

INTRODUCTION TO R (DATA VISUALIZATION)

19-Feb-2022

Dr. P. Rambabu, M. Tech., Ph.D., F.I.E.

Topics

1. Overview and about R
2. R Studio Installation
3. Fundamentals of R Programming
 - a) Data Structures and Data Types
 - b) Operators
 - c) Control Statements
 - d) Loop Statements
 - e) Functions
4. Descriptive Analysis using R
 - a) Maximum, Minimum, Range
 - b) Mean, Median and Mode
 - c) Variance, Standard Deviation
 - d) Quantiles, IQR
 - e) Summary

Introduction to R

“**R**” is a programming language and software environment for Statistical analysis, Graphics Representation and Reporting.

R was first implemented in the early 1990's by **Ross Ihaka** and **Robert Gentleman** at the University of Auckland, New Zealand, and it is currently developed by the R Development Core Team.

R is freely available under the GNU **General Public License**, and pre-compiled binary versions are provided for various **operating systems like Linux, Windows and Mac.**

Installation of R

Step 1: Go to [CRAN R](https://cran.r-project.org/) project website.

Step 2: Click on the **Download R** for Windows link.

Step 3: Click on the base subdirectory link or **install R** for the first time link.

Step 4: Click Download **R X.X.X** for Windows and save the executable .exe file.

Step 5: Run the **.exe** file and follow the installation instructions.

- a) Select the desired language and then click Next.
- b) Read the license agreement and click Next.
- c) Select the components to install (it is recommended to install all the components). Click Next.
- d) Enter/browse the folder/path you wish to install R into and then confirm by clicking Next.
- e) Select additional tasks like creating desktop shortcuts etc. then click Next.
- f) Wait for the installation process to complete.
- g) Click on Finish to complete the installation.

Installation of RStudio

Install RStudio on Windows

Step 1: With R-base installed, let's move on to installing RStudio. To begin, go to download RStudio and click on the download button for **RStudio desktop**.

Step 2: Click on the link for the windows version of RStudio and save the .exe file.

Step 3: **Run the .exe** and follow the installation instructions.

- a) Click Next on the welcome window.
- b) Enter/browse the path to the installation folder and click Next to proceed.
- c) Select the folder for the start menu shortcut or click on do not create shortcuts and then click Next.
- d) Wait for the installation process to complete.
- e) Click Finish to end the installation.

"Hello, World!" Program

Depending on the needs, you can program either at R command prompt or you can use an R script file to write your program.

```
# My first program in R Programming (using R Script File)
```

```
myString <- "Hello, World!"
```

```
print(myString)
```

Output:

```
[1] "Hello, World!"
```

R does not support multi-line comments but you can perform a trick which is something as follows:

```
if (FALSE)
```

```
{
```

```
  "This is a demo for multi-line comments and it should be put inside either a single OR double quote"
```

```
}
```

Variable:

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name, var.name	valid	Can start with a dot(.) but the dot(.) should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

Variable Assignment:

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using **print()** or **cat()** function. The **cat()** function combines multiple items into a continuous print output.

```
# Assignment using equal operator.
var.1 = c(0,1,2,3)
# Assignment using leftward operator.
var.2 <- c("learn","R")
# Assignment using rightward operator.
c(TRUE,1) -> var.3

print(var.1)
cat ("var.1 is ", var.1 ,"\n")
cat ("var.2 is ", var.2 ,"\n")
cat ("var.3 is ", var.3 ,"\n")
```

When we execute the above code, it produces the following result -

```
[1] 0 1 2 3
var.1 is  0 1 2 3
var.2 is  learn R
var.3 is  1 1
```

Note - The vector `c(TRUE,1)` has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable:

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x <- "Hello"
cat("The class of var_x is ",class(var_x),"\n")

var_x <- 34.5
cat(" Now the class of var_x is ",class(var_x),"\n")

var_x <- 27L
cat(" Next the class of var_x becomes ",class(var_x),"\n")
```

When we execute the above code, it produces the following result –

