4 - Conditional Statements, Iteration, Vectorization

Conditional Statements

Decision making or conditional execution of program logic is an important aspect of programming. This can be achieved in R programming using the conditional if...else statement.

The R if statement

The syntax of the R if statement is:

```
if (test_expression) {
      statement 1
      statement 2
      ...
      statement n
```

```
}
```

where the test_expression is a logical expression involving a single value. If the test_expression is TRUE, the "body" or "block" of statements between {} get executed. But if it's FALSE, nothing happens.

Flowchart of the if statement

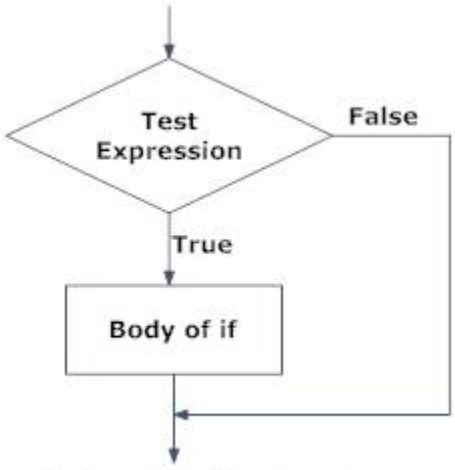

Fig: Operation of if statement

Example if statement

```
x <- 5

if(x %% 2 == 1) {   # using the %% modulus operator

      cat("That", x, "is an odd number.\n")
}

# output
That 5 is an odd number.

if(fahrenheit_to_celsius(32) <= 0) {
      print("It's freezing out there.")
}

# output
[1] "It's freezing out there."
```

R if...else statement

The syntax of if...else statement is:

```
if (test_expression) {
      statement 1
      statement 2
```

```
      …
      statement n
} else {
      statement 1
      statement 2
      …
      statement n
}
```

The else part is optional and is only evaluated if test_expression is FALSE. Note that *else* must be in the same line as the closing brace } of the *if* statement.

Flowchart of the if…else statement

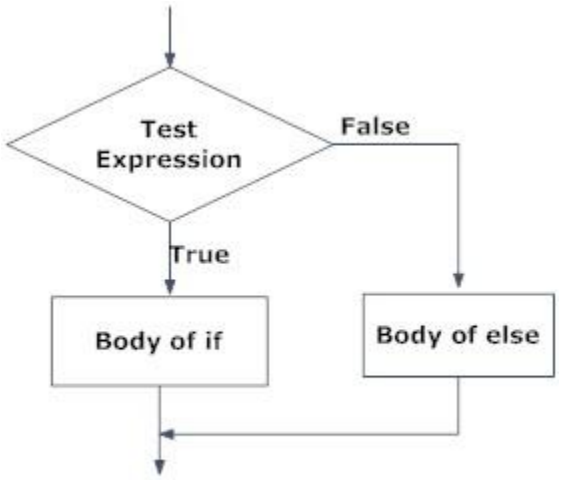

Fig: Operation of if…else statement

Example of if…else statement

```
x <- 44
if(x %% 2 == 1) {
      prarg <- paste("That", x, "is an odd number.\n")
      print(prarg)
} else {
      prarg <- paste("That", x, "is an even number.\n")
      print(prarg)
}
```

The if…else expression

The if…else statement can be treated as an expression in R, so you can have expressions that look like this:

```
# value of numeric y will be set conditionally, depending on the value of x
y <- if(x > 0) 5 else 6
```

```
# value of string prarg will be set conditionally, depending on the value of x
prarg <- if(x %% 2 == 1) paste("That", x, "is an odd number.\n") else paste("That", x, "is an even number.\n")
```

if...else "Ladder" or nested if...else

The if...else ladder (if...else...if) statement allows you execute a block of code among more than 2 alternatives.

The syntax of if...else statement is:

```
if ( test_expression1) {
      statement(s)
} else if ( test_expression2) {
      statement(s)
} else if ( test_expression3) {
      statement(s)
} else {
      statement(s)
}
```

Only one set of statements will get executed, depending upon the test_expressions.

