R Looping

Loops are used to repeat the process until the expression (condition) is TRUE. R uses three keywords **for, while and repeat** for looping purpose. **Next and break**, provide additional control over the loop.

The break statement exits the control from the innermost loop. The next statement immediately transfers control to return to the start of the loop and statement after next is skipped.

The value returned by a loop statement is always NULL and is returned invisibly.

R repeat

The repeat statement executes the body of the loop until a break occurs. Be careful while using it because of the infinite nature of the loop. We must use the break statement to terminate the loop. The syntax of the repeat loop is :, and JVM

Syntax

The basic syntax for creating a repeat loop in R:

repeat {

   commands

**if**(condition) {

**break**

   }

}

R While

While is another way to create a loop in R. It is very similar to the repeat loop. The syntax of the while loop is:

Syntax

The basic syntax for creating a while loop in R:

**while** (expression) {

   statement

}

In while loop, first, the expression is evaluated, and if it is TRUE, a block of statements execute. This process continues until the expression evaluated to FALSE.

R For loop

In R, for loop is a loop which is used to iterate the collection from the data structure like a vector. It uses **in** keyword to get individual elements from the vector. The syntax of the loop is:

Syntax

**for** ( data in data\_collection )

   statement1

This loop looks for each element in the data\_collection and store keep into a variable data. So, we can use that variable to get the stored value.

R repeat loop

A repeat loop is used to iterate a block of code. It is a special type of loop in which there is no condition to exit from the loop. For exiting, we include a break statement with a user-defined condition. This property of the loop makes it different from the other loops.

A repeat loop constructs with the help of the repeat keyword in R. It is very easy to construct an infinite loop in R.

The basic syntax of the repeat loop is as follows:

repeat {

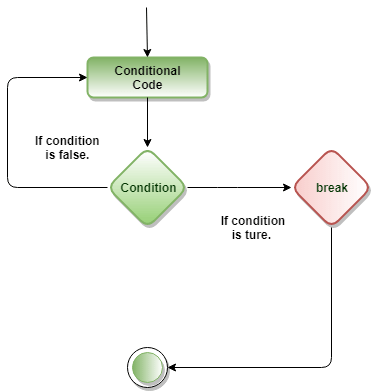
   commands

   if(condition) {

      break

   }

}

**Flowchart** 

1. First, we have to initialize our variables than it will enter into the Repeat loop.
2. This loop will execute the group of statements inside the loop.
3. After that, we have to use any expression inside the loop to exit.
4. It will check for the condition. It will execute a break statement to exit from the loop
5. If the condition is true.
6. The statements inside the repeat loop will be executed again if the condition is false.

**Example 1:**

v **<-** c("Hello","repeat","loop")

cnt **<-** 2

repeat {

   print(v)

    cnt **<-** cnt+1

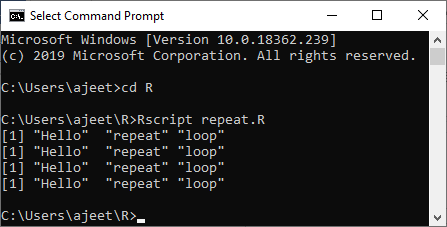
   if(cnt **>** 5) {

      break

   }

}

**Output**



R while loop

A while loop is a type of control flow statements which is used to iterate a block of code several numbers of times. The while loop terminates when the value of the Boolean expression will be false.

In while loop, firstly the condition will be checked and then after the body of the statement will execute. In this statement, the condition will be checked n+1 time, rather than n times.

The basic syntax of while loop is as follows:

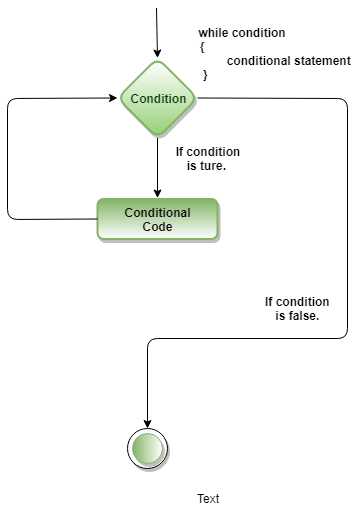
1. while (test\_expression) {
2. statement
3. }

**Flowchart**

12.7M

255

Java Try Catch



**Example 1:**

v **<-** c("Hello","while loop","example")

cnt **<-** 2

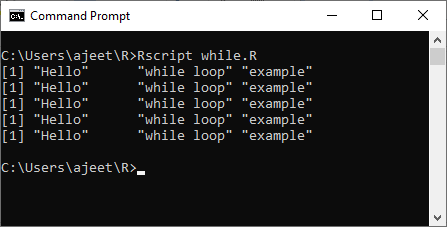
while (cnt **<** **7**) {

   print(v)

   cnt= cnt + 1

}}

**Output**



R For Loop

A for loop is the most popular control flow statement. A for loop is used to iterate a vector. It is similar to the while loop. There is only one difference between for and while, i.e., in while loop, the condition is checked before the execution of the body, but in for loop condition is checked after the execution of the body.