```
The class of var_x is character
Now the class of var_x is numeric
Next the class of var_x becomes integer
```

Data Structures

In contrast to other programming languages like C and java in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are –

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type

Example

- | | |
|--------------|-------------------------------------|
| 1. Logical | True, False |
| 2. Numerical | 12.3, 5, 99 |
| 3. Integer | 2L, 34L, 0L |
| 4. Complex | 3 + 2i |
| 5. Character | 'a' , "good", "TRUE", '23.4' |
| 6. Raw | "Hello" is stored as 48 65 6c 6c 6f |

Data Type	Example	Verify
Logical	TRUE, FALSE	<div> <div> v <- TRUE print(class(v)) </div> <div> Live Demo </div> </div> <p>it produces the following result -</p> <pre>[1] "logical"</pre>
Numeric	12.3, 5, 999	<div> <div> v <- 23.5 print(class(v)) </div> <div> Live Demo </div> </div> <p>it produces the following result -</p> <pre>[1] "numeric"</pre>
Integer	2L, 34L, 0L	<div> <div> v <- 2L print(class(v)) </div> <div> Live Demo </div> </div> <p>it produces the following result -</p> <pre>[1] "integer"</pre>

Complex	$3 + 2i$	<pre>v <- 2+5i print(class(v))</pre> <p>Live Demo ↗</p> <p>it produces the following result -</p> <pre>[1] "complex"</pre>
Character	'a', "good", "TRUE", '23.4'	<pre>v <- "TRUE" print(class(v))</pre> <p>Live Demo ↗</p> <p>it produces the following result -</p> <pre>[1] "character"</pre>
Raw	"Hello" is stored as 48 65 6c 6c 6f	<pre>v <- charToRaw("Hello") print(class(v))</pre> <p>Live Demo ↗</p> <p>it produces the following result -</p> <pre>[1] "raw"</pre>

Vectors

In R programming, the very basic data types are the R-objects called **vectors** which hold elements of different classes (Data Types).

Vectors:

When you want to create vector with more than one element, you should use `c()` function which means to combine the elements into a vector.

```
# Create a vector.  
apple <- c('red','green',"yellow")  
print(apple)  
  
# Get the class of the vector.  
print(class(apple))
```

[Live Demo](#)

When we execute the above code, it produces the following result –

```
[1] "red"    "green"  "yellow"  
[1] "character"
```

Vector Manipulation

Vector Arithmetic

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

```
# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)

# Vector addition.
add.result <- v1+v2
print(add.result)
# Vector subtraction.
sub.result <- v1-v2
print(sub.result)
# Vector multiplication.
multi.result <- v1*v2
print(multi.result)
# Vector division.
divi.result <- v1/v2
print(divi.result)
```

When we execute the above code, it produces the following result –

```
[1] 7 19 4 13 1 13
[1] -1 -3 4 -3 -1 9
[1] 12 88 0 40 0 22
[1] 0.7500000 0.7272727 Inf 0.6250000 0.0000000 5.5000000
```

Vector Manipulation

Vector Element Sorting

Elements in a vector can be sorted using the **sort()** function.

```
v <- c(3,8,4,5,0,11, -9, 304)
# Sort the elements of the vector.
sort.result <- sort(v)
print(sort.result)

# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)

# Sorting character vectors.
v <- c("Red", "Blue", "yellow", "violet")
sort.result <- sort(v)
print(sort.result)

# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
```

When we execute the above code, it produces the following result -

```
[1] -9 0 3 4 5 8 11 304
[1] 304 11 8 5 4 3 0 -9
[1] "Blue" "Red" "violet" "yellow"
[1] "yellow" "violet" "Red" "Blue"
```


Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.  
list1 <- list(c(2,5,3),21.3,sin)  
  
# Print the list.  
print(list1)
```

When we execute the above code, it produces the following result –

```
[[1]]  
[1] 2 5 3  
  
[[2]]  
[1] 21.3  
  
[[3]] function (x) .Primitive("sin")
```