Example of nested if...else

```
x <- 44

if(x == 0) {
      prarg <- paste("That", x, "is a zero.\n")
} else if(x %% 2 == 1) {
      prarg <- paste("That", x, "is an odd number.\n")
} else {
      prarg <- paste("That", x, "is an even number.\n")
}

print(prarg)
```

Vectors, if...else, and ifelse()

In R, vectors are "first-class objects". They can be used in expressions like single numbers...

```
v <- c(13, 14, 15, 16, 17)
v + 1

# output
[1] 14 15 16 17 18
```

```
# ...including logical (boolean) expressions
v > 14

# output is a vector of logicals
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

However, vectors can't be directly used in if...else expressions:

```
v <- c(13, 14, 15, 16, 17)

# if an element in v is > 14, add 10 to it, else make it NA
w <- if(v > 14) v+10 else NA     # NA is a special value, used for missing values

# output
# Warning message:
# In if (v > 14) v + 10 else NA :
#  the condition has length > 1 and only the first element will be used

w
[1] NA       # w contains only a single NA value

# Or, an error may occur:
#     Error in if (v > 14) v + 10 else NA : the condition has length > 1
# and w will not be defined at all.
```

There is a vector equivalent form of the if...else statement in R, the *ifelse()* function.

Syntax of ifelse() function

```
ifelse(test_expression, x, y)
```

Here, test_expression must result in a logical vector. What the ifelse() function returns is a vector with the same length as the test_expression-generated vector.

The returned vector has an element from vector x if the corresponding value of the test_expression vector is TRUE or from vector y if the corresponding value of test_expression is FALSE. The i-th element of the returned vector will be x[i] if test_expression[i] is TRUE else it will take the value of y[i].

The vectors x and y are recycled whenever necessary (if x and y are not the same length as the input vector).

Example: ifelse() function

```
a <- c(5, 7, 2, 9)
ifelse(a %% 2 == 0, 'even', 'odd')

# output
[1] "odd"  "odd"  "even" "odd"
```

In the above example, the test_expression is **a %% 2 == 0** which will result in a logical vector (FALSE, FALSE, TRUE, FALSE). The other two function arguments, "even" and "odd", are single element vectors, so they get recycled to ("even", "even", "even", "even") and ("odd", "odd", "odd", "odd"), respectively, to match the length of the test_expression vector.

Without recycling of the TRUE and FALSE vectors, we would have to write the ifelse() like this instead:

```
a = c(5, 7, 2, 9)
ifelse(a %% 2 == 0, c('even', 'even', 'even', 'even'), c('odd', 'odd', 'odd', 'odd'))

# output
[1] "odd"  "odd"  "even" "odd"
```

Vector recycling will happen whenever the TRUE or FALSE vectors are not long enough:

```
a = c(5, 7, 2, 9, 4, 10)
ifelse(a %% 2 == 0, c('EVEN', 'even'), c('ODD', 'odd'))

# the TRUE and FALSE vectors are recycled to
# ('EVEN', 'even', 'EVEN', 'even', 'EVEN', 'even') and
# ('ODD', 'odd', 'ODD', 'odd', 'ODD', 'odd')

# output
[1] "ODD"  "odd"  "EVEN" "odd"  "EVEN" "even"
```

Back to the original problem:

```
v <- c(13, 14, 15, 16, 17)

# if an element in v is > 14, add 10 to it, else make it NA
ifelse(v > 14, v + 10, NA)

# output
[1] NA NA 25 26 27

# if fully expanded, the ifelse() function call looks like
#      ifelse(v > 14, c(23, 24, 25, 26, 27), c(NA, NA, NA, NA, NA))
# where the expansion of NA to a 5-element vector would happen through automatic recycling
```

Iteration

Loops are used in programming to perform *iteration*, i.e., to repeat a specific block of code a certain number of times or while/until some condition is true.

The R for loop

In R, the ***for*** loop can be used to iterate over a vector.

The syntax of the R *for* loop is:

```
for (value in sequence)
{
       statement 1
       statement 2
       …
       statement n
}
```

Here, '*sequence*' is a vector and '*value*' takes on each of the '*sequence*' values during the loop. In each iteration, the statement(s) are evaluated.

Flowchart of the *for* loop

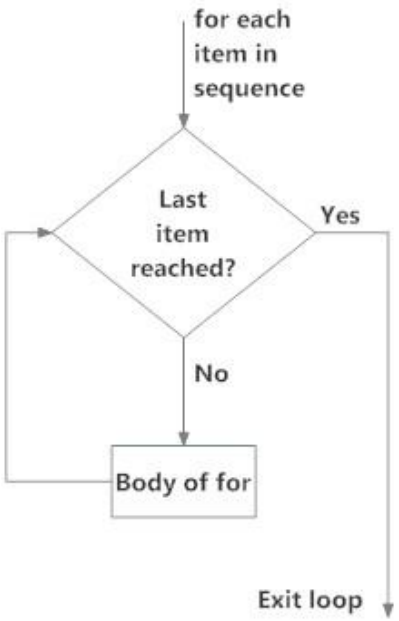

Fig: operation of for loop

Example *for* loop

Below is an example to count the number of even numbers in a vector.

