Data Analysis using R

Table of Contents

[R - Overview 9](#_Toc444203393)

[Evolution of R 9](#_Toc444203394)

[Features of R 9](#_Toc444203395)

[R - Environment Setup 11](#_Toc444203396)

[Try it Option Online 11](#_Toc444203397)

[Local Environment Setup 11](#_Toc444203398)

[Windows Installation 12](#_Toc444203399)

[Linux Installation 12](#_Toc444203400)

[R - Basic Syntax 14](#_Toc444203401)

[R Command Prompt 14](#_Toc444203402)

[R Script File 14](#_Toc444203403)

[Comments 15](#_Toc444203404)

[R - Data Types 16](#_Toc444203405)

[Vectors 18](#_Toc444203406)

[Lists 18](#_Toc444203407)

[Matrices 19](#_Toc444203408)

[Arrays 19](#_Toc444203409)

[Factors 20](#_Toc444203410)

[Data Frames 21](#_Toc444203411)

[R - Operators 23](#_Toc444203412)

[Types of Operators 23](#_Toc444203413)

[Arithmetic Operators 23](#_Toc444203414)

[Relational Operators 25](#_Toc444203415)

[Logical Operators 27](#_Toc444203416)

[Assignment Operators 28](#_Toc444203417)

[Miscellaneous Operators 29](#_Toc444203418)

[R - Decision making 31](#_Toc444203419)

[R - If Statement 33](#_Toc444203420)

[Syntax 33](#_Toc444203421)

[Flow Diagram 33](#_Toc444203422)

[Example 34](#_Toc444203423)

[R - If...Else Statement 35](#_Toc444203424)

[Syntax 35](#_Toc444203425)

[Flow Diagram 35](#_Toc444203426)

[Example 36](#_Toc444203427)

[The if...else if...else Statement 36](#_Toc444203428)

[Syntax 36](#_Toc444203429)

[Example 37](#_Toc444203430)

[R - Switch Statement 38](#_Toc444203431)

[Syntax 38](#_Toc444203432)

[Flow Diagram 39](#_Toc444203433)

[Example 39](#_Toc444203434)

[R - Loops 40](#_Toc444203435)

[Loop Control Statements 41](#_Toc444203436)

[R - Repeat Loop 42](#_Toc444203437)

[Syntax 42](#_Toc444203438)

[Flow Diagram 42](#_Toc444203439)

[Example 42](#_Toc444203440)

[R - While Loop 44](#_Toc444203441)

[Syntax 44](#_Toc444203442)

[Flow Diagram 44](#_Toc444203443)

[Example 44](#_Toc444203444)

[R - For Loop 46](#_Toc444203445)

[Syntax 46](#_Toc444203446)

[Flow Diagram 46](#_Toc444203447)

[Example 46](#_Toc444203448)

[R - Function 47](#_Toc444203449)

[Function Definition 47](#_Toc444203450)

[Function Components 47](#_Toc444203451)

[Example 48](#_Toc444203452)

[Built-in Function 48](#_Toc444203453)

[User Defined Function 48](#_Toc444203454)

[Calling a Function 49](#_Toc444203455)

[Calling a Function without an Argument 49](#_Toc444203456)

[Calling a Function with Argument Values (by position and by name) 50](#_Toc444203457)

[Calling a Function with Default Argument 51](#_Toc444203458)

[Lazy Evaluation of Function 51](#_Toc444203459)

[R - Strings 53](#_Toc444203460)

[Rules Applied in String Construction 53](#_Toc444203461)

[Examples of Valid Strings 53](#_Toc444203462)

[Examples of Invalid Strings 54](#_Toc444203463)

[String Manipulation 55](#_Toc444203464)

[Concatenating Strings - paste() function 55](#_Toc444203465)

[Syntax 55](#_Toc444203466)

[Example 55](#_Toc444203467)

[Formatting numbers & strings - format() function 56](#_Toc444203468)

[Syntax 56](#_Toc444203469)

[Example 56](#_Toc444203470)

[Counting number of characters in a string - nchar() function 57](#_Toc444203471)

[Syntax 57](#_Toc444203472)

[Example 58](#_Toc444203473)

[Changing the case - toupper() & tolower() functions 58](#_Toc444203474)

[Syntax 58](#_Toc444203475)

[Example 58](#_Toc444203476)

[Extracting parts of a string - substring() function 59](#_Toc444203477)

[Syntax 59](#_Toc444203478)

[Example 59](#_Toc444203479)

[R - Vectors 60](#_Toc444203480)

[Vector Creation 60](#_Toc444203481)

[Single Element Vector 60](#_Toc444203482)

[Multiple Elements Vector 61](#_Toc444203483)

[Accessing Vector Elements 62](#_Toc444203484)

[Vector Manipulation 63](#_Toc444203485)

[Vector arithmetic 63](#_Toc444203486)

[Vector element recycling 64](#_Toc444203487)

[Vector Element Sorting 64](#_Toc444203488)

[R - Lists 66](#_Toc444203489)

[Creating a List 66](#_Toc444203490)

[Naming List Elements 67](#_Toc444203491)

[Accessing List Elements 68](#_Toc444203492)

[Manipulating List Elements 69](#_Toc444203493)

[Merging Lists 70](#_Toc444203494)

[Converting List to Vector 71](#_Toc444203495)

[R - Matrices 73](#_Toc444203496)

[Syntax 73](#_Toc444203497)

[Example 73](#_Toc444203498)

[Accessing Elements of a Matrix 74](#_Toc444203499)

[Matrix Computations 75](#_Toc444203500)

[Matrix Addition & Subtraction 76](#_Toc444203501)

[Matrix Multiplication & Division 77](#_Toc444203502)

[R - Arrays 79](#_Toc444203503)

[Example 79](#_Toc444203504)

[Naming Columns and Rows 80](#_Toc444203505)

[Accessing Array Elements 81](#_Toc444203506)

[Manipulating Array Elements 82](#_Toc444203507)

[Calculations Across Array Elements 83](#_Toc444203508)

[Syntax 83](#_Toc444203509)

[Example 83](#_Toc444203510)

[R - Factors 85](#_Toc444203511)

[Example 85](#_Toc444203512)

[Factors in Data Frame 86](#_Toc444203513)

[Changing the Order of Levels 87](#_Toc444203514)

[Generating Factor Levels 87](#_Toc444203515)

[Syntax 87](#_Toc444203516)

[Example 88](#_Toc444203517)

[R - Data Frames 89](#_Toc444203518)

[Create Data Frame 89](#_Toc444203519)

[Get the Structure of the Data Frame 90](#_Toc444203520)

[Summary of Data in Data Frame 90](#_Toc444203521)

[Extract Data from Data Frame 91](#_Toc444203522)

[Expand Data Frame 93](#_Toc444203523)

[Add Column 93](#_Toc444203524)

[Add Row 94](#_Toc444203525)

[R - Packages 96](#_Toc444203526)

[Check Available R Packages 96](#_Toc444203527)

[Get the list of all the packages installed 96](#_Toc444203528)

[Install a New Package 97](#_Toc444203529)

[Install directly from CRAN 97](#_Toc444203530)

[Install package manually 98](#_Toc444203531)

[Load Package to Library 98](#_Toc444203532)

[R - Data Reshaping 99](#_Toc444203533)

[Joining Columns and Rows in a Data Frame 99](#_Toc444203534)

[Merging Data Frames 101](#_Toc444203535)

[Melting and Casting 103](#_Toc444203536)

[Melt the Data 103](#_Toc444203537)

[Cast the Molten Data 105](#_Toc444203538)

[R - CSV Files 107](#_Toc444203539)

[Getting and Setting the Working Directory 107](#_Toc444203540)

[Input as CSV File 108](#_Toc444203541)

[Reading a CSV File 108](#_Toc444203542)

[Analyzing the CSV File 109](#_Toc444203543)

[Get the maximum salary 109](#_Toc444203544)

[Get the details of the person with max salary 110](#_Toc444203545)

[Get all the people working in IT department 110](#_Toc444203546)

[Get the persons in IT department whose salary is greater than 600 111](#_Toc444203547)

[Get the people who joined on or after 2014 111](#_Toc444203548)

[Writing into a CSV File 111](#_Toc444203549)

[Excel File 113](#_Toc444203550)

[Install xlsx Package 113](#_Toc444203551)

[Verify and Load the "xlsx" Package 113](#_Toc444203552)

[Input as xlsx File 114](#_Toc444203553)

[Reading the Excel File 114](#_Toc444203554)

[R - Pie Charts 116](#_Toc444203555)

[Syntax 116](#_Toc444203556)

[Example 116](#_Toc444203557)

[Pie Chart Title and Colors 118](#_Toc444203558)

[Example 118](#_Toc444203559)

[Slice Percentages and Chart Legend 119](#_Toc444203560)

[3D Pie Chart 120](#_Toc444203561)

[R - Bar Charts 122](#_Toc444203562)

[Syntax 122](#_Toc444203563)

[Example 122](#_Toc444203564)

[Bar Chart Labels, Title and Colors 124](#_Toc444203565)

[Example 124](#_Toc444203566)

