R Control Flow Tools

Norman Lo

1/10/2022

R Control Flow Tools

In order to control the execution of the expression flow in R, we make use of the logical **Control flow tools** or control structures in R. R provides various standard control flow tools for testing different conditions and facilitates the flow of execution to be controled inside a function. We cover the following control flow tools in R in this lesson:

- if-else
- switch
- ifelse
- for loop
- while loop
- break and next

4.1 - if and else statements

Like most of the programming languages, logical test is usually performed using the '**if**' statement. The 'if' statement evaluates an expression(s) and check whether a condition is met, then returns a **boolean** condition, where one value (1) if TRUE and another value (0) if FALSE. The code(s) will be executed only if the condition is returned as TRUE; otherwise, it either jumps into the an '**else**' condition test or ends the program flow. As mentioned in the previous sections, TRUE has the equal value of "1" and FALSE is equal to "0" in R.

```
# Check the numeric value of TRUE
as.numeric(TRUE)
```

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

```
# Check the numeric value of FALSE
as.numeric(FALSE)
```

```
## [1] 0
```

When we apply the logical test, it returns either TRUE or FALSE. We can evaluate the following condtions and check the return values.

```
# Check the condition 1 equals 1
1 == 1
```

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

```r
# Check the condition 1 is less than 1
1 < 1
```

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

```r
# Check the condition 1 is less than or equal to 1
1 <= 1
```

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

```r
# Check the condition 1 is greater than 1
1 > 1
```

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

```r
# Check the condition 1 is greater than or equal to 1
1 >= 1
```

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

```r
# Check the condition 1 is not equal to 1
1 != 1
```

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

Now, let's combine the condition test with the 'if' statement for contolling the executions.

```r
# Assign 1 to a variable
toCheck <- 1

# Use the 'if' statement that return "hello" if the variable equals 1
if (toCheck == 1){
  print("hello")
}
```

```
## [1] "hello"
```

```r
# Use the 'if' statement that returns "hello" if the variable equals 0
if (toCheck == 0){
  print("hello")
}  # no return result in this case
```

'if' statement is similar to the R functions, the expression(s) is/are wrapped inside the curly brackets "{ }". In the previous example, when the condition is met or equals TRUE, the expression is executed. In more general cases, there could be multiple expressions or nested control statements to be wrapped under a single condition test. When the condition test fails or equals FALSE, no action will be taken. If we want to assign different action(s) for failing the condition test, we could use the 'else' statement to define.

```r
# Create a function and assign to a variable
check.bool <- function(x){
  # Create a condition test for this function
  if (x == 1) {
    print("hello")
  }
  # Create an alternative action
  else {
    print("goodbye")
  }
}
```

In this example, we added the 'else' statement to the function, so when the first condition is not met, the function returns "goodbye". It's important to pay attention to the indentation when writing the 'if' and 'else' statements. If they are not aligned in the more complex nested conditional tests, it would be easy to run into an error.

```r
# Using the function with different inputs
check.bool(1)
```

```
## [1] "hello"
```

```r
check.bool(0)
```

```
## [1] "goodbye"
```

```r
check.bool("k")
```

```
## [1] "goodbye"
```

```r
check.bool(TRUE)
```

```
## [1] "hello"
```

If we need to add multiple testing conditions to the function, an 'else if' statement would help. Let's build a function that test the condition if the variable equals 1, print "hello", if the variable equals 0, print "goodbye", for all other cases, print "confused".

```r
# Create a function with multiple condition tests
check.bool <- function(x){
  # First condition for x equals 1
  if (x == 1) {
    print("hello")
  }
  # Second condtion for x equals 0
  else if (x == 0) {
    print("goodbye")
  }
  # For all other cases
  else {
    print("confused")
  }
}

# Check the function with different values
check.bool(1)
```

```
## [1] "hello"
```

```
check.bool(0)
```

```
## [1] "goodbye"
```

```
check.bool(2)
```

```
## [1] "confused"
```

```
check.bool("k")
```

```
## [1] "confused"
```

4.2 - Switch

When there are multiple conditions to test, the code will get complex and difficult to read using if-else statements. R provides an alternative function “switch()”, which can complete to the task efficiently. The switch() function in R tests an expression against element of a list. If the value evaluated from the expression matches item from the list, the corresponding value is returned.

