R Tutorial: Introduction to Loops (While, For, Repeat, Apply)

Our target for this tutorial is to provide the basic fundamentals of loops in R language to each reader. Hence, a little bit of a general introduction to this programming language becomes essential. So, let us first start with a basic introduction to R, and then we will carry on with the more specific learning of loops in R. To become a real master of R language, one article isn't enough; you might want to take a full **Introduction to R** tutorial that really teaches you all the nuts and bolts.

R Language Introduction

As you might be aware, R language is an open-source language for programmers, and the language is the product of collaborative evolution from a combination of brilliant minds too numerous to count. Maybe the best result of this is that R Language compilers have been designed keeping a newbie programmer in mind, which makes it an easy language to adapt. However, numerous minds bring numerous differences as well, and due to that, the language has been facing some critical issues in performance, which you will come to know more about from reading this tutorial.

Just like man pages facility in UNIX, R language has implemented the facility of the help command. Hence, if you face any issue related to any syntax while programming in R language, you can use the help command. Here is an example of help command. To receive help related to the solve command, you can type in:

```
> help(solve)
```

Additionally, windows version of R also offers the command like:

```
> ?solve
```

To receive the information on solve command in R language. That syntax works for other commands, too, not just solve.

One more advantage of R language is its facility to cater to a programmer with example programs. To refer to example programs on any topic related to R, you can use example command. i.e., if you want to look at the reference programs on solve command, you can type in something like:

```
> example(solve)
```

This will give you access to an example program, which can make your programming life much easier.

Loops in R

Mastering programming languages require a programmer to master the essential challenge of learning loops. Learning an advanced concept in any language is not possible without learning loops. And things are no different, even in the relatively simple R language. The R language has those traditional looping structures like the for loop and while loop, but that's not all.

Additionally, R has an unconventional Repeat Loop and a family of loops related to the Apply loop, which contains apply, sapply, lapply, and tapply. Use of traditional loops like for and while is for applying conventional repetitive transactions. However, the apply loop family has been designed to handle a few specific cases, i.e. tapply loop has been designed to handle the cases of ragged arrays. As in the other languages, it is quite possible for a user to use different loops inside each other in R language. I.e. a while loop can be used inside a for loop.

(This might seem like a lot to learn, but stick with it: once you've mastered the language, you can **Build Web Apps with R**.)

At present, there are very few articles available on the internet that can help you understand the loops of logic in R. Hence, we have taken up this task to describe each loop of R in detail. And we will first start with the conventional For Loop.

For Loop in R

If you are a programmer who develops software applications using different languages, you will have an idea about how a for loop works. For loop implementation of R language is not any different from the other language for loops.

Just like other programming languages, for loop in R language has been used to execute repetitive code statements for a particular number of times. Generally, a for loop construction looks something like:

```
for (vector counter)

{

Statements

}
```

To explain the concept of the for loop in detail, let us take an example of creating a for loop to square all the elements of a dataset sqr, which has all odd numbers in the range of 1 to 100. A notable thing of this example is the faster vectorizing.

```
sqr = seq(1, 100 by=2)

sqr.squared = NULL

for (n in 1:50)

{

sqr.squared[n] = sqr[n]^2

}
```

At any point of time, if your goal is to create a new vector, the first processing step shall be of setting up a vector to store member variables before creating a loop. sqr.squared = NULL statement basically depicts a set up of a vector.

This was a quite hard lesson that I learnt after spending about half an hour understanding my mistake. R basically does not like to operate upon a vector which does not have an existence.

Hence, we set up a vector to add the date in it at a later stage. Even though that's neither an efficient nor speedy way, it is a full-proof way for sure.

Once the vector is set up, the main task for setting up a for loop starts. A for loop in example is programmed to run 50 iterations, set up by variable 'n' in the example (You can program this iteration variable to anything you want). Square of nth (where n = 1, n =2 ... n = 50) number will be stored at the nth (where n = 1, n =2 ... n = 50) location in the new vector sqr.squared. Hence, for loop will run 50 iterations to store the squares of odd numbers in the range of 1 to 100 at 1 to 50th location in vector sqr.squared.