[Group Bar Chart and Stacked Bar Chart 125](#_Toc444203567)

[R - Line Graphs 127](#_Toc444203568)

[Syntax 127](#_Toc444203569)

[Example 127](#_Toc444203570)

[Line Chart Title, Color and Labels 129](#_Toc444203571)

[Example 129](#_Toc444203572)

[Multiple Lines in a Line Chart 130](#_Toc444203573)

[R - Histograms 132](#_Toc444203574)

[Syntax 132](#_Toc444203575)

[Example 132](#_Toc444203576)

[Range of X and Y values 134](#_Toc444203577)

[R - Mean, Median & Mode 135](#_Toc444203578)

[Mean 135](#_Toc444203579)

[Syntax 135](#_Toc444203580)

[Example 135](#_Toc444203581)

[Applying Trim Option 136](#_Toc444203582)

[Applying NA Option 136](#_Toc444203583)

[Median 137](#_Toc444203584)

[Syntax 137](#_Toc444203585)

[Example 137](#_Toc444203586)

[Mode 138](#_Toc444203587)

[Example 138](#_Toc444203588)

[R - Linear Regression 140](#_Toc444203589)

[Steps to Establish a Regression 140](#_Toc444203590)

[Input Data 141](#_Toc444203591)

[lm() Function 141](#_Toc444203592)

[Syntax 141](#_Toc444203593)

[Create Relationship Model & get the Coefficients 142](#_Toc444203594)

[Get the Summary of the Relationship 142](#_Toc444203595)

[predict() Function 143](#_Toc444203596)

[Syntax 143](#_Toc444203597)

[Predict the weight of new persons 143](#_Toc444203598)

[Visualize the Regression Graphically 144](#_Toc444203599)

# R - Overview

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

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.

R is free software distributed under a GNU-style copyleft, and an official part of the GNU project called GNU S.

## Evolution of R

R was initially written by **Ross Ihaka** and **Robert Gentleman** at the Department of Statistics of the University of Auckland in Auckland, New Zealand. R made its first appearance in 1993.

* A large group of individuals has contributed to R by sending code and bug reports.
* Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.

## Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. There are following important features of R:

* R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
* R has an effective data handling and storage facility,
* R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
* R provides a large, coherent and integrated collection of tools for data analysis.
* R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

As a conclusion, R is world’s most widely used statistics programming language. It's the # 1 choice of data scientists and supported by a vibrant and talented community of contributors. R is taught in universities and deployed in mission critical business applications. This tutorial will teach you R programming along with suitable examples in simple and easy steps.

# R - Environment Setup

## Try it Option Online

You really do not need to set up your own environment to start learning R programming language. Reason is very simple, we already have set up R Programming environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using **Try it** option available at the top right corner of the below sample code box:

# Print Hello World.

print("Hello World")

# Add two numbers.

print(23.9 + 11.6)

For most of the examples given in this tutorial, you will find **Try it** option, so just make use of it and enjoy your learning.

## Local Environment Setup

If you are still willing to set up your environment for R, you can follow the following steps in your computer.

## Windows Installation

You can download the Windows installer version of R from [R-3.2.2 for Windows (32/64 bit)](https://cran.r-project.org/bin/windows/base/) and save it in a local directory.

As it is a Windows installer (.exe) with a name "R-version-win.exe". You can just double click and run the installer accepting the default settings. If your Windows is 32-bit version, it installs the 32-bit version. But if your windows is 64-bit, then it installs both the 32-bit and 64-bit versions.

After installation you can locate the icon to run the Program in a directory structure "R\R-3.2.2\bin\i386\Rgui.exe" under the Windows Program Files. Clicking this icon brings up the R-GUI which is the R console to do R Programming.

## Linux Installation

R is available as a binary for many version of Linux at the location [R Binaries](https://cran.r-project.org/bin/linux/).

The instruction to install for various flavors of Linux varies. These steps are mentioned under each type of Linux version in the mentioned link. Still you are in hurry, then you can use **yum** command to install R as follows:

$ yum install R

Above command will install core functionality of R programming along with standard packages, still you need additional package then you can launch R prompt as follows:

$ R

R version 3.2.0 (2015-04-16) -- "Full of Ingredients"

Copyright (C) 2015 The R Foundation for Statistical Computing

Platform: x86\_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.

You are welcome to redistribute it under certain conditions.

Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.

Type 'contributors()' for more information and

'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or

'help.start()' for an HTML browser interface to help.

Type 'q()' to quit R.

>

Now you can use install command at R prompt to install a required package. For example following command will install **plotrix** package which is required for 3D charts.

> install("plotrix")

# R - Basic Syntax

As a convention we will start learning R programming by writing a "Hello, World!" program. Depending on the needs, you can program either at R command prompt or you can use a R script file to write your program. Let's check both one by one.

## R Command Prompt

Once you have R environment setup then its easy to start your R command prompt just type the following command at your command prompt:

$ R

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows:

> myString <- "Hello, World!"

> print ( myString)

[1] "Hello, World!"

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

## R Script File

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called **Rscript**. So let's start with writing following code in a text file called test.R:

# My first program in R Programming

myString <- "Hello, World!"

print ( myString)

Save above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

$ Rscript test.R

When we run above program, it produces the following result.

[1] "Hello, World!"

## Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows:

# My first program in R Programming

R does not support multi-line comments but you can do some trick something as follows:

if(FALSE){

"This is a demo for multi-line comments and it should be put

inside either a single of double quote"

}

myString <- "Hello, World!"

print ( myString)

Though above comments will be executed by R interpreter but they will not interfere with your actual program. You should put such comments inside either single or double quote.

# R - Data Types

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

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** | **Verify** |
| Logical | TRUE, FALSE | v <- TRUE  print(class(v))  it produces the following result −  [1] "logical" |
| Numeric | 12.3, 5, 999 | v <- 23.5  print(class(v))  it produces the following result −  [1] "numeric" |
| Integer | 2L, 34L, 0L | v <- 2L  print(class(v))  it produces the following result −  [1] "integer" |
| Complex | 3 + 2i | v <- 2+5i  print(class(v))  it produces the following result −  [1] "complex" |
| Character | 'a' , '"good", "TRUE", '23.4' | v <- "TRUE"  print(class(v))  it produces the following result −  [1] "character" |
| Raw | "Hello" is stored as 48 65 6c 6c 6f | v <- charToRaw("Hello")  print(class(v))  it produces the following result −  [1] "raw" |

In R programming, the very basic data types are the R-objects called **vectors**which hold elements of different classes as shown above. Please note in R the number of classes is not confined to only the above six types. For example, we can use many atomic vectors and create an array whose class will become array.

## 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))

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

[1] "red" "green" "yellow"

[1] "character"

## 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")

## 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"

## 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** functions 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 yellow green

Levels: green red yellow

# applying the nlevels function we can know the number of distinct values

[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

# R - 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 −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Assignment Operators
* Miscellaneous Operators

## Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two vectors | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v+t)  it produces the following result −  [1] 10.0 8.5 10.0 |
| − | Subtracts second vector from the first | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v-t)  it produces the following result −  [1] -6.0 2.5 2.0 |
| \* | Multiplies both vectors | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v\*t)  it produces the following result −  [1] 16.0 16.5 24.0 |
| / | Divide the first vector with the second | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v/t)  When we execute the above code, it produces the following result −  [1] 0.250000 1.833333 1.500000 |
| %% | Give the remainder of the first vector with the second | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v%%t)  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)  it produces the following result −  [1] 0 1 1 |
| ^ | The first vector raised to the exponent of second vector | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v^t)  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. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v>t)  it produces the following result −  [1] FALSE TRUE FALSE FALSE |
| < | Checks if each element of the first vector is less than the corresponding element of the second vector. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v < t)  it produces the following result −  [1] TRUE FALSE TRUE FALSE |
| == | 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)  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)  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)  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)  t <- c(8,2.5,14,9)  print(v!=t)  it produces the following result −  [1] TRUE TRUE TRUE FALSE |

## Logical Operators

Following table shows the logical operators supported by R language. 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. | v <- c(3,1,TRUE,2+3i)  t <- c(4,1,FALSE,2+3i)  print(v&t)  it produces the following result −  [1] TRUE TRUE FALSE TRUE |
| | | 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)  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)  print(!v)  it produces the following result −  [1] FALSE TRUE FALSE FALSE |

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE. | v <- c(3,0,TRUE,2+2i)  t <- c(1,3,TRUE,2+3i)  print(v&&t)  it produces the following result −  [1] TRUE |
| || | Called Logical OR operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE. | v <- c(0,0,TRUE,2+2i)  t <- c(0,3,TRUE,2+3i)  print(v||t)  it produces the following result −  [1] FALSE |

## Assignment Operators

These operators are used to assign values to vectors.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| <−  or  =  or  <<− | Called Left Assignment | v1 <- c(3,1,TRUE,2+3i)  v2 <<- c(3,1,TRUE,2+3i)  v3 = c(3,1,TRUE,2+3i)  print(v1)  print(v2)  print(v3)  it produces the following result −  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 1+0i 2+3i |
| ->  or  ->> | Called Right Assignment | c(3,1,TRUE,2+3i) -> v1  c(3,1,TRUE,2+3i) ->> v2  print(v1)  print(v2)  it produces the following result −  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 1+0i 2+3i |

