn't terminated (`Inf` is returned), but we get a warning message:

```
my.ratio(x = 3, y = 0)

## Warning in my.ratio(x = 3, y = 0):  Attempt made to divide by 0

## [1] Inf
```

Section 9.3 Exercises

Exercise 9 Here's a function:

```
f1 <- function(x) {
    y <- x + 1
    return(y)
  }
```

If we replace `return(y)` by just `y`:

```
f2 <- function(x) {
    y <- x + 1
    y
  }
```

does the function do the same thing? Check your answer by passing a few values to both `f1()` and `f2()` and comparing the results.

Exercise 10 Here's a (mathematical) function $f(x, y)$ that returns the absolute value of the *relative difference* between x and y:

$$f(x, y) = \left| \frac{x - y}{y} \right|.$$

The function could be defined in R by:

```
f <- function(x, y) {
      rel.diff <- (x - y)/y
      abs.rel.diff <- abs(rel.diff)
      abs.rel.diff
    }
```

a) What happens when you pass `f()` the values `x = 1` and `y = 0`? What about when you pass it `x = 0` and `y = 0`.

b) How would you rewrite the first line of the function,

```
f <- function(x, y) {
```

so that it specifies a *default* value of 1 for `y`?