However, it may happen that you cannot handle the straight difference between the numbers of loops run against nth number of element of vector operated upon. I.e., once 7 loop iterations have been completed, the next iteration number would be 8 (since n = 8). However, 8th element of sqr vector will be sqr[8], which is 15. Hence, sqr.squared[8] should have the value 225, which is square of 15.

You can take a tutorial of **Data Analysis with R** for better understanding of vector processing in R language.

Debugging for loops in R Language

During programming of for loop in R language, if you come across any problem in the loop, you can consider below given points:

1. You may have reset a vector inside your loop. I.e. It is quite possible that you might have used a statement like "sqr.vector = NULL" within a loop instead of resetting it before starting your loop. This is a possible debugging point. Believe me, I spent more than an hour to figure out this mistake.

2. Did you miss out on subscribing a new vector? I.e. you may have put a statement like:

```
{sqr.squared = sqr[n]^2}
```

Didn't' understand your mistake? Let me explain it.

If you compare this statement with our example loop, you would understand that you have forgotten to put in your square brackets with a counter inside, at the left-hand operand in a statement. Hence, your new vector sqr.squared will always be filled with the last calculated value (since your new vector can only store one value).

Debugging experience of code in R language can also help you in **Building Web Apps using R language**. Having a practical goal like wanting to build a specific web app can be a great motivation to becoming a better coder.

Stop Aspect of For Loop

Let us take yet another example of for loop in R to understand the stop aspect of the loop.

```
n <- 1:5

r <- NULL

for (i in seq (along=n)) {

   if (n[i] <3) {

   r <- c(r, n[i]-1)

   } else {

   stop ("values shall be <3")

   }

}
```

Here is the output of the example for loop given above:

```
Error: values shall be <3

z

[1] 0 1
```

In the above example, for loop has been processed to store the value in vector n for 4 times. However, the limit of n has been set to 2, which allows storing the value only twice. In such conditions, stop statements can be used to display error messages inside a for loop. The above example gives a good explanation of use of stop statement inside for loop.

Advanced For Loop Concepts

Setting up Counter inside for loop:

Let's say, if you are creating a for loop inside a large program, number of iterations you want for loop to run may depend upon a vector length, which may have dependency on other factors as well. R language provides the facility to set up a loop counter inside the vector part of for loop, as shown below:

```
for (n in 0:length (sqr))

{

Statements to execute the loop sqr number of times

}
```

Well Constructed For Loop

As we discussed earlier, there are few performance related issues with for loops in R. If you are running in speed related troubles, you must follow the concept of well-constructed loops to get rid of such issues.

1. Try and get rid of as much code as possible inside the loop and take that out of the loop.

2. If you assess the vector operations possible prior to for loop, try to fetch them outside for loop.

You can learn more about well-constructed loops if you take up a **Basic R Language Tutorial**

Issues with Dynamic Memory Allocation

As we all know, the concept of dynamic memory allocation has its hazards in the programming world, and the same is applicable in the R language, particularly with the vectors.

Dynamic memory allocation to vectors is one of the important reasons that affect the performance of for loops with the use of such vectors. Basically, when you fill a vector with dynamic memory allocation inside for loop, the vector object grows by one with each passing loop iteration. A vector object's growth in each iteration takes its own time for loop to complete, which decreases the loop speed.

In our first example towards understanding of a for loop, we declared an empty vector sqr.squared to store squared values during the for loop iterations. Since this vector had a dynamic memory allocation, it grew by one object every time a for loop iteration occurred, affecting the performance of the loop.

