Programming in R

Olusoji Oluwafemi Daniel

Monday, April 27, 2015

Programming in R

In this lecture we will try to move readers from basic users of **R** to advance users. We are going to be examining some general programming concepts (Branching and Looping) and how they can be implemented in **R**. Writing functions will also be discussed and we will also introduce the concepts of bootstraping and simulations in **R**.

Branching in R

In programming, statements are executed line after line i.e. sequentially but there are times you want to skip some steps or you want your program to execute some statements based on some sort of decision, this concept is referred to as branching in programming parlance. **R** has 2 major commands that is used for branching, `if` and `switch` .

The if Statement

The `if` staement is one of the oldest means of branching in programming. It is implemented across all programming languages (all I have come across anyway) but with different syntax. There are three forms of `if` statements in **R**, we have the ordinary `if` , the `if else` statement and the `if else if` statement.

If you have programmed in `PHP` or `JAVA` the syntax in **R** is very similar to them. The syntax is;

if(conditional_expression)

{

```
valid_R_statements
```

}

Let us take a look at what makes up what is written up there.

- the keyword **if** starts the code
- the **conditional_expression** is a valid **R** expression that gives a logical answer, i.e. either **TRUE** or **FALSE**. These expressions are built using what is referred to as logical operators. Logical operators includes; **>** (greater than), **<** (less than), **==** (equal to), **>=** (greater than or equal to), **<=** (less than or equal to) and **!=** (not equal to).
- valid **R** statements inserted in curly brackets.

At this point, I will digress a bit into logical expressions and operators.

Example 2.1

Some logical expressions in **R**

```
x = 35
y = 50
# check if x > y
x > y
```

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

```
# check if x < y
x < y
```

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

```
# check if x = y
x == y
```

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

```
# check if x!=y
x != y
```

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

```
# check if 'a' > 'b'
'a' > 'b'
```

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

```
# check if 35 = -35
35 == -35
```

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

All statements above are executed in form of asking a question to which **R** provides a **TRUE** or **FALSE** answer. You are encourage to explore more on this expressions. Let us take a look at the an example where we supply 2 variables and the program returns the name of the variable with the greater value.

Example 2.2

Checking for the greater number between x and y

```
x <- 32
y <- 45
if(x > y){
    print('x > y')
} else{
  print('y > x')
}
```

```
## [1] "y > x"
```

The example above looks perfect but it isn’t really finished, this program wouldn’t print anything if the 2 variables being compared are equal (Oh! I didn’t see that as well, now you see it as well). Let us improve the code to take care of that.

Example 2.3

What of equality?

```
x <- 32
y <- 32
if(x > y){
    print('x > y')
  } else if(x < y) {
  print('y > x')
} else {
 print('x == y')
}
```

```
## [1] "x == y"
```

The `else` part of the example above isn’t necessary, the whole chunk will still function well without the `else` part of the code.

Another form of the `if else` statement is the `ifelse` function in **R**. This function takes in an argument that will resolve to a **TRUE** or **FALSE** then it returns either of the answers you provide for it, the first answer is the answer to return if the test results in a **TRUE** and the second is an answer ot return if the test results in a **FALSE**. The syntax of this function is `ifelse(test, yes, no)` . The `test` is the argument that should resolve to a logical answer, `yes` is the answer to return for **TRUE** and `no` is the answer to return for **FALSE**. Let us consider an example where you want **R** to return `Even` for even numbers and `Odd` for odd numbers between 10 and -9;

Example 2.4

Odd and Even Numbers!

```
nums <- seq(from = 10,to = -9,by = -1)
ifelse(nums%%2==0,'Even','Odd')
```

```
##  [1] "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"
## [11] "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"
```

You will notice that the roots of numbers from -1 to -9 are stored as NAs. Let me explain in details what the chunk of code above is doing;

- firstly, I created a sequence from 10 to -9 using a step of -1 (i.e. I reduced each new number by -1 till I arrived at -9),
- secondly, I used the ifelse function to check if each number in the `nums` vector to see if it is divisible by 2, print even if it is and odd if it is not.