Naming List Elements

The list elements can be given names and they can be accessed using these names.

```
# Create a list containing a vector, a matrix and a list.  
list_data <- list(c("Jan", "Feb", "Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),  
  list("green", 12.3))  
  
# Give names to the elements in the list.  
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")  
  
# Show the list.  
print(list_data)
```

When we execute the above code, it produces the following result –

```
$`1st_Quarter`  
[1] "Jan" "Feb" "Mar"
```

```
$A_Matrix  
      [,1] [,2] [,3]  
[1,]    3    5  -2  
[2,]    9    1    8
```

```
$A_Inner_list  
$A_Inner_list[[1]]  
[1] "green"
```

```
$A_Inner_list[[2]]  
[1] 12.3
```

Manipulating Lists

We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan", "Feb", "Mar"),
  matrix(c(3,9,5,1,-2,8), nrow = 2),
  list("green", 12.3))
# Give names to the elements in the list.
names(list_data) <- c("1st Quarter",
  "A_Matrix", "A Inner list")
# Add element at the end of the list.
list_data[4] <- "New element"
print(list_data[4])

# Remove the last element.
list_data[4] <- NULL
# Print the 4th Element.
print(list_data[4])
# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

When we execute the above code, it produces the following result -

```
[[1]]
[1] "New element"

$<NA>
NULL

$`A Inner list`
[1] "updated element"
```

Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion, we use the **unlist()** function. It takes the list as input and produces a vector.

```
# Create lists.  
list1 <- list(1:5)  
print(list1)  
  
list2 <-list(10:14)  
print(list2)  
  
# Convert the lists to vectors.  
v1 <- unlist(list1)  
v2 <- unlist(list2)  
print(v1)  
print(v2)  
  
# Now add the vectors  
result <- v1+v2  
print(result)
```

When we execute the above code, it produces the following result –

```
[[1]]  
[1] 1 2 3 4 5  
  
[[1]]  
[1] 10 11 12 13 14  
  
[1] 1 2 3 4 5  
[1] 10 11 12 13 14  
[1] 11 13 15 17 19
```

Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.  
list1 <- list(1,2,3)  
list2 <- list("Sun","Mon","Tue")  
  
# Merge the two lists.  
merged.list <- c(list1,list2)  
  
# Print the merged list.  
print(merged.list)
```

When we execute the above code, it produces the following result -

```
[[1]]  
[1] 1  
  
[[2]]  
[1] 2  
  
[[3]]  
[1] 3  
  
[[4]]  
[1] "Sun"  
  
[[5]]  
[1] "Mon"  
  
[[6]]  
[1] "Tue"
```

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.  
M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)  
print(M)
```

When we execute the above code, it produces the following result –

```
      [,1] [,2] [,3]  
[1,] "a"  "a"  "b"  
[2,] "c"  "b"  "a"
```

Accessing Elements of Matrix

Elements of a matrix can be accessed by using the column and row index of the element. We consider the matrix P above to find the specific elements below

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix.
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames
= list(rownames, colnames))

# Access the element at 3rd column and 1st row.
print(P[1,3])

# Access the element at 2nd column and 4th row.
print(P[4,2]) # Access only the 2nd row.
print(P[2,])

# Access only the 3rd column.
print(P[,3])
```

When we execute the above code, it produces the following result -

```
[1] 5
[1] 13
col1 col2 col3
  6    7    8
row1 row2 row3 row4
  5    8   11   14
```

Matrix Computations

Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix. The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

Matrix Addition & Subtraction

```
# Create two 2x3 matrices.  
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)  
print(matrix1)  
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)  
print(matrix2)  
  
# Add the matrices.  
result <- matrix1 + matrix2  
cat("Result of addition","\n")  
print(result)  
  
# Subtract the matrices  
result <- matrix1 - matrix2  
cat("Result of subtraction","\n")  
print(result)
```

	[,1]	[,2]	[,3]
[1,]	3	-1	2
[2,]	9	4	6

	[,1]	[,2]	[,3]
[1,]	5	0	3
[2,]	2	9	4

Result of addition

	[,1]	[,2]	[,3]
[1,]	8	-1	5
[2,]	11	13	10

Result of subtraction

	[,1]	[,2]	[,3]
[1,]	-2	-1	-1
[2,]	7	-5	2

When we execute the above code, it produces the following result –

Matrix Computations

Matrix Multiplication & Division

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)

# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
print(result)
```