## Miscellaneous Operators

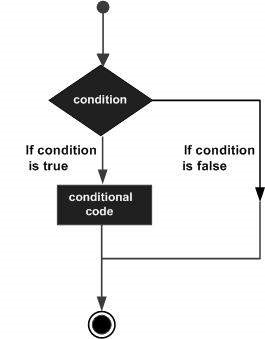
These operators are used to for specific purpose and not general mathematical or logical computation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| : | Colon operator. It creates the series of numbers in sequence for a vector. | v <- 2:8  print(v)  it produces the following result −  [1] 2 3 4 5 6 7 8 |
| %in% | This operator is used to identify if an element belongs to a vector. | v1 <- 8  v2 <- 12  t <- 1:10  print(v1 %in% t)  print(v2 %in% t)  it produces the following result −  [1] TRUE  [1] FALSE |
| %\*% | This operator is used to multiply a matrix with its transpose. | M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)  t = M %\*% t(M)  print(t)  it produces the following result −  [,1] [,2]  [1,] 65 82  [2,] 82 117 |

# R - Decision making

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 | [**if statement**](http://www.tutorialspoint.com/r/r_if_statement.htm)  An **if** statement consists of a Boolean expression followed by one or more statements. |
| 2 | [**if...else statement**](http://www.tutorialspoint.com/r/r_if_else_statement.htm)  An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false. |
| 3 | [**switch statement**](http://www.tutorialspoint.com/r/r_switch_statement.htm)  A **switch** statement allows a variable to be tested for equality against a list of values. |

# R - 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 −

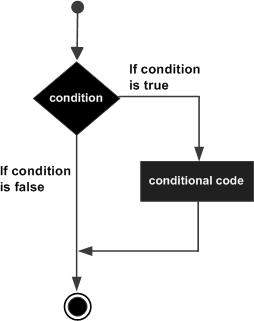
if(boolean\_expression) {

// 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.

## Flow Diagram



## 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"

# R - 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.

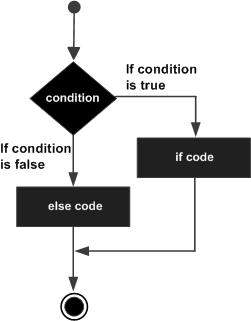
} else {

// 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.

## Flow Diagram



## 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"

Here "Truth" and "truth" are two different strings.

## 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.

## Syntax

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

if(boolean\_expression 1) {

// Executes when the boolean expression 1 is true.

} else if( boolean\_expression 2) {

// Executes when the boolean expression 2 is true.

} else if( boolean\_expression 3) {

// Executes when the boolean expression 3 is true.

} else {

// executes when none of the above condition is true.

}

## Example

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"

# R - 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.

## Syntax

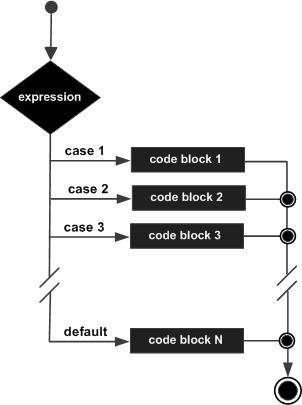
The basic syntax for creating a switch statement in R is −

switch(expression, case1, case2, case3....)

The following rules apply to a switch statement −

* If the value of expression is not a character string it is coerced to integer.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* If the value of the integer is between 1 and nargs()−1 (The max number of arguments)then the corresponding element of case condition is evaluated and the result returned.
* If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.
* If there is more than one match, the first matching element is returned.
* No Default argument is available.
* In the case of no match, if there is a unnamed element of ... its value is returned. (If there is more than one such argument an error is returned.)

## Flow Diagram



## Example

x <- switch(

3,

"first",

"second",

"third",

"fourth"

)

print(x)

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

[1] "third"

# R - Loops

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 following is the general from of a loop statement in most of the programming languages:

R programming language provides following kinds of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [**repeat loop**](http://www.tutorialspoint.com/r/r_repeat_loop.htm) | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [**while loop**](http://www.tutorialspoint.com/r/r_while_loop.htm) | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [**for loop**](http://www.tutorialspoint.com/r/r_for_loop.htm) | Like a while statement, except that it tests the condition at the end of the loop body. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| [**break statement**](http://www.tutorialspoint.com/r/r_break_statement.htm) | Terminates the **loop** statement and transfers execution to the statement immediately following the loop. |
| [**Next statement**](http://www.tutorialspoint.com/r/r_next_statement.htm) | The next statement simulates the behavior of R switch. |

# R - Repeat Loop

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

## Syntax

The basic syntax for creating a repeat loop in R is:

repeat {

commands

if(condition){

break

}

}

## Flow Diagram

Example

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"

# R - While Loop

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

## Syntax

The basic syntax for creating a while loop in R is :

while (test\_expression) {

statement

}

## Flow Diagram

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.

Example

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"

# R - 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.

## Syntax

The basic syntax for creating a for loop statement in R is:

for (value in vector) {

statements

}

## Flow Diagram

R’s for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

Example

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"

# R - Function

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:

* **Function Name:** This is the actual name of the function. It is stored in R environment as an object with this name.
* **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.
* **Function Body:** The function body contains a collection of statements that defines what the function does.
* **Return value:** The return value of a function is the last expression in the function body to be evaluated.

Example

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. You can refer[most widely used R functions.](https://cran.r-project.org/doc/contrib/Short-refcard.pdf)

# 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 frm 41 to 68.

print(sum(41:68))

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

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

[1] 36

## Calling a Function without an Argument

# 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 above code, it produces 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)

When we execute above code, it produces 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)

}

# 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 above code, it produces following result:

[1] 18

[1] 45

## Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

# Create a function with arguments.

new.function <- function(a, b) {

print(a^2)

print(a)

print(b)

}

# Evaluate the function without supplying one of the arguments.

new.function(6)

When we execute above code, it produces following result:

[1] 36

[1] 6

Error in print(b) : argument "b" is missing, with no default

# R - Strings

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.

## Rules Applied in String Construction

* The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
* Double quotes can be inserted into a string starting and ending with single quote.
* Single quote can be inserted into a string starting and ending with double quotes.
* Double quotes can not be inserted into a string starting and ending with double quotes.
* Single quote can not be inserted into a string starting and ending 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)

When the above code is run we get the following output −

[1] "Start and end with single quote"

[1] "Start and end with double quotes"

[1] "single quote ' in between double quote"

[1] "Double quote \" in between single quote"

## Examples of Invalid Strings

e <- 'Mixed quotes"

print(e)

f <- 'Single quote ' inside single quote'

print(f)

g <- "Double quotes " inside double quotes"

print(g)

When we run the script it fails giving below results.

...: unexpected INCOMPLETE\_STRING

.... unexpected symbol

1: f <- 'Single quote ' inside

unexpected symbol

1: g <- "Double quotes " inside

## String Manipulation

### 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.

### Syntax

The basic syntax for paste function is −

paste(..., sep = " ", collapse = NULL)

Following is the description of the parameters used −

* **...** represents any number of arguments to be combined.
* **sep** represents any separator between the arguments. It is optional.
* **collapse** is used to eliminate the space in between two strings. But not the space within two words of one string.

### 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? "

### Formatting numbers & strings - format() function

Numbers and strings can be formatted to a specific style using **format()**function.

### Syntax

The basic syntax for format function is −

format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))

Following is the description of the parameters used −

* **x** is the vector input.
* **digits** is the total number of digits displayed.
* **nsmall** is the minimum number of digits to the right of the decimal point.
* **scientific** is set to TRUE to display scientific notation.
* **width** indicates the minimum width to be displayed by padding blanks in the beginning.
* **justify** is the display of the string to left, right or center.

### Example

# Total number of digits displayed. Last digit rounded off.

result <- format(23.123456789, digits = 9)

print(result)

# Display numbers in scientific notation.

result <- format(c(6, 13.14521), scientific = TRUE)

print(result)

# The minimum number of digits to the right of the decimal point.

result <- format(23.47, nsmall = 5)

print(result)

# Format treats everything as a string.

result <- format(6)

print(result)

# Numbers are padded with blank in the beginning for width.

result <- format(13.7, width = 6)

print(result)

# Left justify strings.

result <- format("Hello", width = 8, justify = "l")

print(result)

# Justfy string with center.

result <- format("Hello", width = 8, justify = "c")

print(result)

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

[1] "23.1234568"

[1] "6.000000e+00" "1.314521e+01"

[1] "23.47000"

[1] "6"

[1] " 13.7"

[1] "Hello "

[1] " Hello "

## Counting number of characters in a string - nchar() function

This function counts the number of characters including spaces in a string.

## Syntax

The basic syntax for nchar() function is −

nchar(x)

Following is the description of the parameters used −

* **x** is the vector input.

### Example

result <- nchar("Count the number of characters")

print(result)

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

[1] 30

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

These functions change the case of characters of a string.

## Syntax

The basic syntax for toupper() & tolower() function is −

toupper(x)

tolower(x)

Following is the description of the parameters used −

* **x** is the vector input.