The best solution to get rid of such speed issues is to predefine a size of vector and fill it up with the values inside for loop, whenever possible. I.e. Setting up a vector sqr.squared with the size of 50 (sqr.squared = numeric length = 50), created a vector with 50 zeros in it. Now, if the for loop is run upon this vector, it wouldn't grow with the iteration which can help us increase the loop speed.

It is quite possible that sometimes we may not be in a situation of making out a resultant size of a vector after its processing in for loop. In such cases, it is wise to define an upper-bound size of a vector and process it instead of dynamically allocating memory to such vectors. That would always prove faster than the dynamic memory allocation method. It is also possible to create a large vector with NA values and abandon them towards the end.

Assignment For You:

Create two different programs to use below given loops in each one of them and measure the speed difference between two of them:

```
Loop 1:

table = seq(1,100000, by=5)

table.squared = rep(NA, 100000)

for (n in 1:length(table) ) {

table.squared[n] = table[n]^2}

#get rid of excess NAs

table.squared = table.squared[!is.na(table.squared)]

summary(table.squared)

Loop 2:

table = seq(1, 100000, by=5)

table.squared = NULL
```

```
for (n in 1:length(table) ) {

table.squared[i] = table[n]^2}

summary(table.squared)
```

Controversies Of For Loop In R Language

However, execution of for loops in R has always been a controversial affair due to its speed issues. A programmer of any other programming language can take up the discussion of loop speed if the programmer wants to win the debate over languages.

However, one can also argue that R loop is written in C language (and few other variants of C++), underneath runs the faster C code, which can help R language achieve the results faster. And there are instances where programmers have managed to handle bigger data inside for loops of R. Even the use of apply loop family has been recommended at a few places (not always though), instead of for loops.

Points to ponder while implementing for loop in R

1. You must learn the **Basics of Loops in R** before starting to learn for loop in R
2. It is very important to initialize a vector outside for loop in which new data is to be filled inside for loop. Missing this condition can create unpredicted behaviours and eat up your time figuring out the mistake.
3. Never reset vectors inside the results. Such conditions are bound to produce false output.
4. Never miss subscribing a new vector. Else, you can have only last value filled in during the loop instead of a vector with numerous values.
5. Use stop statements to print error messages inside the loop.

Apply Loop Family in R Language

Writing traditional loops like for loop and while loop are easy while programming in an IDE. However, it is difficult to write these loops while writing the same code command line. Hence, R language has the facility to few functions that internally implement loops to make a programmer's life

easy. All these functions fall under the category of apply loop. Here is a brief introduction to each of these loops.

1. apply: Function to process array margins
2. lapply: Loop for a list, to evaluate a function on element
3. sapply: Similar to lapply and used for enhancing the results in simple format
4. tapply: function to process vector subsets

Going forward, we will understand each one of the above loops in detail with examples in this tutorial.

Learning the concept of Apply Loop family will also help you in **Data Analysis with R** Language.

Apply Loop

Generally, apply loop is used for evaluating a function (anonymous functions) over the array margins. Generally, apply loop is used to apply a loop over the matrix's columns and rows. Apply loop can be used over the arrays as well. I.e. it can be used to calculate the average of the matrices arrays. Writing apply loop function is not notably faster than writing an actual loop. However, apply loop can be written simply in one line.

As mentioned earlier, apply is a function, and it can simply be written in a function style. Here is a general denomination of apply function:

> str (apply)

function (Y, MAR, FUNCTION, ...)

Where, Arguments

- Y represents an array

- MAR represents an integer vector that indicates the margins to retain

- FUNCTION is a function or function name to be applied

- ... is for other arguments to be passed to FUNCTION

Let us now look at the example to understand more about apply loop:

Example:

i) Finding mean values

```
> y <- matrix (rnorm (100), 10, 5)

> apply (y, 2, mean)
```

Here, apply loop has been used to measure the mean values of matrix stored in y.

ii)

```
> y <- matrix (rnorm (100), 10, 5)

> apply(y, 1, sum)
```

Here, apply loop has been used to calculate the summation of matrix values stored in y.