When we execute the above code, it produces the following result –

```
      [,1] [,2] [,3]
[1,]    3  -1    2
[2,]    9   4    6

      [,1] [,2] [,3]
[1,]    5   0    3
[2,]    2   9    4
Result of multiplication
      [,1] [,2] [,3]
[1,]   15   0    6
[2,]   18  36   24
Result of division
      [,1]      [,2]      [,3]
[1,]   0.6      -Inf  0.6666667
[2,]   4.5  0.4444444  1.5000000
```

Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.  
a <- array(c('green','yellow'),dim = c(3,3,2))  
print(a)
```

When we execute the above code, it produces the following result:

, , 1

	[,1]	[,2]	[,3]
[1,]	"green"	"yellow"	"green"
[2,]	"yellow"	"green"	"yellow"
[3,]	"green"	"yellow"	"green"

, , 2

	[,1]	[,2]	[,3]
[1,]	"yellow"	"green"	"yellow"
[2,]	"green"	"yellow"	"green"
[3,]	"yellow"	"green"	"yellow"

Factors

Factors are the R-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modeling.

Factors are created using the **factor()** function. The **nlevels** function gives the count of levels.

```
# Create a vector.
apple_colors <-
  c('green','green','yellow','red','red','red','green')

# Create a factor object.
factor_apple <- factor(apple_colors)

# Print the factor.
print(factor_apple)
print(nlevels(factor_apple))
```

When we execute the above code, it produces the following result:

```
[1] green green yellow red red red green
Levels: green red yellow
[1] 3
```

Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the **data.frame()** function.

```
# Create the data frame.  
BMI <- data.frame( gender = c("Male", "Male","Female"), height = c(152, 171.5, 165),  
                  weight = c(81,93, 78), Age = c(42,38,26) )  
  
print(BMI)
```

When we execute the above code, it produces the following result -

	gender	height	weight	Age
1	Male	152.0	81	42
2	Male	171.5	93	38
3	Female	165.0	78	26

Operators



An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operators

We have the following types of operators in R programming –

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Miscellaneous Operators

Operator	Description	Example
+	Adds two vectors	<div> Live Demo </div> <pre>v <- c(2,5.5,6) t <- c(8, 3, 4) print(v+t)</pre> <p>it produces the following result -</p> <pre>[1] 10.0 8.5 10.0</pre>
-	Subtracts second vector from the first	<div> Live Demo </div> <pre>v <- c(2,5.5,6) t <- c(8, 3, 4) print(v-t)</pre> <p>it produces the following result -</p> <pre>[1] -6.0 2.5 2.0</pre>

•	Multiplies both vectors	<div data-bbox="864 56 1665 212"><pre>v <- c(2,5.5,6)</pre><pre>t <- c(8, 3, 4)</pre><pre>print(v*t)</pre><div data-bbox="1437 56 1619 88">Live Demo </div></div> <p data-bbox="864 243 1365 290">it produces the following result -</p> <div data-bbox="864 326 1665 440"><pre>[1] 16.0 16.5 24.0</pre></div>
/	Divide the first vector with the second	<div data-bbox="864 543 1665 699"><pre>v <- c(2,5.5,6)</pre><pre>t <- c(8, 3, 4)</pre><pre>print(v/t)</pre><div data-bbox="1437 543 1619 574">Live Demo </div></div> <p data-bbox="864 730 1665 828">When we execute the above code, it produces the following result -</p> <div data-bbox="864 865 1665 979"><pre>[1] 0.250000 1.833333 1.500000</pre></div>

%%

Give the remainder of the first vector with the second

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v%%t)
```

[Live Demo](#)

it produces the following result -

```
[1] 2.0 2.5 2.0
```

/%

The result of division of first vector with second (quotient)