### 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"

# R - Vectors

Vectors are the most basic R data objects and there are six types of atomic vectors. They are logical, integer, double, complex, character and raw.

## Vector Creation

## Single Element Vector

Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

# Atomic vector of type character.

print("abc");

# Atomic vector of type double.

print(12.5)

# Atomic vector of type integer.

print(63L)

# Atomic vector of type logical.

print(TRUE)

# Atomic vector of type complex.

print(2+3i)

# Atomic vector of type raw.

print(charToRaw('hello'))

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

[1] "abc"

[1] 12.5

[1] 63

[1] TRUE

[1] 2+3i

[1] 68 65 6c 6c 6f

## Multiple Elements Vector

**Using colon operator with numeric data**

# Creating a sequence from 5 to 13.

v <- 5:13

print(v)

# Creating a sequence from 6.6 to 12.6.

v <- 6.6:12.6

print(v)

# If the final element specified does not belong to the sequence then it is discarded.

v <- 3.8:11.4

print(v)

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

[1] 5 6 7 8 9 10 11 12 13

[1] 6.6 7.6 8.6 9.6 10.6 11.6 12.6

[1] 3.8 4.8 5.8 6.8 7.8 8.8 9.8 10.8

**Using sequence (Seq.) operator**

# Create vector with elements from 5 to 9 incrementing by 0.4.

print(seq(5, 9, by = 0.4))

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

[1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0

**Using the c() function**

The non-character values are coerced to character type if one of the elements is a character.

# The logical and numeric values are converted to characters.

s <- c('apple','red',5,TRUE)

print(s)

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

[1] "apple" "red" "5" "TRUE"

## Accessing Vector Elements

Elements of a Vector are accessed using indexing. The **[ ] brackets** are used for indexing. Indexing starts with position 1. Giving a negative value in the index drops that element from result.**TRUE**,**FALSE** or **0** and **1** can also be used for indexing.

# Accessing vector elements using position.

t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")

u <- t[c(2,3,6)]

print(u)

# Accessing vector elements using logical indexing.

v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]

print(v)

# Accessing vector elements using negative indexing.

x <- t[c(-2,-5)]

print(x)

# Accessing vector elements using 0/1 indexing.

y <- t[c(0,0,0,0,0,0,1)]

print(y)

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

[1] "Mon" "Tue" "Fri"

[1] "Sun" "Fri"

[1] "Sun" "Tue" "Wed" "Fri" "Sat"

[1] "Sun"

## 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 substraction.

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 element recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

v1 <- c(3,8,4,5,0,11)

v2 <- c(4,11)

# V2 becomes c(4,11,4,11,4,11)

add.result <- v1+v2

print(add.result)

sub.result <- v1-v2

print(sub.result)

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

[1] 7 19 8 16 4 22

[1] -1 -3 0 -6 -4 0

## 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"

# R - Lists

Lists are the R objects which contain elements of different types like − numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

## Creating a List

Following is an example to create a list containing strings, numbers, vectors and a logical values

# Create a list containing strings, numbers, vectors and a logical values.

list\_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)

print(list\_data)

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

[[1]]

[1] "Red"

[[2]]

[1] "Green"

[[3]]

[1] 21 32 11

[[4]]

[1] TRUE

[[5]]

[1] 51.23

[[6]]

[1] 119.1

## 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

## Accessing List Elements

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

We continue to use the list in the above example −

# 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")

# Access the first element of the list.

print(list\_data[1])

# Access the thrid element. As it is also a list, all its elements will be printed.

print(list\_data[3])

# Access the list element using the name of the element.

print(list\_data$A\_Matrix)

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

$`1st\_Quarter`

[1] "Jan" "Feb" "Mar"

$A\_Inner\_list

$A\_Inner\_list[[1]]

[1] "green"

$A\_Inner\_list[[2]]

[1] 12.3

[,1] [,2] [,3]

[1,] 3 5 -2

[2,] 9 1 8

## Manipulating List Elements

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"

$

NULL

$`A Inner list`

[1] "updated element"

## 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"

## 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

# R - Matrices

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.

A Matrix is created using the **matrix()** function.

## Syntax

The basic syntax for creating a matrix in R is −

matrix(data, nrow, ncol, byrow, dimnames)

Following is the description of the parameters used −

* **data** is the input vector which becomes the data elements of the matrix.
* **nrow** is the number of rows to be created.
* **ncol** is the number of columns to be created.
* **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
* **dimname** is the names assigned to the rows and columns.

## Example

Create a matrix taking a vector of numbers as input

# Elements are arranged sequentially by row.

M <- matrix(c(3:14), nrow = 4, byrow = TRUE)

print(M)

# Elements are arranged sequentially by column.

N <- matrix(c(3:14), nrow = 4, byrow = FALSE)

print(N)

# Define the column and row names.

rownames = c("row1", "row2", "row3", "row4")

colnames = c("col1", "col2", "col3")

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

print(P)

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

[,1] [,2] [,3]

[1,] 3 4 5

[2,] 6 7 8

[3,] 9 10 11

[4,] 12 13 14

[,1] [,2] [,3]

[1,] 3 7 11

[2,] 4 8 12

[3,] 5 9 13

[4,] 6 10 14

col1 col2 col3

row1 3 4 5

row2 6 7 8

row3 9 10 11

row4 12 13 14

## Accessing Elements of a 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)

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 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

## 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

# R - Arrays

Arrays are the R data objects which can store data in more than two dimensions. For example − If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.

An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array.

## Example

The following example creates an array of two 3x3 matrices each with 3 rows and 3 columns.

# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.

result <- array(c(vector1,vector2),dim = c(3,3,2))

print(result)

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

, , 1

[,1] [,2] [,3]

[1,] 5 10 13

[2,] 9 11 14

[3,] 3 12 15

, , 2

[,1] [,2] [,3]

[1,] 5 10 13

[2,] 9 11 14

[3,] 3 12 15

## Naming Columns and Rows

We can give names to the rows, columns and matrices in the array by using the**dimnames** parameter.

# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

column.names <- c("COL1","COL2","COL3")

row.names <- c("ROW1","ROW2","ROW3")

matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.

result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(column.names,row.names,

matrix.names))

print(result)

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

, , Matrix1

ROW1 ROW2 ROW3

COL1 5 10 13

COL2 9 11 14

COL3 3 12 15

, , Matrix2

ROW1 ROW2 ROW3

COL1 5 10 13

COL2 9 11 14

COL3 3 12 15

## Accessing Array Elements

# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

column.names <- c("COL1","COL2","COL3")

row.names <- c("ROW1","ROW2","ROW3")

matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.

result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(column.names,

row.names, matrix.names))

# Print the third row of the second matrix of the array.

print(result[3,,2])

# Print the element in the 1st row and 3rd column of the 1st matrix.

print(result[1,3,1])

# Print the 2nd Matrix.

print(result[,,2])

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

ROW1 ROW2 ROW3

3 12 15

[1] 13

ROW1 ROW2 ROW3

COL1 5 10 13

COL2 9 11 14

COL3 3 12 15

## Manipulating Array Elements

As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.

# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.

array1 <- array(c(vector1,vector2),dim = c(3,3,2))

# Create two vectors of different lengths.

vector3 <- c(9,1,0)

vector4 <- c(6,0,11,3,14,1,2,6,9)

array2 <- array(c(vector1,vector2),dim = c(3,3,2))

# create matrices from these arrays.

matrix1 <- array1[,,2]

matrix2 <- array2[,,2]

# Add the matrices.

result <- matrix1+matrix2

print(result)

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

[,1] [,2] [,3]

[1,] 10 20 26

[2,] 18 22 28

[3,] 6 24 30

## Calculations Across Array Elements

We can do calculations across the elements in an array using the **apply()**function.

### Syntax

apply(x, margin, fun)

Following is the description of the parameters used −

* **x** is an array.
* **margin** is the name of the data set used.
* **fun** is the function to be applied across the elements of the array.

### Example

We use the apply() function below to calculate the sum of the elements in the rows of an array across all the matrices.

# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.

new.array <- array(c(vector1,vector2),dim = c(3,3,2))

print(new.array)

# Use apply to calculate the sum of the rows across all the matrices.

result <- apply(new.array, c(1), sum)

print(result)

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

, , 1

[,1] [,2] [,3]

[1,] 5 10 13

[2,] 9 11 14

[3,] 3 12 15

, , 2

[,1] [,2] [,3]

[1,] 5 10 13

[2,] 9 11 14

[3,] 3 12 15

[1] 56 68 60

# R - Factors

Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like "Male, "Female" and True, False etc. They are useful in data analysis for statistical modeling.

Factors are created using the **factor ()** function by taking a vector as input.

## Example

# Create a vector as input.

data <- c("East","West","East","North","North","East","West","West","West","East","North")

print(data)

print(is.factor(data))

# Apply the factor function.

factor\_data <- factor(data)

print(factor\_data)

print(is.factor(factor\_data))

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

[1] "East" "West" "East" "North" "North" "East" "West" "West" "West" "East" "North"

[1] FALSE

[1] East West East North North East West West West East North

Levels: East North West

[1] TRUE

## Factors in Data Frame

On creating any data frame with a column of text data, R treats the text column as categorical data and creates factors on it.