```
x <- c(2, 5, 3, 9, 8, 11, 6)
count <- 0   # a counter variable

for (val in x) {
```

```
      if (val %% 2 == 0) {
            count <- count + 1
      }
}

print(count)

# output
[1] 3
```

In the above example, the loop iterates 7 times as the vector x has 7 elements.

In each iteration, val takes on the value of each element of x.

A counter variable (count) is used to count the number of even numbers in x.

Example: Find the factorial of a number

```
# get input from the user (Run the following line first in R.)
num <- as.integer(readline(prompt='Enter an integer: '))

factorial <- 1

# check if the number is negative, positive or zero
if (num < 0) {
      print("Sorry, factorial cannot be computed for negative numbers.")
} else if (num == 0) {
      print("The factorial of 0 is 1.")
} else {
      for(i in 1:num) {
            factorial <- factorial * i
      }

      print(paste("The factorial of", num ,"is", factorial, "."))
}
```

Note: We could have just used the built-in function factorial() to do this.

Example: Loop through columns of a data frame

```
# loop through mtcars data frame columns and print
# max, min, mean of each column (minus the last 4 columns)

# get names of first 7 columns in mtcars data frame
col_names <- names(mtcars)[1:7]

for(col_name in col_names) {
   # using cat function to print
   cat(col_name, '-',
       '\n',
       '\tmax:\t',  max (mtcars[[col_name]]), '\n',
```

```
      '\tmin:\t',  min (mtcars[[col_name]]), '\n',
      '\tmean:\t', mean(mtcars[[col_name]]), '\n',
      '\n')

   # note: mtcars$col_name won't work; $ can't be used before variable names,
   #       so have to use double bracket notation [[ ]], not the $ shorthand
}
```

The R while loop

The *while* loop is used to loop until a specific condition is not met.

The syntax of *while* loop is:

```
while (test_expression)
{
      statement 1
      statement 2
      ...
      statement n
}
```

Here, '*test_expression*' is a logical (TRUE/FALSE) expression that is evaluated. The body of the loop inside {} is entered if the result is TRUE.

The statements inside the loop body are executed and the flow returns to evaluate the '*test_expression*' again.

This is repeated until '*test_expression*' evaluates to FALSE, in which case, the loop exits.

Flowchart of the *while* loop

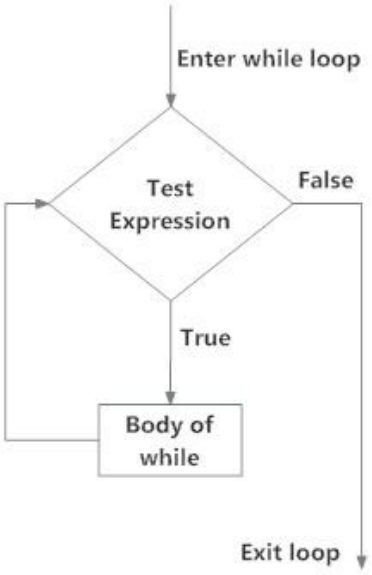

Fig: operation of while loop

Example *while* loop

```
i <- 1              # counter variable
while (i < 6) {
      print(i)
      i <- i + 1    # increment i by 1
}

# output:
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

In the above example, i is first initialized to 1. The test_expression is i < 6 which evaluates to TRUE since 1 is less than 6. So, the body of the loop is entered for the first iteration, and i is printed and incremented by 1. **Incrementing i is important as this will eventually make the test_expression FALSE**; failing to do that will result in an "infinite loop."

In the second iteration, the value of i is 2 and the loop continues. This will continue until i takes the value 6, making the test_expression 6 < 6, which is FALSE, causing the while loop to finally exit.