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v/%t)
```

[Live Demo](#)

it produces the following result -

```
[1] 0 1 1
```


A

The first vector raised to the exponent of second vector

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v^t)
```

[Live Demo](#) 

it produces the following result -

```
[1] 256.000 166.375 1296.000
```

Relational Operators:

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	<pre>v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v>t)</pre> <p>it produces the following result -</p> <pre>[1] FALSE TRUE FALSE FALSE</pre>
<	Checks if each element of the first vector is less than the corresponding element of the second vector.	<pre>v <- c(2,5.5,6,9) t <- c(8,2.5,14,9) print(v < t)</pre> <p>it produces the following result -</p> <pre>[1] TRUE FALSE TRUE FALSE</pre>

==

Checks if each element of the first vector is equal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9) Live Demo  
t <- c(8,2.5,14,9)  
print(v == t)
```

it produces the following result -

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

<=

Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9) Live Demo  
t <- c(8,2.5,14,9)  
print(v<=t)
```

it produces the following result -

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

>=

Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9) Live Demo  
t <- c(8,2.5,14,9)  
print(v>=t)
```

it produces the following result -

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

!=

Checks if each element of the first vector is unequal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9) Live Demo  
t <- c(8,2.5,14,9)  
print(v!=t)
```

it produces the following result -

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

Logical Operators:

It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	<pre>v <- c(3,1,TRUE,2+3i) t <- c(4,1,FALSE,2+3i) print(v&t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE TRUE FALSE TRUE</pre>

|

It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.

```
v <- c(3,0,TRUE,2+2i) Live Demo  
t <- c(4,0,FALSE,2+3i)  
print(v|t)
```

it produces the following result -

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

!

It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.

```
v <- c(3,0,TRUE,2+2i) Live Demo  
print(!v)
```

it produces the following result -

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

Miscellaneous Operators:

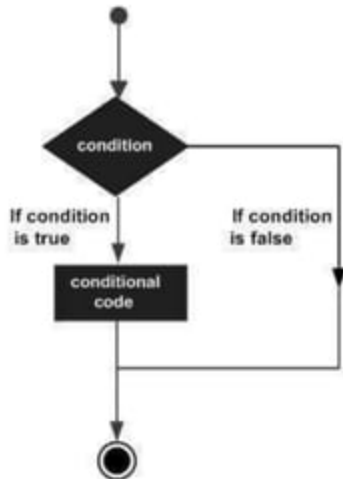
These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example
:	Colon operator. It creates the series of numbers in sequence for a vector.	<pre>v <- 2:8</pre> <pre>print(v)</pre> <p>it produces the following result -</p> <pre>[1] 2 3 4 5 6 7 8</pre> <p>Live Demo</p>
%in%	This operator is used to identify if an element belongs to a vector.	<pre>v1 <- 8</pre> <pre>v2 <- 12</pre> <pre>t <- 1:10</pre> <pre>print(v1 %in% t)</pre> <pre>print(v2 %in% t)</pre> <p>it produces the following result -</p> <pre>[1] TRUE</pre> <pre>[1] FALSE</pre> <p>Live Demo</p>

Control Statements

Control Statements or Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

Following is the general form of a typical decision making structure found in most of the programming languages –



R provides the following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	<p>if statement ↗</p> <p>An if statement consists of a Boolean expression followed by one or more statements.</p>
2	<p>if...else statement ↗</p> <p>An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.</p>
3	<p>switch statement ↗</p> <p>A switch statement allows a variable to be tested for equality against a list of values.</p>

If Statement:

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

The basic syntax for creating an **if** statement in R is -

```
if(boolean_expression) {  
  // statement(s) will execute if the boolean expression is true.  
}
```