# Create the vectors for data frame.

height <- c(132,151,162,139,166,147,122)

weight <- c(48,49,66,53,67,52,40)

gender <- c("male","male","female","female","male","female","male")

# Create the data frame.

input\_data <- data.frame(height,weight,gender)

print(input\_data)

# Test if the gender column is a factor.

print(is.factor(input\_data$gender))

# Print the gender column so see the levels.

print(input\_data$gender)

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

height weight gender

1 132 48 male

2 151 49 male

3 162 66 female

4 139 53 female

5 166 67 male

6 147 52 female

7 122 40 male

[1] TRUE

[1] male male female female male female male

Levels: female male

## Changing the Order of Levels

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

data <- c("East","West","East","North","North","East","West","West","West","East","North")

# Create the factors

factor\_data <- factor(data)

print(factor\_data)

# Apply the factor function with required order of the level.

new\_order\_data <- factor(factor\_data,levels = c("East","West","North"))

print(new\_order\_data)

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

[1] East West East North North East West West West East North

Levels: East North West

[1] East West East North North East West West West East North

Levels: East West North

## Generating Factor Levels

We can generate factor levels by using the **gl()** function. It takes two integers as input which indicates how many levels and how many times each level.

### Syntax

gl(n, k, labels)

Following is the description of the parameters used −

* **n** is a integer giving the number of levels.
* **k** is a integer giving the number of replications.
* **labels** is a vector of labels for the resulting factor levels.

### Example

v <- gl(3, 4, labels = c("Tampa", "Seattle","Boston"))

print(v)

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

Tampa Tampa Tampa Tampa Seattle Seattle Seattle Seattle Boston

[10] Boston Boston Boston

Levels: Tampa Seattle Boston

# R - Data Frames

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

* The column names should be non-empty.
* The row names should be unique.
* The data stored in a data frame can be of numeric, factor or character type.
* Each column should contain same number of data items.

## Create Data Frame

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Print the data frame.

print(emp.data)

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

emp\_id emp\_name salary start\_date

1 1 Rick 623.30 2012-01-01

2 2 Dan 515.20 2013-09-23

3 3 Michelle 611.00 2014-11-15

4 4 Ryan 729.00 2014-05-11

5 5 Gary 843.25 2015-03-27

## Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Get the structure of the data frame.

str(emp.data)

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

'data.frame': 5 obs. of 4 variables:

$ emp\_id : int 1 2 3 4 5

$ emp\_name : chr "Rick" "Dan" "Michelle" "Ryan" ...

$ salary : num 623 515 611 729 843

$ start\_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11" ...

## Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying**summary()** function.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Print the summary.

print(summary(emp.data))

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

emp\_id emp\_name salary start\_date

Min. :1 Length:5 Min. :515.2 Min. :2012-01-01

1st Qu.:2 Class :character 1st Qu.:611.0 1st Qu.:2013-09-23

Median :3 Mode :character Median :623.3 Median :2014-05-11

Mean :3 Mean :664.4 Mean :2014-01-14

3rd Qu.:4 3rd Qu.:729.0 3rd Qu.:2014-11-15

Max. :5 Max. :843.2 Max. :2015-03-27

## Extract Data from Data Frame

Extract specific column from a data frame using column name.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Extract Specific columns.

result <- data.frame(emp.data$emp\_name,emp.data$salary)

print(result)

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

emp.data.emp\_name emp.data.salary

1 Rick 623.30

2 Dan 515.20

3 Michelle 611.00

4 Ryan 729.00

5 Gary 843.25

Extract the first two rows and then all columns

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Extract first two rows.

result <- emp.data[1:2,]

print(result)

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

emp\_id emp\_name salary start\_date

1 1 Rick 623.3 2012-01-01

2 2 Dan 515.2 2013-09-23

Extract 3rd and 5th row with 2nd and 4th column

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Extract 3rd and 5th row with 2nd and 4th column.

result <- emp.data[c(3,5),c(2,4)]

print(result)

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

emp\_name start\_date

3 Michelle 2014-11-15

5 Gary 2015-03-27

## Expand Data Frame

A data frame can be expanded by adding columns and rows.

### Add Column

Just add the column vector using a new column name.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

stringsAsFactors = FALSE

)

# Add the "dept" coulmn.

emp.data$dept <- c("IT","Operations","IT","HR","Finance")

v <- emp.data

print(v)

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

emp\_id emp\_name salary start\_date dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 5 Gary 843.25 2015-03-27 Finance

### Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the**rbind()** function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

# Create the first data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",

"2015-03-27")),

dept = c("IT","Operations","IT","HR","Finance"),

stringsAsFactors = FALSE

)

# Create the second data frame

emp.newdata <- data.frame(

emp\_id = c (6:8),

emp\_name = c("Rasmi","Pranab","Tusar"),

salary = c(578.0,722.5,632.8),

start\_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),

dept = c("IT","Operations","Fianance"),

stringsAsFactors = FALSE

)

# Bind the two data frames.

emp.finaldata <- rbind(emp.data,emp.newdata)

print(emp.finaldata)

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

emp\_id emp\_name salary start\_date dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 5 Gary 843.25 2015-03-27 Finance

6 6 Rasmi 578.00 2013-05-21 IT

7 7 Pranab 722.50 2013-07-30 Operations

8 8 Tusar 632.80 2014-06-17 Fianance

# R - Packages

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called **"library"** in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at [R Packages.](https://cran.r-project.org/web/packages/available_packages_by_name.html)

Below is a list of commands to be used to check, verify and use the R packages.

## Check Available R Packages

Get library locations containing R packages

.libPaths()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

[2] "C:/Program Files/R/R-3.2.2/library"

## Get the list of all the packages installed

library()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

Packages in library ‘C:/Program Files/R/R-3.2.2/library’:

base The R Base Package

boot Bootstrap Functions (Originally by Angelo Canty

for S)

class Functions for Classification

cluster "Finding Groups in Data": Cluster Analysis

Extended Rousseeuw et al.

codetools Code Analysis Tools for R

compiler The R Compiler Package

Get all packages currently loaded in the R environment

search()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

[1] ".GlobalEnv" "package:stats" "package:graphics"

[4] "package:grDevices" "package:utils" "package:datasets"

[7] "package:methods" "Autoloads" "package:base"

## Install a New Package

There are two ways to add new R packages. One is installing directly from the CRAN directory and another is downloading the package to your local system and installing it manually.

## Install directly from CRAN

The following command gets the packages directly from CRAN webpage and installs the package in the R environment. You may be prompted to choose a nearest mirror. Choose the one appropriate to your location.

install.packages("Package Name")

# Install the package named "XML".

install.packages("XML")

## Install package manually

Go to the link [R Packages](https://cran.r-project.org/web/packages/available_packages_by_name.html) to download the package needed. Save the package as a **.zip** file in a suitable location in the local system.

Now you can run the following command to install this package in the R environment.

install.packages(file\_name\_with\_path, repos = NULL, type = "source")

# Install the package named "XML"

install.packages("E:/XML\_3.98-1.3.zip", repos = NULL, type = "source")

## Load Package to Library

Before a package can be used in the code, it must be loaded to the current R environment. You also need to load a package that is already installed previously but not available in the current environment.

A package is loaded using the following command −

library("package Name", lib.loc = "path to library")

# Load the package named "XML"

install.packages("E:/XML\_3.98-1.3.zip", repos = NULL, type = "source")

# R - Data Reshaping

Data Reshaping in R is about changing the way data is organized into rows and columns. Most of the time data processing in R is done by taking the input data as a data frame. It is easy to extract data from the rows and columns of a data frame but there are situations when we need the data frame in a format that is different from format in which we received it. R has many functions to split, merge and change the rows to columns and vice-versa in a data frame.

## Joining Columns and Rows in a Data Frame

We can join multiple vectors to create a data frame using the **cbind()**function. Also we can merge two data frames using **rbind()** function.

# Create vector objects.

city <- c("Tampa","Seattle","Hartford","Denver")

state <- c("FL","WA","CT","CO")

zipcode <- c(33602,98104,06161,80294)

# Combine above three vectors into one data frame.

addresses <- cbind(city,state,zipcode)

# Print a header.

cat("# # # # The First data frame\n")

# Print the data frame.

print(addresses)

# Create another data frame with similar columns

new.address <- data.frame(

city = c("Lowry","Charlotte"),

state = c("CO","FL"),

zipcode = c("80230","33949"),

stringsAsFactors = FALSE

)

# Print a header.

cat("# # # The Second data frame\n")

# Print the data frame.

print(new.address)

# Combine rows form both the data frames.

all.addresses <- rbind(addresses,new.address)

# Print a header.

cat("# # # The combined data frame\n")

# Print the result.

print(all.addresses)

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

# # # # The First data frame

city state zipcode

[1,] "Tampa" "FL" "33602"

[2,] "Seattle" "WA" "98104"

[3,] "Hartford" "CT" "6161"

[4,] "Denver" "CO" "80294"

# # # The Second data frame

city state zipcode

1 Lowry CO 80230

2 Charlotte FL 33949

# # # The combined data frame

city state zipcode

1 Tampa FL 33602

2 Seattle WA 98104

3 Hartford CT 6161

4 Denver CO 80294

5 Lowry CO 80230

6 Charlotte FL 33949

## Merging Data Frames

We can merge two data frames by using the **merge()** function. The data frames must have same column names on which the merging happens.