The first argument is the expression to be evaluated and follow by a list of named items and its values. If none of the items matches to the testing expression, it returns the predefined value. Here is an example to demonstrate:

```
# Create a function using switch()
use.switch <- function(x){
  # the expression x will be evaluated
  switch(x,
         # if "a", return "first"
         "a"="first",
         # if "b", return "second"
         "b"="second",
         # if "c", return "thrid"
         "c"="third",
         # if "z", return "last"
         "z"="last",
         # All other cases, return "other"
         "other")
}

# Test the function with different values
use.switch("a")
```

```
## [1] "first"
```

```r
use.switch("b")
```

```
## [1] "second"
```

```r
use.switch("c")
```

```
## [1] "third"
```

```r
use.switch("d")
```

```
## [1] "other"
```

```r
use.switch("e")
```

```
## [1] "other"
```

```r
use.switch("z")
```

```
## [1] "last"
```

If the evaluated expression is a numeric value, the item names ("a", "b", "c", and "z") will be ignored and the function returns the value of the item in the list based on the position index. If the numeric value is greater than the available index, the function returns NULL.

```r
# Test with numeric values
use.switch(1)
```

```
## [1] "first"
```

```r
use.switch(2)
```

```
## [1] "second"
```

```r
use.switch(3)
```

```
## [1] "third"
```

```r
use.switch(4)
```

```
## [1] "last"
```

```r
use.switch(5)
```

```
## [1] "other"
```

```r
# Nothing return from this expression
use.switch(6)

# Check the return value from the expression
is.null(use.switch(6))
```

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

4.3 - ifelse

The "if" statement in R is similar to the "if" statement in the traditional programming lanugage like C, C++, Java, etc. The ifelse() function in R is more like the application of if() function in Excel. The first argument specifies the conditional statement for testing, the second argument is the return value if the conditional test is TRUE, and the third argument is the return value if the conditional test is FALSE. Unlike the traditional "if" statement, one of the key advantage of ifelse() function in R is that it applies to vector and utilize vectorized operation. Here are few examples demonstrate the advantage of ifelse() function.

```r
# Test condition 1 == 1 with ifelse()
ifelse(1 == 1, "YES", "NO")
```

```
## [1] "YES"
```

```r
# Test condition 1 == 0 with ifelse()
ifelse(1 == 0, "YES", "NO")
```

```
## [1] "NO"
```

```r
# Test each elements in a vector equals 1 with ifelse()
toTest <- c(1, 1, 0, 1, 0, 1)
ifelse(toTest == 1, "YES", "NO")
```

```
## [1] "YES" "YES" "NO"  "YES" "NO"  "YES"
```

```r
# Modify the return values for the test
ifelse(toTest == 1, toTest*3, toTest)
```

```
## [1] 3 3 0 3 0 3
```

```r
# Modify the return values for the test
ifelse(toTest == 1, toTest*2, "ZERO")
```

```
## [1] "2"    "2"    "ZERO" "2"    "ZERO" "2"
```

```r
# Add NA value to the vector
toTest[2] <- NA

# Test each elements in the vector equals 1 with ifelse()
ifelse(toTest == 1, "YES", "NO")
```

```
## [1] "YES" NA    "NO"  "YES" "NO"  "YES"
```

```r
# Modify the return values for the test
ifelse(toTest == 1, toTest*10, toTest)
```

```
## [1] 10 NA  0 10  0 10
```

As you can see from these examples, ifelse() function can compute the conditional test for a vector without a loop operation. The return values can be flexibly customized. When an NA value is passed into the ifelse() function, it always returns an NA value.

All of the examples so far is setup to test a single condition within a statement. In many cases, it may be helpful to test multiple condition within a statement. We can apply the two conditional operators, **&&** and ||, for logical **AND** and **OR** statements, respectively.