You can type the above examples in your command lines to check the output.

Lapply Loop Family

The second loop from the apply family loop is lapply. Lapply is a loop defined for the list. I.e. it is used to evaluate the elements in the list.

Let us look at the general declaration of the lapply loop:

```
> lapply

function (Y, FUNCTION, ...)

{

FUNCTION <- match.fun(FUNCTION)

if (!is.vector(Y) || is.object(Y))

Y <- as.list(y)

.Internal(lapply(Y, FUN))

}
```

Actual looping of the lapply of R language is done in the C language code underneath. The output returned by lapply loop is a list, irrespective of the input class used. Let us look at the example of lapply functionality to understand it in detail:

i)

>y <- list(i = 1:5, n = rnorm(10))

> lapply (y, mean)

In this example, lapply has been used to extract the mean values from list hold by variable y. As mentioned earlier, output of lapply will always be list.

ii)

> y <- c(“cba”, “jklmno”, “a”, “hijz”)

> lapply (y, nchar)

In this example, lapply has been used to count the number of character from each string.

iii)

> y <- 1:8

> lapply (y, runif)

iv)

In this example, a function is written to extract the first column from the matrix.

> lapply (y, function(abc) abc[,1])

You can type these examples of lapply on your R language command prompt to check the output.

Sapply Loop:

The third loop from the apply family is sapply loop. As mentioned earlier, sapply loop is used to simplify the output of lapply loop. With lapply function, if returned result is a list with the element length of 1 for each, sapply will try and return the output in a vector form. In case if returned result is vector with the same length, sapply will return a matrix. However, if sapply loop cannot figure out the results, it would simply return a list like lapply loop.

In general form, sapply looks like:

> sapply

function (Y, FUNCTION, ...)

Let us take an example to understand the simplified result produced by sapply. To make it simpler, we will first look at the output of lapply function for a string and will look at the output of sapply function for the same string.

i)

With lapply:

y <- c(“cba”, “jklmno”, “a”, “hijz”)

> lapply (y, nchar)

Output:

[[1]]

[1] 3

[[2]]

[1] 6

[[3]]

With sapply:

y <- c(“cba”, “jklmno”, “a”, “hijz”)

> sapply(y, nchar)

Output:

cba jklmno a hijz

3 6 1 4

Tapply Loop:

Use of tapply loop is to process the vector subsets. However, tapply loop is also applied on the arrays with variable lengths (ragged arrays). Grouping of the tapply loop is defined by the factor.

Here is a general definition of a tapply loop:

> str(tapply)

function (Y, FACTOR, FUNCTION = NULL, ..., simplify = TRUE)

Where,

Y is a vector

FACTOR is a list of factors or a single factor

FUNCTION is a USED function

... represents the arguments to be supplied to function

Simplify depicts whether simplified results are required.

Let us look at the examples of tapply loop:

i)

> y <- a(rnorm(5), runif(5), rnorm(5, 1))

> b <- gl(3, 5)

> f

ii)

> tapply(y, b, mean)

iii)

Mean the group values without simplified results:

> tapply(y, b, mean, simplify = FALSE)

Points to Ponder During Implementation of Apply Loop Family:

1. Each loop in the apply loop family is represented as a function which in turn implements the loop.
2. Use of apply loop family is to avoid writing complex code related to loops on command prompt.

3. Use of a particular apply family loop depends upon the variable nature of input.

Repeat Loop In R Language

Just like for and while loop, repeat loop is one of the frequently used loops in the R language. As the name suggests, repeat loop is majorly used to execute the statement repetitively until a constraint condition is met like while loop. However, break statement is the only way to come out of the repeat loop (it is the only way to terminate the repeat loop).