In the example below, we consider the data sets about Diabetes in Pima Indian Women available in the library names "MASS". we merge the two data sets based on the values of blood pressure("bp") and body mass index("bmi"). On choosing these two columns for merging, the records where values of these two variables match in both data sets are combined together to form a single data frame.

library(MASS)

merged.Pima <- merge(x = Pima.te, y = Pima.tr,

by.x = c("bp", "bmi"),

by.y = c("bp", "bmi")

)

print(merged.Pima)

nrow(merged.Pima)

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

bp bmi npreg.x glu.x skin.x ped.x age.x type.x npreg.y glu.y skin.y ped.y

1 60 33.8 1 117 23 0.466 27 No 2 125 20 0.088

2 64 29.7 2 75 24 0.370 33 No 2 100 23 0.368

3 64 31.2 5 189 33 0.583 29 Yes 3 158 13 0.295

4 64 33.2 4 117 27 0.230 24 No 1 96 27 0.289

5 66 38.1 3 115 39 0.150 28 No 1 114 36 0.289

6 68 38.5 2 100 25 0.324 26 No 7 129 49 0.439

7 70 27.4 1 116 28 0.204 21 No 0 124 20 0.254

8 70 33.1 4 91 32 0.446 22 No 9 123 44 0.374

9 70 35.4 9 124 33 0.282 34 No 6 134 23 0.542

10 72 25.6 1 157 21 0.123 24 No 4 99 17 0.294

11 72 37.7 5 95 33 0.370 27 No 6 103 32 0.324

12 74 25.9 9 134 33 0.460 81 No 8 126 38 0.162

13 74 25.9 1 95 21 0.673 36 No 8 126 38 0.162

14 78 27.6 5 88 30 0.258 37 No 6 125 31 0.565

15 78 27.6 10 122 31 0.512 45 No 6 125 31 0.565

16 78 39.4 2 112 50 0.175 24 No 4 112 40 0.236

17 88 34.5 1 117 24 0.403 40 Yes 4 127 11 0.598

age.y type.y

1 31 No

2 21 No

3 24 No

4 21 No

5 21 No

6 43 Yes

7 36 Yes

8 40 No

9 29 Yes

10 28 No

11 55 No

12 39 No

13 39 No

14 49 Yes

15 49 Yes

16 38 No

17 28 No

[1] 17

## Melting and Casting

One of the most interesting aspects of R programming is about changing the shape of the data in multiple steps to get a desired shape. The functions used to do this are called **melt()** and **cast()**.

We consider the dataset called ships present in the library called "MASS".

library(MASS)

print(ships)

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

type year period service incidents

1 A 60 60 127 0

2 A 60 75 63 0

3 A 65 60 1095 3

4 A 65 75 1095 4

5 A 70 60 1512 6

.............

.............

8 A 75 75 2244 11

9 B 60 60 44882 39

10 B 60 75 17176 29

11 B 65 60 28609 58

............

............

17 C 60 60 1179 1

18 C 60 75 552 1

19 C 65 60 781 0

............

............

## Melt the Data

Now we melt the data to organize it, converting all columns other than type and year into multiple rows.

molten.ships <- melt(ships, id = c("type","year"))

print(molten.ships)

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

type year variable value

1 A 60 period 60

2 A 60 period 75

3 A 65 period 60

4 A 65 period 75

............

............

9 B 60 period 60

10 B 60 period 75

11 B 65 period 60

12 B 65 period 75

13 B 70 period 60

...........

...........

41 A 60 service 127

42 A 60 service 63

43 A 65 service 1095

...........

...........

70 D 70 service 1208

71 D 75 service 0

72 D 75 service 2051

73 E 60 service 45

74 E 60 service 0

75 E 65 service 789

...........

...........

101 C 70 incidents 6

102 C 70 incidents 2

103 C 75 incidents 0

104 C 75 incidents 1

105 D 60 incidents 0

106 D 60 incidents 0

...........

...........

## Cast the Molten Data

We can cast the molten data into a new form where the aggregate of each type of ship for each year is created. It is done using the **cast()** function.

recasted.ship <- cast(molten.ships, type+year~variable,sum)

print(recasted.ship)

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

type year period service incidents

1 A 60 135 190 0

2 A 65 135 2190 7

3 A 70 135 4865 24

4 A 75 135 2244 11

5 B 60 135 62058 68

6 B 65 135 48979 111

7 B 70 135 20163 56

8 B 75 135 7117 18

9 C 60 135 1731 2

10 C 65 135 1457 1

11 C 70 135 2731 8

12 C 75 135 274 1

13 D 60 135 356 0

14 D 65 135 480 0

15 D 70 135 1557 13

16 D 75 135 2051 4

17 E 60 135 45 0

18 E 65 135 1226 14

19 E 70 135 3318 17

20 E 75 135 542 1

# R - CSV Files

In R, we can read data from files stored outside the R environment. We can also write data into files which will be stored and accessed by the operating system. R can read and write into various file formats like csv, excel, xml etc.

In this chapter we will learn to read data from a csv file and then write data into a csv file. The file should be present in current working directory so that R can read it. Of course we can also set our own directory and read files from there.

## Getting and Setting the Working Directory

You can check which directory the R workspace is pointing to using the**getwd()** function. You can also set a new working directory using**setwd()**function.

# Get and print current working directory.

print(getwd())

# Set current working directory.

setwd("/web/com")

# Get and print current working directory.

print(getwd())

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

[1] "/web/com/1441086124\_2016"

[1] "/web/com"

This result depends on your OS and your current directory where you are working.

## Input as CSV File

The csv file is a text file in which the values in the columns are separated by a comma. Let's consider the following data present in the file named **input.csv**.

You can create this file using windows notepad by copying and pasting this data. Save the file as **input.csv** using the save As All files(\*.\*) option in notepad.

id,name,salary,start\_date,dept

1,Rick,623.3,2012-01-01,IT

2,Dan,515.2,2013-09-23,Operations

3,Michelle,611,2014-11-15,IT

4,Ryan,729,2014-05-11,HR

,Gary,843.25,2015-03-27,Finance

6,Nina,578,2013-05-21,IT

7,Simon,632.8,2013-07-30,Operations

8,Guru,722.5,2014-06-17,Finance

## Reading a CSV File

Following is a simple example of **read.csv()** function to read a CSV file available in your current working directory −

data <- read.csv("input.csv")

print(data)

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

id, name, salary, start\_date, dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 NA Gary 843.25 2015-03-27 Finance

6 6 Nina 578.00 2013-05-21 IT

7 7 Simon 632.80 2013-07-30 Operations

8 8 Guru 722.50 2014-06-17 Finance

## Analyzing the CSV File

By default the **read.csv()** function gives the output as a data frame. This can be easily checked as follows. Also we can check the number of columns and rows.

data <- read.csv("input.csv")

print(is.data.frame(data))

print(ncol(data))

print(nrow(data))

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

[1] TRUE

[1] 5

[1] 8

Once we read data in a data frame, we can apply all the functions applicable to data frames as explained in subsequent section.

### Get the maximum salary

# Create a data frame.

data <- read.csv("input.csv")

# Get the max salary from data frame.

sal <- max(data$salary)

print(sal)

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

[1] 843.25

### Get the details of the person with max salary

We can fetch rows meeting specific filter criteria similar to a SQL where clause.

# Create a data frame.

data <- read.csv("input.csv")

# Get the max salary from data frame.

sal <- max(data$salary)

# Get the person detail having max salary.

retval <- subset(data, salary == max(salary))

print(retval)

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

id name salary start\_date dept

5 NA Gary 843.25 2015-03-27 Finance

### Get all the people working in IT department

# Create a data frame.

data <- read.csv("input.csv")

retval <- subset( data, dept == "IT")

print(retval)

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

id name salary start\_date dept

1 1 Rick 623.3 2012-01-01 IT

3 3 Michelle 611.0 2014-11-15 IT

6 6 Nina 578.0 2013-05-21 IT

### Get the persons in IT department whose salary is greater than 600

# Create a data frame.

data <- read.csv("input.csv")

info <- subset(data, salary > 600 & dept == "IT")

print(info)

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

id name salary start\_date dept

1 1 Rick 623.3 2012-01-01 IT

3 3 Michelle 611.0 2014-11-15 IT

### Get the people who joined on or after 2014

# Create a data frame.

data <- read.csv("input.csv")

retval <- subset(data, as.Date(start\_date) > as.Date("2014-01-01"))

print(retval)

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

id name salary start\_date dept

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 NA Gary 843.25 2015-03-27 Finance

8 8 Guru 722.50 2014-06-17 Finance

## Writing into a CSV File

R can create csv file form existing data frame. The **write.csv()** function is used to create the csv file. This file gets created in the working directory.

