**Week 3 – Loops & If-else statements**

This week we will focus on the following:

1. Loops
2. Apply
3. if-else
4. UDFs (User Defined Functions)

**If – Else**

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are few points to keep in mind.

1. An if can have zero or one else and it must come after any else if's.
2. An if can have zero to many else if's and they must come before the else.
3. Once an else if succeeds, none of the remaining else if's or else's will be tested.

Base Example:

> y1 <- c('Apple',"Orange","Bananna")

>

> ### raw if-else example

> if("Mango" %in% y1) {

+ print("Mango is found")

+ } else {

+ print("Mango is not found")

+ }

[1] "Mango is not found"

>

> if("Mango" %in% y1) {

+ print("Mango is found")

+ } else if('Bananna' %in% y1){

+ print("Bananna is found")

+ } else {

+ print('Mango and Bananna is not found')

+ }

[1] "Bananna is found"

Functional Example (what we will use most often):

> ### functional if-else (we will use this for dataframes next week)

> ifelse('Mango' %in% y1 # condition

+ ,"Mango is found" # value if true

+ ,"Mango is not found" # value if false

+ )

[1] "Mango is not found"

>

> ifelse('Mango' %in% y1 # condition

+ ,"Mango is found" # value if true

+ ,ifelse('Bananna' %in% y1 # else if condition (if 1st is false)

+ ,'Bananna is found' # 2nd value if true

+ ,'Mango and Bananna is not found' # value if both are false

+ )

+ )

[1] "Bananna is found"

**Loops**

In computer Programming, a Loop is used to execute a group of instructions or a block of code multiple times, without writing it repeatedly. The block of code is executed based on a certain condition. Loops are the control structures of a program. Using Loops in computer programs simplifies rather optimizes the process of coding. There are three kinds of loops in R, for loops, while loops, and repeat loops.

**For Loops**

The for loop in R, also known as for cycle, is a repetitive iteration in loop of any code, where at each iteration some code is evaluated through the elements of a list or vector. This is less like the for keyword in other programming languages and works more like an iterator method as found in other object-orientated programming languages. With the for loop, we can execute a set of statements, once for each item in a vector, array, list, etc.

Examples:

**Basic syntax**

> for(i in 1:10) { # Head of for-loop

+

+ x1 <- i^2 # Code block

+ print(x1) # Print results

+ }

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

[1] 36

[1] 49

[1] 64

[1] 81

[1] 100

**Appending to a vector**

> # appending to a vector

> x3 <- numeric()

> for(i in 1:10) { # Head of for-loop

+ x3 <- c(x3, i^2) # Code block

+ }

> print(x3)

[1] 1 4 9 16 25 36 49 64 81 100

**Nested for loop – the wrong way and the right way (hashing)**

> # nested for loop (bad bad bad

> # --> need to using hashing to eliminate complexity)

> x4 <- character() # Create empty data object

> loop\_work <- 0

> for(i in 1:5) { # Head of first for-loop

+ for(j in 1:5) { # Head of nested for-loop

+ loop\_work <- loop\_work + 1

+ x4 <- c(x4, paste(LETTERS[i], letters[j], sep = "\_")) # Code block

+ }

+ }

> ### A better way using hashing --> this is on almost every tech interview

> hash\_table <- list()

> hash\_work <- 0

> x5 <- c()

> for( i in 1:5 ) {

+ hash\_work <- hash\_work + 1

+ hash\_table[[LETTERS[i]]] <- letters[1:5]

+ }

> for(j in 1:length(hash\_table)){

+ hash\_work <- hash\_work + 1

+ x5 <- c(x5,paste(names(hash\_table[j]),hash\_table[[j]],sep='\_'))

+ }

> print(hash\_work)

[1] 10

> print(loop\_work)

[1] 25

> #make sure we just made the same two vectors

> all(x4 == x5)

[1] TRUE

**Break statements**

> # for loop with break statement

> for(i in 1:10) { # Head of for-loop

+ x6 <- i^2 # Code block

+ print(x6) # Print results

+ if(i >= 5) { # Conditionally stop for-loop

+ break # Using break-statement

+ }

+ }

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

**Next statements**

> # for loop with next statement (skip)

> for(i in 1:10) { # Head of for-loop

+ if(i %in% c(1, 3, 5, 7, 9)) { # Conditionally skip iteration

+ next # Using next-statement

+ }

+ x7 <- i^2 # Code block

+ print(x7) # Print results

+ }

[1] 4

[1] 16

[1] 36

[1] 64

[1] 100

**Iterating over a data frame**

> # iterating over a dataframe

> iris\_new1 <- iris

> for(i in 1:ncol(iris\_new1)) { # Head of for-loop

+ if(grepl("Width", colnames(iris\_new1)[i])) { # Logical condition

+ iris\_new1[ , i] <- iris\_new1[ , i] + 1000 # Code block

+ }

+ }

> head(iris\_new1)

Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1 5.1 1003.5 1.4 1000.2 setosa

2 4.9 1003.0 1.4 1000.2 setosa

3 4.7 1003.2 1.3 1000.2 setosa

4 4.6 1003.1 1.5 1000.2 setosa

5 5.0 1003.6 1.4 1000.2 setosa

6 5.4 1003.9 1.7 1000.4 setosa

**While Loops**

While loops are used to iterate until a specific condition is meant. These loops can essentially run for ever, so be careful!