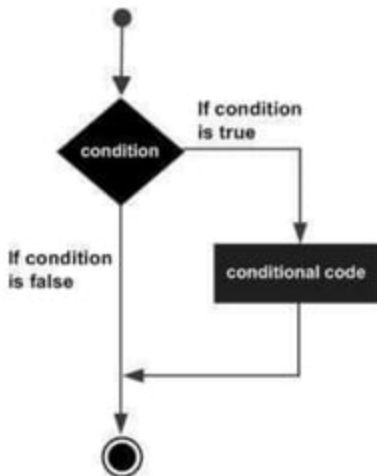
If the Boolean expression evaluates to be **true**, then the block of code inside the if statement will be executed. If Boolean expression evaluates to be **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Example

```
x <- 30L  
  
if(is.integer(x))  
{  
  print("X is an Integer")  
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "X is an Integer"
```



If ... Else .. Statement:

An **if** statement can be followed by an optional **else** statement which executes when the boolean expression is false.

Syntax

The basic syntax for creating an **if...else** statement in R is –

```
if(boolean_expression) {  
  // statement(s) will execute if the boolean expression is true.  
} else {  
  // statement(s) will execute if the boolean expression is false.  
}
```

If the Boolean expression evaluates to be **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

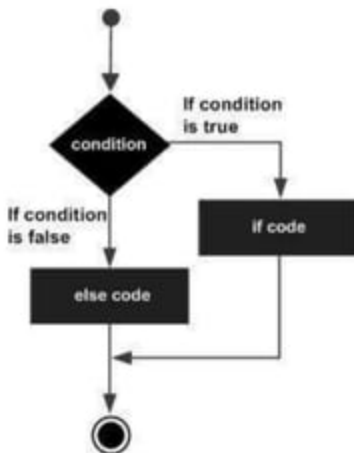
Example

```
x <- c("what","is","truth")

if("Truth" %in% x) {
  print("Truth is found")
} else {
  print("Truth is not found")
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "Truth is not found"
```



The if...else if...else Statement

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.

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

```
x <- c("what","is","truth")

if("Truth" %in% x) {
  print("Truth is found the first time")
} else if ("truth" %in% x) {
  print("truth is found the second time")
} else {
  print("No truth found")
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "truth is found the second time"
```

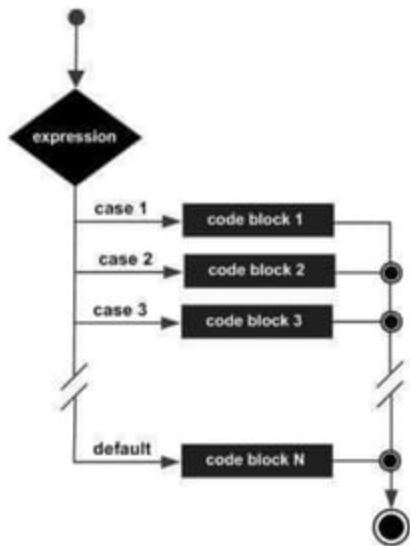
Switch Statement:

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

```
x <- switch(  
  3,  
  "first",  
  "second",  
  "third",  
  "fourth"  
)  
  
print(x)
```

When the above code is compiled and executed, it produces the following result -

```
[1] "third"
```

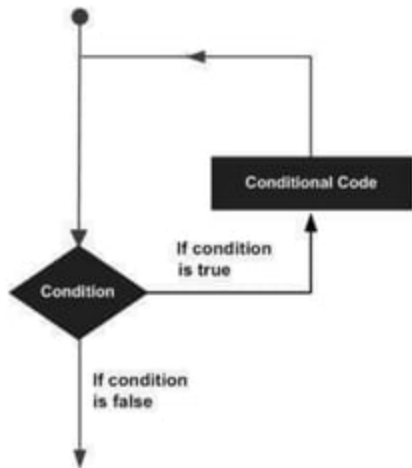





Loop Statements

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages –



Sr.No.	Loop Type & Description
1	<p>repeat loop </p> <p>Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.</p>
2	<p>while loop </p> <p>Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.</p>
3	<p>for loop </p> <p>Like a while statement, except that it tests the condition at the end of the loop body.</p>

Repeat Loop:

The Repeat loop executes the same code again and again until a stop condition is met.

