Introduction to R: control flow

You can go a long way in R without doing things that are typically associated with computer programming like writing your own loops and functions. In fact, it is quite possible to conduct data analysis and predictive modeling tasks using the tools built into R and its packages without writing your own code. That said, it is useful to learn a few basic programming constructs, even if you mainly want to use R as a tool for calculation and analysis. When you run code, R executes statements in the order in which they appear. Programming languages like R let you change the order in which code executes, which allows you to skip certain statements or run certain statements over and over again. Programming constructs that let you alter the order in which code executes are known as control flow statements.

If, Else and Else If

The most basic control flow statement in R is the if statement. An if statement checks if some logical expression is true or false and executes a specified block of code if the logical expression is true. In R, an if statement starts with if, followed by a logical expression in parentheses, followed by the code to execute when the if statement is true in curly braces:

```
x <- 10                 # Assign some variables
y <- 5

if                      # The keyword "if" starts an if statement
(x > y)                 # A logical expression follows the "if"
{print (x)}             # Code to execute is wrapped in curly braces "{}"
```

```
## [1] 10
```

In this case the logical expression was true– x was greater than y –so the print(x) statement was executed. The code above breaks the if statement out across 3 lines for descriptive purposes, but in practice it is considered good style to put the if, the logical expression and the opening curly brace on the same line, the code to execute on the following lines and then the final closing brace on its own line:

```
if (x > y) {             # This is standard R style for if statements
    print(x)
}
```

```
## [1] 10
```

If statements are often accompanied by else statements. Else statements come after if statements and allow you to execute code in the event that the logical expression of an if statement is false:

```
y <- 25                 # Reassign variable y to make it larger than x

if (x > y) {            # The original if statement
    print(x)
} else {                # With a new else statement added
    print(y)
}
```

```
## [1] 25
```

In this case, (x > y) evaluates to false so the code in the if block is skipped and the code in the else block is executed instead. You can extend this basic if/else construct to perform multiple logical checks in a row by adding one or more "else if" statements between the opening if and the closing else. Each else if statement performs another logical check and executes its code if the check is true:

```r
y <- 10

if (x > y) {
    print(x)
} else if (y == x) {              # A new else if statement that checks equality
    print("x and y are equal")
} else {
    print(y)
}
```

```
## [1] "x and y are equal"
```

For Loops

For loops are a programming construct that let you go through each item in a sequence and then perform some operation on each one. For instance, you could use a for loop to go through all the values in a vector and check whether each conforms to some logical expression or to print each value to the console:

```r
my_sequence <- seq(0,100,10)      # Create a numeric vector

for (item in my_sequence) {       # Create a new for loop over the specified items
    print(item)                   # Code to execute
}
```

```
## [1] 0
## [1] 10
## [1] 20
## [1] 30
## [1] 40
## [1] 50
## [1] 60
## [1] 70
## [1] 80
## [1] 90
## [1] 100
```

For loops support a couple of keywords that help control the flow of the loop: next and break. The next keyword causes a for loop to skip to the next iteration of the loop:

```r
for (item in my_sequence) {
    if (item < 50){               # This if statement skips items less than 50
        next
    }
    print(item)
}
```

```
## [1] 50
## [1] 60
## [1] 70
## [1] 80
## [1] 90
## [1] 100
```

*Note: In other programming languages, the keyword "continue" is often used instead of next. The break keyword halts the execution of the loop entirely. Use break to "break out" of a loop:

```
for (item in my_sequence) {
    if (item > 50){                # Break out of the loop if item exceeds 50
        break
    }
    print(item)
}
```

```
## [1] 0
## [1] 10
## [1] 20
## [1] 30
## [1] 40
## [1] 50
```

In the for loop above, substituting the next keyword for break would actually result in the exact same output but the code would take longer to run because it would still go through each item instead of breaking out of the for loop early. It is best to break out of loops early if possible to reduce execution time.