The *while* loop is suitable for when you don't know how long the input list or sequence will be. This is common when doing ***simulation***-type work. *For example, if simulating coin flips, you might want to loop until you get three heads in a row. You can't do that sort of iteration with the for loop.*

A *while* loop is more general than a *for* loop, because you can rewrite any *for* loop as a *while* loop, but you can't rewrite every *while* loop as a *for* loop:

```
for (i in vector_x) {
  # body
}

# is equivalent to
i <- 1
while (i <= length(vector_x)) {
  # body
  i <- i + 1
}
```

Example: Simulate required number of coin flips to get three heads in a row:

```
# flips: return total number of coin flips that were required
#        to get three heads in a row
#
flips <- function() {
   total_flips <- 0
   nheads <- 0

   # if nheads becomes 3, we got three 'heads' in a row
   while (nheads < 3) {
     if (sample(c("T", "H"), 1) == "H") {
       nheads <- nheads + 1
     } else {
       # didn't get heads; must reset to 0
       nheads <- 0
     }

     total_flips <- total_flips + 1
   } # end while

   return(total_flips)

} # end flips()


# call flips function
flips()

# output
[1] 21
```

Vectorization

Functions in R are "vectorized" which makes loops less important in R. Many looped tasks can be performed using functions instead.

For example, counting even numbers in a vector:

```
x <- c(2, 5, 3, 9, 8, 11, 6)

# using a loop requires some setup
count <- 0   # a counter variable
# then the loop
for (val in x) {
       if (val %% 2 == 0) {
              count <- count + 1
      }
}

print(count)

# output
[1] 3




# could use functions instead
#
# x %% 2 == 0                    -> TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
# as.integer(x %% 2 == 0)        -> 1 0 0 0 1 0 1
# sum(as.integer(x %% 2 == 0))   -> 3
#
# Also works: sum(x %% 2 == 0)   -> 3
```

or sum two numeric vectors:

```
a <- 1:10
b <- 1:10

result <- vector(mode = 'numeric', length = length(a))     # create result vector using vector()

# use a loop
for (i in a) {
  result[i] <- a[i] + b[i]
}

result

# output
[1]  2  4  6  8 10 12 14 16 18 20



# could use a vectorized function instead
```

```
#
result2 <- a + b

# use another vectorized function to see if two vectors are all equal
all.equal(result, result2)
```

In R, the arithmetic operators, like + and %%, are vectorized functions which can operate on entire vectors at once.

The *apply* function

A *for* loop can be used to call the same function on a collection of objects. R has a family of functions, the *apply* family, which can be used to "apply" the same function to a collection of objects *without a loop*. Recall the use of sapply for the iris data frame,

```
# show the class of each column in the iris data frame
#
sapply(iris, class)
```

We could write a *for* loop to do the same job:

```
# show the class of each column in the iris data frame
#
for (column in names(iris)) {
   cat(column, ':\t',
       class(iris[[column]]),
       '\n')
}
```

The *apply* functions are "higher-order" functions, in that these functions take another function name as an input parameter. The *apply* function is the base function that requires several input arguments:

```
# apply the mean function to the first four columns of iris
# see ?apply
#
# MARGIN is 1 for row, 2 for columns;
# apply returns a named vector of four elements
apply(X = iris[1:4], MARGIN = 2, FUN = mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333

# can be simplified to
apply(iris[1:4], 2, mean)
```

The *lapply* and *sapply* functions are variations of apply:

```
# apply the mean function to the first four columns of iris
# see ?lapply
```

```
#
# lapply returns a list of four components
lapply(iris[1:4], mean)
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333

# sapply returns a vector or matrix by default
sapply(iris[1:4], max)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
         7.9          4.4          6.9          2.5
```

User-defined functions can also be applied:

```
# define a very short 'midrange' function
# input1: v, must be a numeric vector
# returns: (min(v) + max(v)) / 2       (even though 'return' is not used)
midrange <- function(v) (min(v) + max(v)) / 2

# apply it to first four columns of iris
sapply(iris[1:4], midrange)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
        6.10         3.20         3.95         1.30

# sapply(iris[1:4], midrange) returns a vector with named elements:
is.vector(sapply(iris[1:4], midrange))
[1] TRUE

# a different way to get the same information as the mtcars for loop example above
# by 'sapply'ing the 'summary' statistics function:
sapply(mtcars[1:7], summary)
             mpg    cyl     disp       hp     drat      wt     qsec
Min.    10.40000 4.0000  71.1000  52.0000 2.760000 1.51300 14.50000
1st Qu. 15.42500 4.0000 120.8250  96.5000 3.080000 2.58125 16.89250
Median  19.20000 6.0000 196.3000 123.0000 3.695000 3.32500 17.71000
Mean    20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875
3rd Qu. 22.80000 8.0000 326.0000 180.0000 3.920000 3.61000 18.90000
Max.    33.90000 8.0000 472.0000 335.0000 4.930000 5.42400 22.90000
```