There is the following syntax of For loop in C/C++:

**for** (initialization\_Statement; test\_Expression; update\_Statement)

{

    // statements inside the body of the loop

}

How For loop works in C/C++?

The for loop in C and C++ is executed in the following way:

* The initialization statement of for loop is executed only once.
* After the initialization process, the test expression is evaluated. The for loop is terminated when the test expression is evaluated to false.
* The statements inside the body of for loop are executed, and expression is updated if the test expression is evaluated to true.
* The test expression is again evaluated.
* The process continues until the test expression is false. The loop terminates when the test expression is false.

For loop in R Programming

In R, a for loop is a way to repeat a sequence of instructions under certain conditions. It allows us to automate parts of our code which need repetition. In simple words, a for loop is a repetition control structure. It allows us to efficiently write the loop that needs to execute a certain number of time.Java Try Catch

In R, a for loop is defined as :

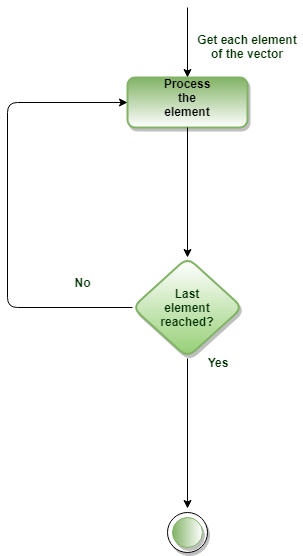
1. It starts with the keyword for like C or C++.
2. Instead of initializing and declaring a loop counter variable, we declare a variable which is of the same type as the base type of the vector, matrix, etc., followed by a colon, which is then followed by the array or matrix name.
3. In the loop body, use the loop variable rather than using the indexed array element.
4. There is a following syntax of for loop in R:

**for** (value in vector) {

   statements

}

**Flowchart**



**Example 1:** We iterate all the elements of a vector and print the current value.

# Create fruit vector

fruit <- c('Apple', 'Orange',"Guava", 'Pinapple', 'Banana','Grapes')

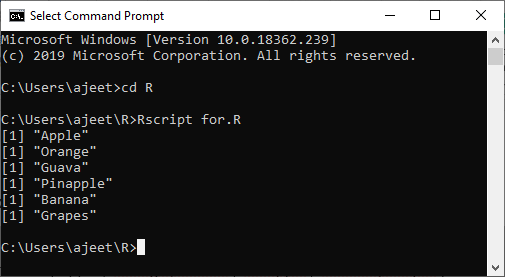
# Create the **for** statement

**for** ( i in fruit){

    print(i)

}

**Output**



**Vectorization Missing Values**

**Missing Values**

No matter how carefully we collect our data, there will always be situations where we don't know the value of a particular variable. For example, we might conduct a survey where we ask people 10 questions, and occasionally we forget to ask one, or people don't know the proper answer. We don't want values like this to enter into calculations, but we can't just eliminate them because then observations that have missing values won't "fit in" with the rest of the data.

In R, missing values are represented by the string *NA*. For example, suppose we have a vector of 10 values, but the fourth one is missing. I can enter a missing value by passing *NA* to the *c* function just as if it was a number (no quotes needed):

x = c(1,4,7,NA,12,19,15,21,20)

R will also recognize the unquoted string *NA* as a missing value when data is read from a file or URL.

Missing values are different from other values in R in two ways:

1. Any computation involving a missing value will return a missing value.
2. Unlike other quantities in R, we can't directly test to see if something is equal to a missing value with the equality operator (*==*). We must use the builtin *is.na* function, which will return *TRUE* if a value is missing and *FALSE* otherwise.

Here are some simple R statements that illustrate these points:

> x = c(1,4,7,NA,12,19,15,21,20)

> mean(x)

[1] NA

> x == NA

[1] NA NA NA NA NA NA NA NA NA

Fortunately, these problems are fairly easy to solve. In the first case, many functions (like *mean*, *min*, *max*, *sd*, *quantile*, etc.) accept an *na.rm=TRUE* argument, that tells the function to remove any missing values before performing the computation:

> mean(x,na.rm=TRUE)

[1] 12.375

In the second case, we just need to remember to always use *is.na* whenever we are testing to see if a value is a missing value.

> is.na(x)

[1] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE

By combining a call to *is.na* to the logical "not" operator (*!*) we can filter out missing values in cases where no *na.rm=* argument is available:

> x[!is.na(x)]

[1] 1 4 7 12 19 15 21 20

**Missing Values in R**

**Missing Values**

A missing value is one whose value is unknown. Missing values are represented in R by the NA symbol. NA is a special value whose properties are different from other values. NA is one of the very few reserved words in R: you cannot give anything this name. (Because R is case-sensitive, na and Na are okay to use, although I don't recommend them.) Missing values are often legitimate: values really are missing in real life. NAs can arise when you read in a Excel spreadsheet with empty cells, for example. You will also see NA when you try certain operations that are illegal or don't make sense. Here are some examples of operations that produce NA's.