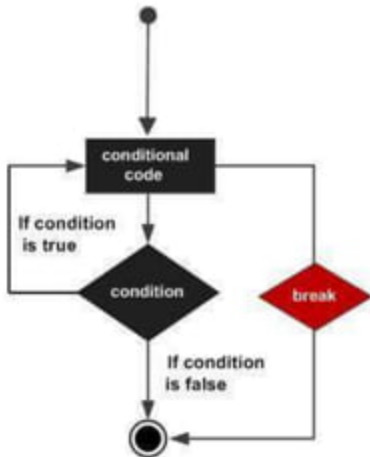
```
v <- c("Hello","loop")
cnt <- 2

repeat {
  print(v)
  cnt <- cnt+1

  if(cnt > 5) {
    break
  }
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```



While Loop:

The While loop executes the same code again and again until a stop condition is met.

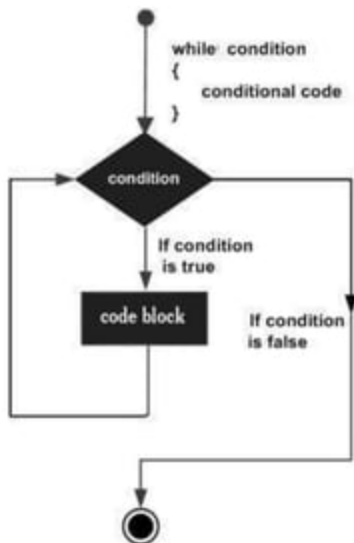
Here key point of the **while** loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

```
v <- c("Hello","while loop")
cnt <- 2

while (cnt < 7) {
  print(v)
  cnt = cnt + 1
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
[1] "Hello" "while loop"
```



For Loop:

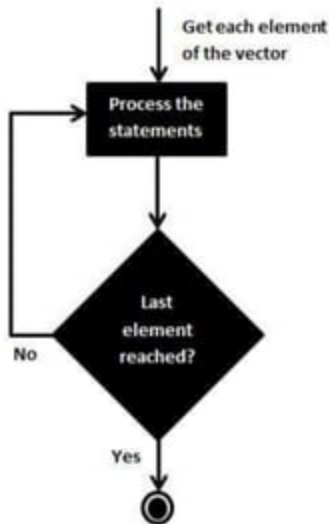
A **For loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

```
v <- LETTERS[1:4]

for ( i in v ) {
  print(i)
}
```

When the above code is compiled and executed, it produces the following result –

```
[1] "A"
[1] "B"
[1] "C"
[1] "D"
```



Functions

A **function** is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows -

```
function_name <- function(arg_1, arg_2, ...) {  
  Function body  
}
```

Function Components

The different parts of a function are –

1. **Function Name** – This is the actual name of the function. It is stored in R environment as an object with this name.
2. **Arguments** – An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
3. **Function Body** – The function body contains a collection of statements that defines what the function does.
4. **Return Value** – The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

Built-in Function:

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc.

They are directly called by user written programs.

```
# Create a sequence of numbers from 32 to 44.  
print(seq(32,44))  
  
# Find mean of numbers from 25 to 82.  
print(mean(25:82))  
  
# Find sum of numbers from 41 to 68.  
print(sum(41:68))
```

When we execute the above code, it produces the following result –

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44  
[1] 53.5  
[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}
```

Calling a Function

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

When we execute the above code, it produces the following result -

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```


Calling a Function without an Argument

[Live Demo](#)

```
# Create a function without an argument.  
new.function <- function() {  
  for(i in 1:5) {  
    print(i^2)  
  }  
}  
  
# Call the function without supplying an argument.  
new.function()
```

When we execute the above code, it produces the following result –

```
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.  
new.function <- function(a,b,c) {  
  result <- a * b + c  
  print(result)  
}  
  
# Call the function by position of arguments.  
new.function(5,3,11)  
  
# Call the function by names of the arguments.  
new.function(a = 11, b = 5, c = 3)
```

[Live Demo](#)

When we execute the above code, it produces the following result -

```
[1] 26  
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
```

```
new.function <- function(a = 3, b = 6) {  
  result <- a * b  
  print(result)  
}
```