A repeat loop looks something like this in general:

```
repeat

{

 //statements

 if (constraint condition)

 {

    break //breaks repeat loop

 }

}
```

Let us understand how repeat loop works:

1. A repeat statement at the start in above code is a repeat keyword that marks the start of a repeat loop.
2. //statements inside the curly brackets depicts the execution statements written inside the repeat loop.
3. If (constraint condition) defines if loop with the constraint condition.
4. Break statement inside if condition is used to break the repeat loop.
5. With the satisfaction of constraint condition, code control goes inside if condition has the break statement. Execution of the break statement breaks the repeat loop to terminate the loop.

Let us take an example for further understanding of repeat loop:

```
> sum <- 1

> repeat
```

```
{

sum <- sum + 2;

print(sum);

if (sum > 11)

break;

}
```

Output:

3

5

7

9

11

13

A program given in the example is designed to repetitively add 2 in the self-number until the resultant number reached 11.

Here, sum > 11 is a constraint condition which is used with if loop. If loop contains a break statement to break the repeat loop once the constraint condition is satisfied.

Usage Of Repeat Loop

Repeat loop has almost the same functionality as that of for loop and while loop. However, it is being used generally when constraint conditions are known to the programmer and the programmer does not want to go into the complexity of while and for loop.

In fact, repeat loop is used as a do-while loop at many places in R language. If you observe the general definition of repeat loop, it reflects the same meaning as of do-while loop, which says “execute statements till constraint condition is satisfied”. **Learning Data Analysis in R** language would make learning loops even easier for you.

Points to Ponder for Repeat Loop implementation:

1. Ensure that constraint condition variable used to break repeat loop is being processed inside the repeat loop.
2. Break statement implemented inside constraint conditioned if loop is the only way out from the repeat loop.
3. Repeat loop can be used as do-while loop in the R language programs.

While Loop In R:

While loop is one of the most simple loops used for repeated executions of statements. In R language, while loop is often used by the programmers. The structure of the while loop in R language is as simple as in other programming languages. The skeleton structure of a while loop consists of a constraint condition to start with. This condition is given after the keyword “while”. Programming statements are written inside while loop brackets, for which repetitive execution is to be carried out till it meets the constraint condition.

Let us look at the syntax used for while loop in R language:

The syntax is as:

```
while (constraint condition) // while is a keyword

  //returns bool (true/false) value

{ //opening curly brackets

//Statements

} // closing curly brackets
```

The output or the evaluation result of the given condition is a Boolean value that is either true or false. Based on this Boolean value, code control enters or quit the loop to execute the next statement. If the constraint condition is not fulfilled, code control goes inside the loop and executes the statements written in the while loop. An execution of a while loop is known as an iteration.

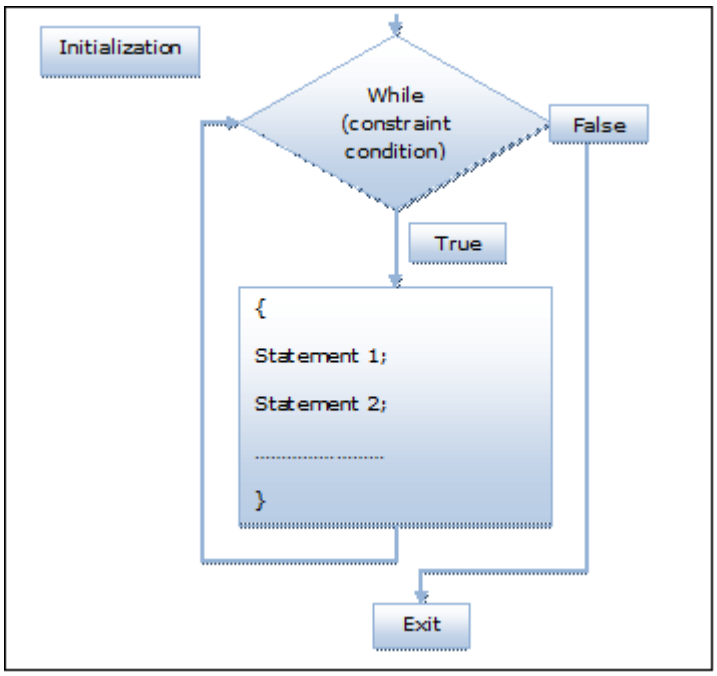
Once the iteration is completed, the control again goes back to the while keyword and checks the constraint condition. Based on the result, it jumps out the while loop or goes inside it to execute the statements in the loop.