Example:

> i <- 1 # set the initial value

> while (i < 6) { # Head of while loop + test condition

+ print(i) # Code block

+ i = i+1 # Code block (make sure you add 1 or the condition will not be met!)

+ }

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

**Apply family of functions**

The apply family functions in R are a well-known set of R vectorized functions that allows you to perform complex tasks over arrays, avoiding the use of for loops.

The apply command in R allows you to apply a function across an array, matrix or data frame. You can do this in several ways, depending on the value you specify to the MARGIN argument, that can be normally set to 1, 2 or c(1, 2).

**apply**(X, *# Array, matrix or data frame*

MARGIN, *# 1: columns, 2: rows, c(1, 2): rows and columns*

FUN, *# Function to be applied*

...) *# Additional arguments to FUN*

**Applying a function to each row**

You can apply a function to every row of an array in R setting 1 as parameter of the MARGIN argument. For this first example we are going to apply the sum function over the data frame.

C

1 2 3 4 5

10.2 9.5 9.4 9.4 10.2

**Applying a function to each column**

Setting MARGIN = 2 will apply the function you specify to each column of the array you are working with.

> # col means for the 1st 4 columns of IRIS

> apply(iris[,0:4],MARGIN=2,FUN=mean)

Sepal.Length Sepal.Width Petal.Length Petal.Width

5.843333 3.057333 3.758000 1.199333

**Apply any function to all R data frame**

You can set the MARGIN argument to c(1, 2) or, equivalently, to 1:2 to apply the function to each value of the data frame.

> square <- function(x){

+ x^2

+ }

>

> # row & col custom function for the 1st 5 rows and 1st 4 columns of IRIS

> apply(iris[1:5,0:4],MARGIN=c(1,2),FUN=square)

Sepal.Length Sepal.Width Petal.Length Petal.Width

1 26.01 12.25 1.96 0.04

2 24.01 9.00 1.96 0.04

3 22.09 10.24 1.69 0.04

4 21.16 9.61 2.25 0.04

5 25.00 12.96 1.96 0.04

>

> iris[1:5,0:4]

Sepal.Length Sepal.Width Petal.Length Petal.Width

1 5.1 3.5 1.4 0.2

2 4.9 3.0 1.4 0.2

3 4.7 3.2 1.3 0.2

4 4.6 3.1 1.5 0.2

5 5.0 3.6 1.4 0.2

**User Defined Functions (UDFs)**

By now you have seen me create custom functions which we have used for some examples (just look above). Here we will talk about these UDFs and how to write them.

There are several parts of a function:

**Function name:** This is what you use when you call a function. For example, plot(my\_data) is a function with the name plot. You pass that function my\_data and it plots accordingly.

**function():** The function() is actually a function that allows you to create a function.

**Function Arguments**: An argument is what you pass the function. The function will take that object of value provided in the argument and use it to perform some task. In the example above, my\_data is actually an argument value. In the case of plot, my\_data provides that data that you wish to plot. main = in the plot function is the argument that allows you to pass a title to the plot.

**Documentation**: Documentation is not required for the function to work. However good documentation will save you time in the future when you need to use this code again.

Below is an example function. Notice that the part of the function that actually runs or evaluates things, is enclosed in curly braces {}.

> # this is an example function

> my\_sum <- function(num1, num2) {

+ # document your function here

+ # what the function does

+ # function inputs and outputs

+ some\_calculated\_output <- (num1 + num2 )

+ return(some\_calculated\_output) # return statement

+ }

>

> my\_sum(50,30)

[1] 80

**Default Values**

You can assign default values to arguments.

> # Using default values for arguments

> my\_sum2 <- function(num1=20, num2=30) {

+ # sum with default values for each input

+ some\_calculated\_output <- (num1 + num2 )

+ return(some\_calculated\_output) # return statement

+ }

>

> my\_sum2()

[1] 50

**Ensuring that inputs meet our criteria**

We can use an ifelse to ensure users supply the correct information while using **stop()** if the criteria are not meant.

> # Using an ifelse to ensure inputs are numeric

> my\_sum3 <- function(num1, num2=30) {

+ # sum two numbers

+ # if else to handle non-numeric values

+ if(is.numeric(num1)&is.numeric(num2)){

+ output <- (num1 + num2)

+ } else{

+ stop("ERROR: Both inputs must be numeric")

+ }

+

+ return(output) # return statement

+ }

>

> my\_sum3(num1="a",num2=4)

Error in my\_sum3(num1 = "a", num2 = 4) :

ERROR: Both inputs must be numeric

> my\_sum3(num1=3)

[1] 33