# Create a data frame.

data <- read.csv("input.csv")

retval <- subset(data, as.Date(start\_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.

write.csv(retval,"output.csv")

newdata <- read.csv("output.csv")

print(newdata)

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

X id name salary start\_date dept

1 3 3 Michelle 611.00 2014-11-15 IT

2 4 4 Ryan 729.00 2014-05-11 HR

3 5 NA Gary 843.25 2015-03-27 Finance

4 8 8 Guru 722.50 2014-06-17 Finance

Here the column X comes from the data set newper. This can be dropped using additional parameters while writing the file.

# Create a data frame.

data <- read.csv("input.csv")

retval <- subset(data, as.Date(start\_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.

write.csv(retval,"output.csv", row.names = FALSE)

newdata <- read.csv("output.csv")

print(newdata)

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

id name salary start\_date dept

1 3 Michelle 611.00 2014-11-15 IT

2 4 Ryan 729.00 2014-05-11 HR

3 NA Gary 843.25 2015-03-27 Finance

4 8 Guru 722.50 2014-06-17 Finance

# Excel File

Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format. R can read directly from these files using some excel specific packages. Few such packages are - XLConnect, xlsx, gdata etc. We will be using xlsx package. R can also write into excel file using this package.

## Install xlsx Package

You can use the following command in the R console to install the "xlsx" package. It may ask to install some additional packages on which this package is dependent. Follow the same command with required package name to install the additional packages.

install.packages("xlsx")

## Verify and Load the "xlsx" Package

Use the following command to verify and load the "xlsx" package.

# Verify the package is installed.

any(grepl("xlsx",installed.packages()))

# Load the library into R workspace.

library("xlsx")

When the script is run we get the following output.

[1] TRUE

Loading required package: rJava

Loading required package: methods

Loading required package: xlsxjars

## Input as xlsx File

Open Microsoft excel. Copy and paste the following data in the work sheet named as sheet1.

id name salary start\_date dept

1 Rick 623.3 1/1/2012 IT

2 Dan 515.2 9/23/2013 Operations

3 Michelle 611 11/15/2014 IT

4 Ryan 729 5/11/2014 HR

5 Gary 843.25 3/27/2015 Finance

6 Nina 578 5/21/2013 IT

7 Simon 632.8 7/30/2013 Operations

8 Guru 722.5 6/17/2014 Finance

Also copy and paste the following data to another worksheet and rename this worksheet to "city".

name city

Rick Seattle

Dan Tampa

Michelle Chicago

Ryan Seattle

Gary Houston

Nina Boston

Simon Mumbai

Guru Dallas

Save the Excel file as "input.xlsx". You should save it in the current working directory of the R workspace.

## Reading the Excel File

The input.xlsx is read by using the **read.xlsx()** function as shown below. The result is stored as a data frame in the R environment.

# Read the first worksheet in the file input.xlsx.

data <- read.xlsx("input.xlsx", sheetIndex = 1)

print(data)

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

id, name, salary, start\_date, dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 NA Gary 843.25 2015-03-27 Finance

6 6 Nina 578.00 2013-05-21 IT

7 7 Simon 632.80 2013-07-30 Operations

8 8 Guru 722.50 2014-06-17 Finance

# R - Pie Charts

R Programming language has numerous libraries to create charts and graphs. A pie-chart is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart.

In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input. The additional parameters are used to control labels, color, title etc.

## Syntax

The basic syntax for creating a pie-chart using the R is −

pie(x, labels, radius, main, col, clockwise)

Following is the description of the parameters used −

* **x** is a vector containing the numeric values used in the pie chart.
* **labels** is used to give description to the slices.
* **radius** indicates the radius of the circle of the pie chart.(value between −1 and +1).
* **main** indicates the title of the chart.
* **col** indicates the color palette.
* **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

## Example

A very simple pie-chart is created using just the input vector and labels. The below script will create and save the pie chart in the current R working directory.

# Create data for the graph.

x <- c(21, 62, 10, 53)

labels <- c("London", "New York", "Singapore", "Mumbai")

# Give the chart file a name.

png(file = "city.jpg")

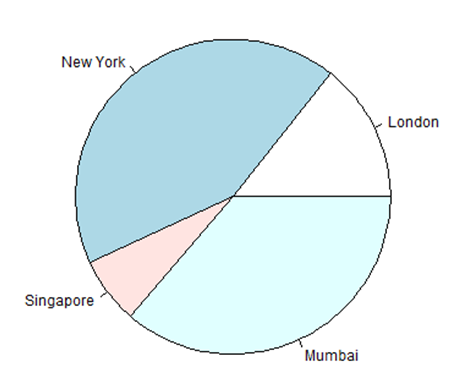
# Plot the chart.

pie(x,labels)

# Save the file.

dev.off()

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



## Pie Chart Title and Colors

We can expand the features of the chart by adding more parameters to the function. We will use parameter **main** to add a title to the chart and another parameter is **col** which will make use of rainbow colour pallet while drawing the chart. The length of the pallet should be same as the number of values we have for the chart. Hence we use length(x).

## Example

The below script will create and save the pie chart in the current R working directory.

# Create data for the graph.

x <- c(21, 62, 10, 53)

labels <- c("London", "New York", "Singapore", "Mumbai")

# Give the chart file a name.

png(file = "city\_title\_colours.jpg")

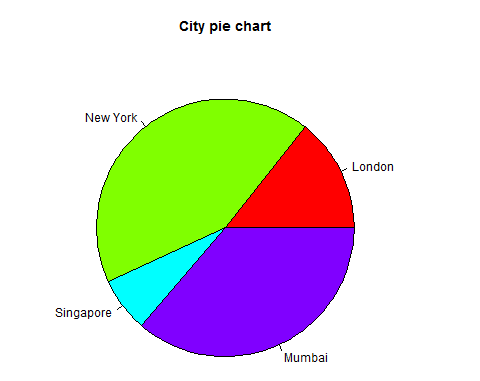
# Plot the chart with title and rainbow color pallet.

pie(x, labels, main = "City pie chart", col = rainbow(length(x)))

# Save the file.

dev.off()

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



## Slice Percentages and Chart Legend

We can add slice percentage and a chart legend by creating additional chart variables.

# Create data for the graph.

x <- c(21, 62, 10,53)

labels <- c("London","New York","Singapore","Mumbai")

piepercent<- round(100\*x/sum(x), 1)

# Give the chart file a name.

png(file = "city\_percentage\_legends.jpg")

# Plot the chart.

pie(x, labels = piepercent, main = "City pie chart",col = rainbow(length(x)))

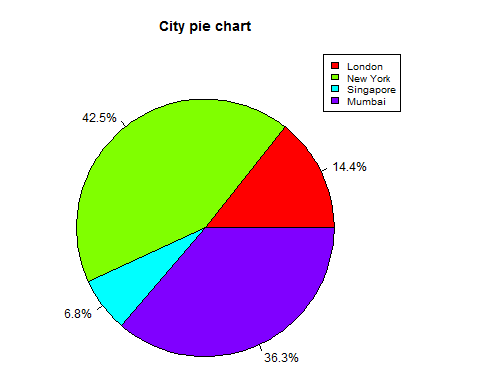
legend("topright", c("London","New York","Singapore","Mumbai"), cex = 0.8,

fill = rainbow(length(x)))

# Save the file.

dev.off()

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



## 3D Pie Chart

A pie chart with 3 dimensions can be drawn using additional packages. The package **plotrix** has a function called **pie3D()** that is used for this.

# Get the library.

library(plotrix)

# Create data for the graph.

x <- c(21, 62, 10,53)

lbl <- c("London","New York","Singapore","Mumbai")

# Give the chart file a name.

png(file = "3d\_pie\_chart.jpg")

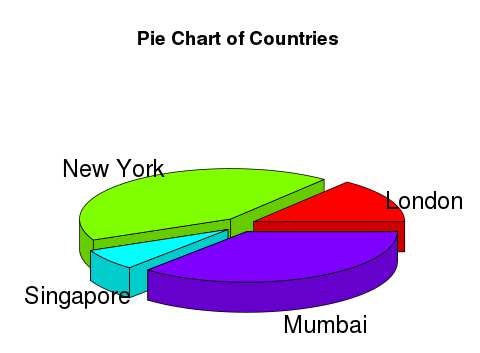
# Plot the chart.

pie3D(x,labels = lbl,explode = 0.1, main = "Pie Chart of Countries ")

# Save the file.

dev.off()

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



# R - Bar Charts

A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable. R uses the function **barplot()** to create bar charts. R can draw both vertical and horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

## Syntax

The basic syntax to create a bar-chart in R is −

barplot(H, xlab, ylab, main, names.arg, col)

Following is the description of the parameters used −

* **H** is a vector or matrix containing numeric values used in bar chart.
* **xlab** is the label for x axis.
* **ylab** is the label for y axis.
* **main** is the title of the bar chart.
* **names.arg** is a vector of names appearing under each bar.
* **col** is used to give colors to the bars in the graph.

## Example

A simple bar chart is created using just the input vector and the name of each bar.

The below script will create and save the bar chart in the current R working directory.