Code snippet of while loop:

```
while (constraint condition )
{
        Statement 1;
        Statement 2;
        Statement 3;
}
```

Loop Body

A visual representation of while loop:



1. Initialization constraint variable:

Initialization is the first point from where the loop structure starts. In the above diagram, “Initialization” depicts the initialization of constraint condition. It is very much important to initialize a constraint variable before using it in a while loop. If not, it can even cause a program to crash or there can be a situation where while loop goes into the definite loop since constraint variable picks up a garbage value from the memory location it is defined at.

2. Condition check:

Constraint condition is a condition where the while loop is terminated. In the above diagram, the value of constraint condition is checked. If the constraint condition is not satisfied, code control enters inside the loop and

executes the statement inside the code. While loop iterations are executed until constraint condition is satisfied.

3. Termination of loop:

Termination is a state, where the constraint condition given is satisfied. Once the constraint condition is satisfied, the while loop terminates and code control stops the execution of while loop.

To get the deeper understanding of while loop in R language, let us look at a few examples.

The examples below are explained in the order of ascending order of complexity.

1. A code snippet in R using while loop to print the even numbers:

```
x<-0;

while (x < 10)

{

x<- x+4;

print (x);

}
```

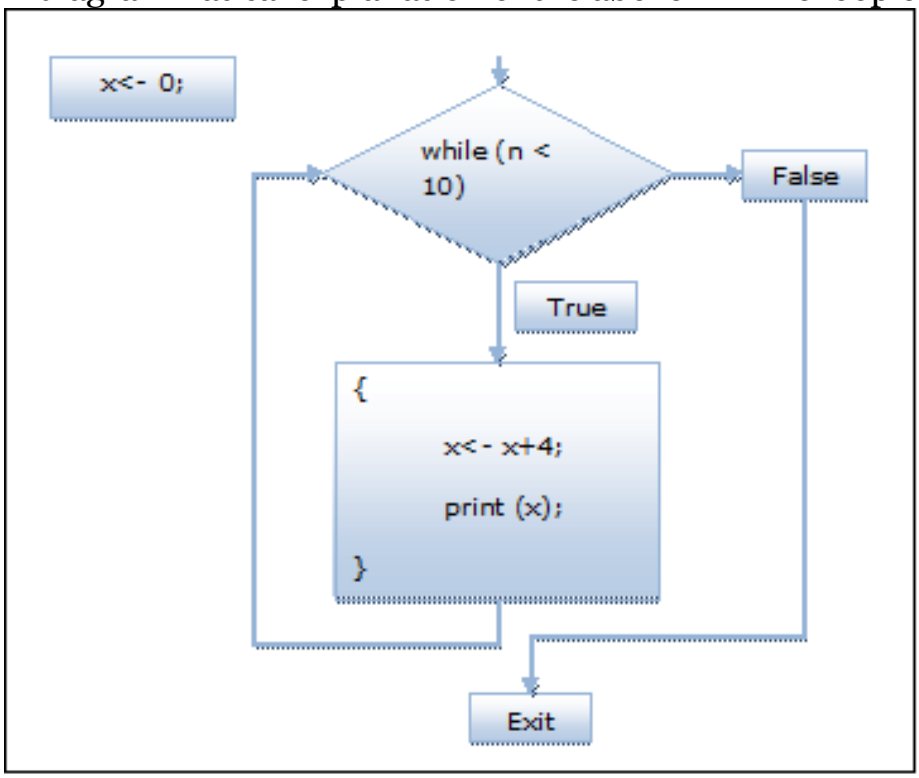
Answer:

4

8

12

A diagrammatical explanation of the above R while loop example



Here is the explanation of the above example:

In the above diagram, a statement x<-0 depicts the initialization of constraint variable. A statement while (x < 10) shows a while loop statement with a constraint condition where x < 10 servers as a constraint condition.