> var (8) # Variance of one number

[1] NA

> as.numeric (c("1", "2", "three", "4")) # Illegal conversion

[1] 1 2 NA 4

Warning message:

NAs introduced by coercion

> c(1, 2, 3)[4] # Vector subscript out of range

[1] NA

> NA - 1 # Most operations on NAs produce NAs

[1] NA

> a <- data.frame (a = 1:3, b = 2:4)

> a[4,] # Data frame row subscript out of range

a b

NA NA NA # The first NA there is the row number

> a[,4] # Specifying a non-existent column just produces an error

Error in `[.data.frame`(a, , 4) : undefined columns selected

#

# Here's one that's particularly irksome

#

> a[1,2] <- NA # Suppose you have an NA in your dataframe...

> a[a$b < 4,] # ...and you try to index on that column

a b

NA NA NA # You get one of these NA rows for each NA in that column.

2 2 3

Note that if you specify a row number out of range for a *data frame*, that's not an error. You just get a row full of NAs. Interestingly, if you specify a row index of a *matrix*that's too big, you get a different respose altogether. That *is* an error. It's also an error to specify too big an index for a column of either a matrix or a data frame.

**Operations on Missing Values**

Almost every operation performed on an NA produces an NA. For example:

> x <- c(1, 2, NA, 4) # Set up a numeric vector

> x # There's an NA in there

[1] 1 2 NA 4

> x + 1 # NA + 1 = NA

[1] 2 3 NA 5

> sum(x) # This produces NA because we can't add NAs

[1] NA

> length(x) # This is okay

[1] 4

By the way, the default mode of NA is logical. That generally won't affect us.

**Detecting NAs**

You can't find missing values by looking at x == NA. Like most other functions, the == operator returns NA when either argument is NA. The is.na() function will find missing values for you: this function returns a logical vector the same length as its argument, with T for missing values and F for non-missings. It's fairly common to want to know the index of the missing values, and the which() function will help do this for you. For example:

> x # Here's my vector

[1] 1 2 NA 4

> is.na(x) # Is it NA?

[1] F F T F # Answer: no, no, yes, no.

> which (is.na(x)) # Which one is NA?

[1] 3 # Answer: the third one

To find all the rows in a data frame with at least one NA, try this:

> unique (unlist (lapply (your.data.frame, function (x) which (is.na (x)))))

lapply() applies the function to each column and returns a list whose i-th element is a vector containing the indices of the elements which have missing values in column i. unlist() turns that list into a vector and unique() gets rid of the duplicates.

**Ways to Exclude Missing Values**

Math functions generally have a way to exclude missing values in their calculations. mean(), median(), colSums(), var(), sd(), min() and max() all take the na.rm argument. When this is TRUE, missing values are omitted. The default is FALSE, meaning that each of these functions returns NA if any input number is NA. Note that cor() and its relatives don't work that way: with those you need to supply the use= argument. This is to permit more complicated handling of missing values than simply omitting them.

R's modeling functions accept an na.action argument that tells the function what to do when it encounters an NA. This causes the modeling function to call one of the *missing value filter* functions. These functions replace the original data set by a new data set in which the NAs have been altered. The default setting is na.omit, which excludes all rows with any missing values. An alternative is na.action=na.fail, which just stops when it encounters any missing values. This is useful if you didn't know you had any. The filter functions are:

* na.fail: Stop if any missing values are encountered
* na.omit: Drop out any rows with missing values anywhere in them and forgets them forever.
* na.exclude: Drop out rows with missing values, but keeps track of where they were (so that when you make predictions, for example, you end up with a vector whose length is that of the original response.)
* na.pass: Take no action.

A couple of other packages supply more alternatives:

* na.tree.replace (library (tree): For discrete variables, adds a new category called "NA" to replace the missing values.
* na.gam.replace (library gam): Operates on discrete variables like na.tree.replace(); for numerics, NAs are replaced by the mean of the non-missing entries.

Here are examples of these functions at work. You can call them directly, as I will do here, but they are also commonly used as values for the na.action= argument to the modeling functions.

#

# Set up a data frame, make a couple of elements NA.

#

> a <- data.frame (c1 = 1:8, c2 = factor (c("a", "b", "a", "c", "b", "c", "a", "b")))

> a[4,1] <- a[6,2] <- NA # This repeated assignment is legal and does what you expect.

> a

c1 c2

1 1 a

2 2 b

3 3 a

4 NA c

5 5 b

6 6 # Note the slightly different display of missings inside factors

7 7 a

8 8 b

> levels(a$c2) # Note the levels of c2 are "a," "b" and "c."

#NA is not a level.

[1] "a" "b" "c"

> na.fail (a) # Fails if NAs are present

Error in na.fail.default(a) : missing values in object

> na.exclude (a) # Omits rows with NAs in them

c1 c2

1 1 a

2 2 b

3 3 a

5 5 b

7 7 a

8 8 b

> na.gam.replace (a)

# Replace missing in c1 with the mean of the non-missings;

c1 c2 # Add a new level to c2

1 1.000000 a

2 2.000000 b

3 3.000000 a

4 4.571429 c

5 5.000000 b

6 6.000000 NA

7 7.000000 a

8 8.000000 b

> levels (na.gam.replace(a)$c2) # There's now a fourth level in that column

[1] "a" "b" "c" "NA"