Get Your Hands Dirty

Imagine we have nothing called complex numbers then square root of a negative number is something you would want to avoid, using the **seq**, **ifelse** and **sqrt** function only, try to find out the square root of only positive numbers between 50 and -50, return **NA** for the negative numbers.

The switch Statement

Well, if you are thinking that this staement works just like turning on and off a bulb switch, you are a little on track. The **switch** statement is an effective alternative to the **if else if** statement discussed in the previous section but with a little difference. unlike the **if else if** where logical expressions will evaluate to either of **TRUE** or **FALSE**, you are looking at an expression where an expression can result to several possible values.

Example 2.5

Some Real Analysis

```
x <- 38
y <- 22
z <- 100
switch(x+y < z,"both x and y are lesser than z","x added to y is greater han z")
```

```
## [1] "both x and y are lesser than z"
```

Although the `switch` statement can come in handy, it can be a little confusing for new users of **R** so I will encourage the use of the `if` till you get a good grasp of the `switch` statement(if you are using RStudio, type ?switch on the console to get the help page on `switch`).

Looping in R

Having discussed branching, the next is looping. Repeating operations is a common thing done in mathematics (especially in numerical mathematics). Hence, the idea of loop is very important. I must say, **R** looping structure is quite slow but there are faster and powerful loop functions that does what is required of a loop effectively (this will be discussed later). **R** has 3 loop construct, which are; `for` , `while` and `repeat` . Let us discuss each of this construct one after the other.

The for Loop

This is the most commonly used loop construct I have come across, it is quite easy to use and is best suited for processes that you have an idea of how many times it will execute, hence, it is a determinate loop. The syntax for the for loop is;

for(i_counter in vec)

{

valid_R_Statements to be repeated

}

Let's take a look at this codes in details;

- the keyword **for** followed by curved brackets
- **i_counter** is a variable that acts a counter and will not unlike every other langauge increase by 1, it will take on the next value in vec (you have to be careful about this if you have programmed in other languages before)
- curly braces with **R** statements in between. These statements will be repeated accoring to the length of vec.

I can bet this all sounds like some mumbo-jumbo, so let's some examples.

Example 2.6

Imagine you have a vector called `foo` with 5 elements which you will love print out

```
foo <- c(2,4,6,7)
for( i in foo){
  print(i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 7
```

You will notice that each element contained in `foo` was printed out in a column-wise manner.

Get Your Hands Dirty

What will happen if **print(i)** in the code above is replaced with **print(foo[i])**(Expect to see some NAs). *Hint: refer back to the first tutorial and think about positions of elements in a vector*

Let us take another look at the code above and see if the same thing can be done but this time in a different format.

Example 2.6, part2

Well, turns out we still want to print out the contents of `foo`

```
foo <- c(2,4,6,7)
for( i in 1:4){
  print(foo[i])
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 7
```

At this point, you should read with rapt attention because I know you are wondering, what is happening here!

In the first code, **i** is a counter but it is assuming the values inside `foo` , so what the loop did was to just take the values in `foo` assign them to **i** and orint them one after the other.

In the second code, **i** is a counter counting the numbers **1 - 4**, so **i** is just going to take the values 1,2,3 and 4, using this numbers we printed the values in `foo` by indexing. In a plain language, when **i** is 1, the code prints the value occupying position 1 inside `foo` (foo[1] = 2).

Example 2.7

The fibonacci sequence is some famous sequence in mathematics (I came across it during my calculus and trigonometry classes). it is a sequence where the first 2 elements are 1's and the next element is the sum of the previous two elements. Well, let's imagine we need the first 50 fibonacci elements.

```
#define an empty numeric vector to take in 100 integers
fibonacci <- integer(30)
#set the first two numbers in the fibonacci vector above to be 1
fibonacci[1] <- fibonacci[2] <- 1
for(i in 3:30)
  {
    fibonacci[i] <- fibonacci[i-2] + fibonacci[i-1]
  }
as.integer(fibonacci)
```

```
##  [1]      1      1      2      3      5      8     13     21     34     55
## [11]     89    144    233    377    610    987   1597   2584   4181   6765
## [21]  10946  17711  28657  46368  75025 121393 196418 317811 514229 832040
```