Here,

x < x+4 and

print (x) are the execution statements inside the while loop.

At the first iteration, the constraint condition will not be satisfied since x is initialized with 0 (zero) value. Hence, code control will go inside the while loop and execute the statements inside the while loop.

At the third iteration of while loop, value of x will become 12. Hence, on the fourth iteration of while loop, code control won't go inside the loop and will terminate the while loop.

Usage of While Loop with Unknown Conditions:

Sometimes, you may come across situations where you should apply a while loop at a place in the code. However, you will not know the exact constraint condition to apply due to the unknown nature of the result. In such cases, you can use the constraint condition “true” for the while loop. I.e. you can write your while loop as shown below:

```
While (true)

{

//Statements

}
```

While loop with constraint conditions “true” can be used in places of for loop with the dynamically allocated memory. This can in turn save the issue of for loop performance with dynamically allocated vectors.

Breaking While Loop

As we learnt, constraint condition is a way to break a while loop. However, that’s not the only way to break while loop in R. Usage of break statements or next statements can help with breaking while loops. Here is an example of breaking a while loop with break statement.

```
x<-0;

while (x < 10)

{

    x <- x + 4;

    print (x);

    if ( x = 8)

    {

break;

    }

}
```

In above example, code control would come out of the while loop as soon as the value of x becomes 8, which in above condition may become a2nd while loop iteration.

Here's the output of the above loop. With the use of break statement in this case, while loop will be terminated before its constraint condition is satisfied.

Output:

4

8

Error Conditions With While Loop

As per the expert programmers, constraint conditions use in the while loops are the very essential sources of error conditions in the while. Generally, the nature of errors related to constraint condition remains semantic, so it becomes difficult to detect such errors. Subsequently, they may produce false results as well. To avoid such errors, R language programmers must ensure that constraint variables are being processed inside the while loop. Such errors can even lead to infinite execution of while loop.

Having said that, it is important to mention that a while loop is a concept widely accepted as an option to execute infinite conditions in the real world. A real-time example of such conditions is a data server where infinite server execution is required.

In order to learn while loops in R language, you have to learn the fundamentals of R language. One way to do that is to take a tutorial in **Data Analysis with R.**

Important Points For While Loop Implementation

1. You must learn thebasics of loops in R before starting to learn the while loop in R language. It's an advanced concept, not the first thing you should try.
2. It is essential to initialize a constraint condition variable before execution of while loop. Otherwise, there are chances that a constraint variable will pick up a garbage value from allocated memory location and create an infinite while loop condition.
3. Please make sure that a constraint variable is being processed within the while loop to meet the constraint condition, to avoid the infinite execution of the while loop.

4. While loop is the widely accepted way of executing infinite loops in the real world.
5. Constraint condition is not the only way to come out of a while loop. Use of break conditions can also help break the while loop.

Improve Loop Speed Performances

Implementing loops with the large chunk of data can affect the performance of loops and slow down their iteration speed. However, there is a solution of getting rid of particular operations inside the loop or by avoiding the use of loops on the data intensive objects to overcome the limitation of loop speed performance.

Plan to avoid the use of loops on data intensive objects can be executed by majorly performing matrix- and vector-related computations inside the loops. Such computations run about 100 times quicker than the apply loop family and for loop in R language. In order to achieve it, the programmer can either use the existing optimization functions of R language like rowMeans, tabulate, table, or rowSums, or can design the user-defined functions that can avoid the use of R loops by replacing it with matrix- or vector-based methods. Alternatively, even a program can be written in C-level that can perform time-taking functions.