[Live Demo](#)

```
# Call the function without giving any argument.
```

```
new.function()
```

```
# Call the function with giving new values of the argument.
```

```
new.function(9,5)
```

When we execute the above code, it produces the following result -

```
[1] 18
```

```
[1] 45
```

String:

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

Examples of Valid Strings

Following examples clarify the rules about creating a string in R.

```
a <- 'Start and end with single quote'
print(a)

b <- "Start and end with double quotes"
print(b)

c <- "single quote ' in between double quotes"
print(c)

d <- 'Double quotes " in between single quote'
print(d)
```

Concatenating Strings - paste() function

Many strings in R are combined using the **paste()** function. It can take any number of arguments to be combined together.

Example

```
a <- "Hello"
b <- 'How'
c <- "are you? "
```



```
print(paste(a,b,c))
```



```
print(paste(a,b,c, sep = "-"))
```



```
print(paste(a,b,c, sep = "", collapse = ""))
```

When we execute the above code, it produces the following result -

```
[1] "Hello How are you? "
```



```
[1] "Hello-How-are you? "
```



```
[1] "HelloHoware you? "
```

Changing the case - toupper() & tolower() functions

These functions change the case of characters of a string.

Example

```
# Changing to Upper case.  
result <- toupper("Changing To Upper")  
print(result)  
  
# Changing to lower case.  
result <- tolower("Changing To Lower")  
print(result)
```

When we execute the above code, it produces the following result -

```
[1] "CHANGING TO UPPER"  
[1] "changing to lower"
```

Extracting parts of a string - substring() function

This function extracts parts of a String.

Syntax

The basic syntax for substring() function is -

```
substring(x,first,last)
```

Following is the description of the parameters used -

- ▮ **x** is the character vector input.
- ▮ **first** is the position of the first character to be extracted.
- ▮ **last** is the position of the last character to be extracted.

Example

```
# Extract characters from 5th to 7th position.  
result <- substring("Extract", 5, 7)  
print(result)
```

When we execute the above code, it produces the following result -

```
[1] "act"
```

Some R functions for computing Descriptive Statistics

Description

Mean

Standard deviation

Variance

Minimum

Maximum

Median

Range of values (minimum and maximum)

Sample quantiles

Generic function

Interquartile range

R function

`mean()`

`sd()`

`var()`

`min()`

`maximum()`

`median()`

`range()`

`quantile()`

`summary()`

`IQR()`

Descriptive Statistics

Summary: the function `summary()` is automatically applied to each column. The format of the result depends on the type of the data contained in the column. For example:

If the column is a numeric variable, mean, median, min, max and quartiles are returned. If the column is a factor variable, the number of observations in each group is returned.

```
# Create the data frame.
```

```
emp.data <- data.frame (emp_id = c(1:3), emp_name = c("Ramu","Venkat","Maha"), salary = c(623.3,515.2,611.0))
```

```
g = summary(emp.data, digit=1)
print(g)
```

When we execute the above code, it produces the following result –

	emp_id	emp_name	salary
min.	:1	Length:3	min. :515
1st Qu.	:2	Class :character	1st Qu.:563
Median	:2	Mode :character	Median :611
Mean	:2		Mean :583
3rd Qu.	:2		3rd Qu.:617
Max.	:3		Max. :623

Descriptive Statistics

Summary of a single variable. Five values are returned: the mean, median, Q1 and Q3 quartiles, min and max in one single line call.

```
f=summary(emp.data$salary)
print(f)
```

When we execute the above code, it produces the following result –

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
515.2	563.1	611.0	583.2	617.1	623.3

```
# Compute the mode---
install.packages("modeest")
# Import the library
library(modeest)

v = c(10,20,30,40,20)
# compute the Mode Value
c = mfv (v)
print(c)
```

Dr. Rambabu Palaka

Professor

School of Engineering

Malla Reddy University, Hyderabad

Mobile: +91-9652665840

Email: drrambabu@mallareddyuniversity.ac.in

Reference:

[R Tutorial - Website](#)