While Loops

While loops are similar to for loops in that they allow you to execute code over and over again. For loops execute their contents, at most, a number of iterations equal to the length of the sequence you are looping over. While loops, on the other hand, keep executing their contents as long as a certain logical expression you supply remains true:

```
x <- 5
iters <-0

while (iters < x) {         # Execute the contents as long as iters < x
    print("Study")
    iters <- iters+1        # Increment iters by 1 each time the loop executes
}
```

```
## [1] "Study"
## [1] "Study"
## [1] "Study"
## [1] "Study"
## [1] "Study"
```

While loops can get you into trouble because they keep executing until the logical statement provided is false. If you supply a logical statement that will never become false and don't provide a way to break out of the while loop, it will run forever. For instance, if the while loop above didn't include the statement incrementing the value of iters by 1, the logical statement would never become false and the code would run forever. Infinite while loops are a common cause of program crashes. The next and break statements work inside while loops just like they do in for loops. You can use the break statement to escape a while loop even if the logical expression you supplied is true. Consider the following while loop:

```
while (TRUE) {              # This logical expression is always true!
    print("Study")
    break                   # But we immediately break out of the loop
}
```

```
## [1] "Study"
```

It is important to make sure that while loops contain a logical expression that will eventually be false or a

break statement that will eventually be executed to avoid infinite loops. Although you can use a while loop to do anything a for loop can do, it is best to use for loops whenever you want to perform an operation a specific number of times. While loops should be reserved for cases where you don't know how many times you need to execute the loop.

The ifelse() Function

Although it is important to be able to create your own if/else statements and loops when you need to, R's vectorized nature means you can often avoid using such programming constructs. Whenever you want to perform the same operation to each object in a vector or other R data structure, there's often a way to do it in an efficient vectorized fashion without writing your own loops. For example, imagine you have a vector of numbers and you want to set all the negative values in the vector to zero. One way to do it is to use a for loop with an inner if statement:

```
my_vect <- runif(25, -1, 1)          # Generate some random data between -1 and 1

for (index in 1:length(my_vect) ) {   # loop through the sequence 1:25
    number <- my_vect[index]          # look up the next number using indexing
    if (number < 0) {                 # check if the number is less than 0
        my_vect[index] <- 0           # if so, set it to 0
    }
}

print (my_vect)
```

```
##  [1] 0.19204225 0.00000000 0.01582287 0.00000000 0.08693587 0.27935217
##  [7] 0.00000000 0.00000000 0.57706021 0.71756078 0.00000000 0.93283435
## [13] 0.31933466 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
## [19] 0.00000000 0.58837431 0.82274406 0.00000000 0.00000000 0.00000000
## [25] 0.29471476
```

Using a for loop requires writing quite a bit of code and loops are not particularly fast. Instead we could have used R's ifelse() function to the same thing in a vectorized manner. ifelse() takes a logical test as the first argument, a value to return if the test is true as the second argument and a value to return if the test is false as the third argument:

```
my_vect <- runif(25, -1, 1)           # Generate new random data between -1 and 1

my_vect <-   ifelse((my_vect < 0),    # A logical test
             0,                       # Value to set if the test is true
             my_vect)                 # Value to set if the test is false

print (my_vect)
```

```
##  [1] 0.672590818 0.237607671 0.208995213 0.303552852 0.155484701
##  [6] 0.582867802 0.000000000 0.000000000 0.989371949 0.000000000
## [11] 0.008186402 0.000000000 0.000000000 0.000000000 0.000000000
## [16] 0.000000000 0.208382419 0.313094973 0.000000000 0.000000000
## [21] 0.558702794 0.000000000 0.000000000 0.352448009 0.000000000
```

Since ifelse() is a vectorized function it provides a way to quickly apply a logical test to all values in a vector, change certain values based on the test outcome and keep other values the same. reference:http://hamelg.blogspot.com/