Summary:

If we summarize the concept of loops learnt in this article, we could make out that a lot of care has been taken during the implementation of R language to ensure that a programmer can have options of all type of loops during application implementation.

It is important to note that conditional statements like if, else, and next are quite an important part of loop implementation concept in R language. Hence, it is important for an R language programmer to learn conditional statements to implement the loops effectively. Maybe you'd like to **learn R by doing.**

15.1.2. if statements

The `if`/`else` statement conditionally evaluates two statements. There is a *condition* which is evaluated and if the *value* is `TRUE` then the first statement is evaluated; otherwise the second statement will be evaluated. The `if`/`else` statement returns, as its value, the value of the statement that was selected. The formal syntax is

```
if ( statement1 )
    statement2
else
    statement3
```

First, *statement1* is evaluated to yield *value1*. If *value1* is a logical vector with first element `TRUE` then *statement2* is evaluated. If the first element of *value1* is `FALSE` then *statement3* is evaluated. If *value1* is a numeric vector then *statement3* is evaluated when the first element of *value1* is zero and otherwise *statement2* is evaluated. Only the first element of *value1* is used. All other elements are ignored. If *value1* has any type other than a logical or a numeric vector an error is signalled.

`if`/`else` statements can be used to avoid numeric problems such as taking the logarithm of a negative number. Because `if`/`else` statements are the same as other statements you can assign the value of them. The two examples below are equivalent.

```
> if( any(x <= 0) ) y <- log(1+x) else y <- log(x)
> y <- if( any(x <= 0) ) log(1+x) else log(x)
```

The `else` clause is optional. The statement `if(any(x <= 0)) x <- x[x <= 0]` is valid. When the `if` statement is not in a block the `else`, if present, must appear on the same line as the end of*statement2*. Otherwise the new line at the end of *statement2* completes the `if` and yields a syntactically complete statement that is evaluated. A simple solution is to use a compound statement wrapped in braces, putting the `else` on the same line as the closing brace that marks the end of the statement.

`if`/`else` statements can be nested.

```
if ( statement1 ) {
    statement2
} else if ( statement3 ) {
    statement4
} else if ( statement5 ) {
    statement6
} else
    statement8
```

One of the even numbered statements will be evaluated and the resulting value returned. If the optional `else` clause is omitted and all the odd numbered *statement*s evaluate to `FALSE` no statement will be evaluated and `NULL` is returned.

The odd numbered *statement*s are evaluated, in order, until one evaluates to `TRUE` and then the associated even numbered *statement* is evaluated. In this

example, *statement6* will only be evaluated
if *statement1* is `FALSE` and *statement3* is `FALSE` and *statement5* is `TRUE`. There is no limit to the number of `else if` clauses that are permitted.

Conditional execution is available using the *if* statement and the corresponding *else* statement.

```
> x = 0.1

> if( x < 0.2)

  {

     x <- x + 1

     cat("increment that number!\n")

  }

increment that number!

> x

[1] 1.1
```

The else statement can be used to specify an alternate option. In the example below note that the *else* statement must be on the same line as the ending brace for the previous *if* block.

```
> x = 2.0

> if ( x < 0.2)

 {

    x <- x + 1

    cat("increment that number!\n")

 } else
```

```
 {
    x <- x - 1
    cat("nah, make it smaller.\n");
 }
nah, make it smaller.
> x
[1] 1
```

Finally, the *if* statements can be chained together for multiple options.
The *if* statement is considered a single code block, so more *if*statements can be added after the *else*.

```
> x = 1.0
> if ( x < 0.2)
 {
    x <- x + 1
    cat("increment that number!\n")
 } else if ( x < 2.0)
 {
   x <- 2.0*x
   cat("not big enough!\n")
 } else
 {
```

```
    x <- x - 1

    cat("nah, make it smaller.\n");

 }

not big enough!

> x

[1] 2
```

The argument to the *if* statement is a logical expression. A full list of logical operators can be found in the types document focusing on logical variables (*Logical*).