Many people confuse about the logical operators **&** verus **&&** and | verus ||. Single operator and double operator perform the same logical operation, but the single operator performs elementwise comparisons in much the same way as arithmetic operators, where the double operator only evaluate the first element of each vector. Understanding the difference between the two has an signifcant impact to the performance of the program. Double operartor is best applying on "if" statement or loop operation, whereas single operator is best used with ifelse() function.

```r
# Create two vectors a and b
a <- c(1, 1, 0, 1)
b <- c(2, 1, 0, 1)

# Test each element in both vector equals 1 with single operator
ifelse(a == 1 & b == 1, "YES", "NO")
```

```
## [1] "NO"  "YES" "NO"  "YES"
```

```r
# Test the first element in both vector equals 1 with double operator
ifelse(a == 1 && b == 1, "YES", "NO")
```

```
## [1] "NO"
```

4.4 - for loop

Like we mentioned in the previous sections, many of the R built-in functions utilize the vectorized operation to improve performance and computational power. However, there are some cases where the calculations need to be done by iterating a vector, list, or data.frame. Therefore, R also features the **for** and **while** loop statements for the control flow commands.

The most common loop statement is a **for** loop. A for loop is used to iterate over a vector in R programming. A for loop include three arugments, which is similar to an English sentence. The first argument is a variable to be assigned a value from the vector. The middle argument is the key work "in". The last argument is the vector to be iterated. After for loop statement, a body of statement(s) can be added within the curly brackets "{ }" for execution under the specific condition. Let's take a quick look of this simple for loop example.

```r
# Create a for loop that loop through 1 to 10 and print each element from the vector
for (i in 1:10){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```r
# Print a vector 1 to 10 by vectorized operation
print(1:10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Note: vectorized operation can achieve the same task in this last example.

```r
# Create a vector of fruit names
fruit <- c("apple", "banana", "pomegranate")

# Create a vector of NAs with length same as fruit
fruitLength <- rep(NA, length(fruit))
fruitLength
```

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

```r
# Assign names to each element in the fruitLength vector
names(fruitLength) <- fruit
fruitLength
```

```
##       apple      banana pomegranate 
##          NA          NA          NA
```

```r
# Create a for loop to assign the fruit names' length to each fruit
for (a in fruit){
  fruitLength[a] <- nchar(a)
}

# Print the results
fruitLength
```

```
##       apple      banana pomegranate 
##           5           6          11
```

```r
# Achieve the same task using vectorized operation
# Apply nchar() function to fruit vector
fruitLength2 <- nchar(fruit)

# Name the elements in fruitLength2 vector
names(fruitLength2) <- fruit
```

```
# Print the result
fruitLength2
```

```
##       apple      banana pomegranate
##           5           6          11
```

```
# Check if the two results are identical
identical(fruitLength, fruitLength2)
```

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

4.5 - while loop

In R programming, **while** loop is less used compare to the **for** loop. While loop in R is again similar to while loop in any other programming language, which repeat the specific block of code until the condition is no longer satisfied. First the condition is evaluated and only when it holds TRUE, while loop enters the body. Here is an example to demonstrate the use of while loop and iterate a variable until it reaches 5.

```
# Assigne 1 to x
x <- 1

# Create a while loop that continue to print x while it's less than or equal to 5.
while (x <= 5){
  print(x)
  # After printing x, a value of 1 is added to x
  x <- x + 1
}
```

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

4.6 - break and next statements

Under certain condition, we may need to jump or skip an interation or even break or stop the loop. We can use the **next** or **break** statement to control the flow of the program. Here are two examples illustrate the use of the two statements.

```
# Create a for loop with next statement
for (i in 1:10) {
  # If i equals to 3, it skip to the next iteration
  if (i == 3){
    next
  }
  # If i not equals to 3, print i
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Note that the value 3 is not included in the return output because when i equals 3, it jumps into the **next** statement and jump to the next iteration.

If we want to stop a loop once it hits certain condition test, we can use the **break** statement to force a stop of the iteration.

```
# Create a for loop with break statement
for (i in 1:10){
  # If i equals to 4, break the for loop
  if (i == 4){
    break
  }
  # if i not equals to 4, print i
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

In this example, we see the output break after the value 3 is printed because when the iteration hits 4, it gets into the break statement and the for loop is forced to stop.

Summary:

We only introduce the basic control flow tools in this section, but there are many tools are also available in R. "for" loop is great for iterating a vector of elements and the "while" loop is flexible to repeat the same task(s) until certain condition is met. Like we mentioned many times before, R offers many vectorized operators for different data operation and we should always utilize these tools before using the loop operation because of the advantage of efficiency and performance. Also, unlike other programming language, we should always try to avoid writing nested loops in R because of the performance issue.