# Create the data for the chart.

H <- c(7,12,28,3,41)

# Give the chart file a name.

png(file = "barchart.png")

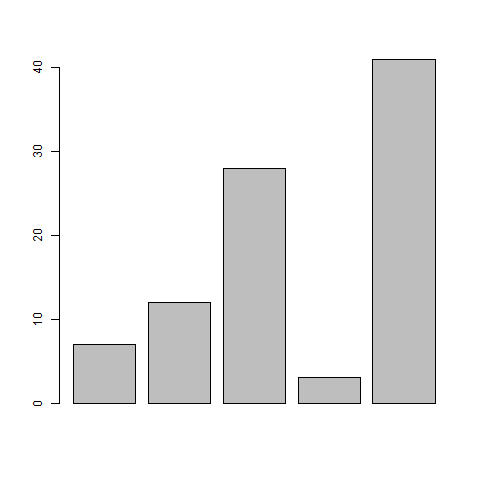
# Plot the bar chart.

barplot(H)

# Save the file.

dev.off()

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



## Bar Chart Labels, Title and Colors

The features of the bar chart can be expanded by adding more parameters. The**main** parameter is used to add **title**. The **col** parameter is used to add colors to the bars. The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.

## Example

The following script will create and save the bar chart in the current R working directory.

# Create the data for the chart.

H <- c(7,12,28,3,41)

M <- c("Mar","Apr","May","Jun","Jul")

# Give the chart file a name.

png(file = "barchart\_months\_revenue.png")

# Plot the bar chart.

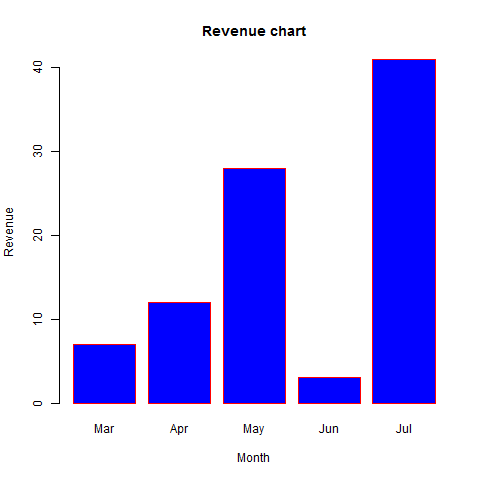
barplot(H,names.arg = M,xlab = "Month",ylab = "Revenue",col = "blue",

main = "Revenue chart",border = "red")

# Save the file.

dev.off()

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



## Group Bar Chart and Stacked Bar Chart

We can create bar chart with groups of bars and stacks in each bar by using a matrix as input values.

More than two variables are represented as a matrix which is used to create the group bar chart and stacked bar chart.

# Create the input vectors.

colors <- c("green","orange","brown")

months <- c("Mar","Apr","May","Jun","Jul")

regions <- c("East","West","North")

# Create the matrix of the values.

Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11),nrow = 3,ncol = 5,byrow = TRUE)

# Give the chart file a name.

png(file = "barchart\_stacked.png")

# Create the bar chart.

barplot(Values,main = "total revenue",names.arg = months,xlab = "month",ylab = "revenue",

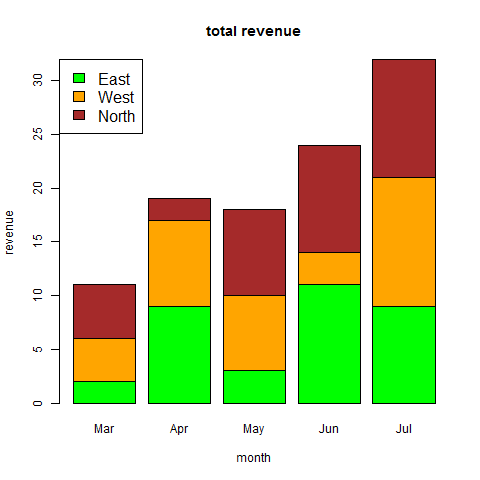
col = colors)

# Add the legend to the chart.

legend("topleft", regions, cex = 1.3, fill = colors)

# Save the file.

dev.off()



# R - Line Graphs

A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

## Syntax

The basic syntax to create a line chart in R is −

plot(v,type,col,xlab,ylab)

Following is the description of the parameters used −

* **v** is a vector containing the numeric values.
* **type** takes the value "p" to draw only the points, "i" to draw only the lines and "o" to draw both points and lines.
* **xlab** is the label for x axis.
* **ylab** is the label for y axis.
* **main** is the Title of the chart.
* **col** is used to give colors to both the points and lines.

## Example

A simple line chart is created using the input vector and the type parameter as "O". The below script will create and save a line chart in the current R working directory.

# Create the data for the chart.

v <- c(7,12,28,3,41)

# Give the chart file a name.

png(file = "line\_chart.jpg")

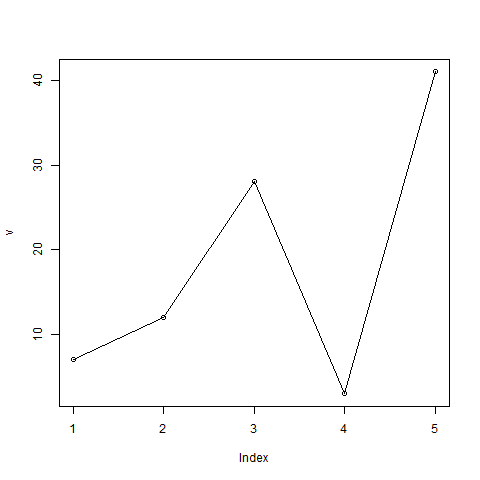
# Plot the bar chart.

plot(v,type = "o")

# Save the file.

dev.off()

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



## Line Chart Title, Color and Labels

The features of the line chart can be expanded by using additional parameters. We add color to the points and lines, give a title to the chart and add labels to the axes.

## Example

# Create the data for the chart.

v <- c(7,12,28,3,41)

# Give the chart file a name.

png(file = "line\_chart\_label\_colored.jpg")

# Plot the bar chart.

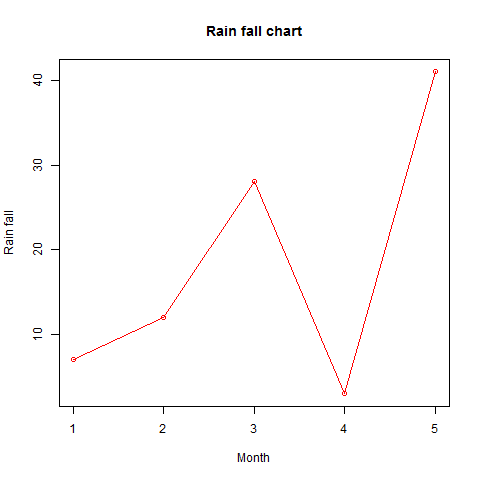
plot(v,type = "o", col = "red", xlab = "Month", ylab = "Rain fall",

main = "Rain fall chart")

# Save the file.

dev.off()

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



## Multiple Lines in a Line Chart

More than one line can be drawn on the same chart by using the**lines()**function.

After the first line is plotted, the lines() function can use an additional vector as input to draw the second line in the chart,

# Create the data for the chart.

v <- c(7,12,28,3,41)

t <- c(14,7,6,19,3)

# Give the chart file a name.

png(file = "line\_chart\_2\_lines.jpg")

# Plot the bar chart.

plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",

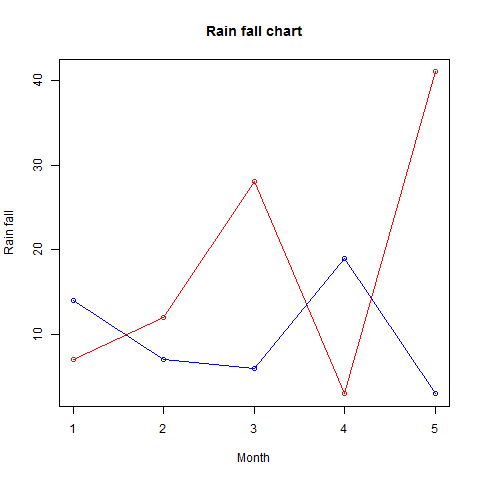
main = "Rain fall chart")

lines(t, type = "o", col = "blue")

# Save the file.

dev.off()

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



# R - Histograms

A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chat but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

R creates histogram using **hist()** function. This function takes a vector as an input and uses some more parameters to plot histograms.

## Syntax

The basic syntax for creating a histogram using R is −

hist(v,main,xlab,xlim,ylim,breaks,col,border)

Following is the description of the parameters used −

* **v** is a vector containing numeric values used in histogram.
* **main** indicates title of the chart.
* **col** is used to set color of the bars.
* **border** is used to set border color of each bar.
* **xlab** is used to give description of x-axis.
* **xlim** is used to specify the range of values on the x-axis.
* **ylim** is used to specify the range of values on the y-axis.
* **breaks** is used to mention the width of each bar.

## Example

A simple histogram is created using input vector, label, col and border parameters.

The script given below will create and save the histogram in the current R working directory.

# Create data for the graph.

v <- c(9,13,21,8,36,22,12,41,31,33,19)

# Give the chart file a name.

png(file = "histogram.png")

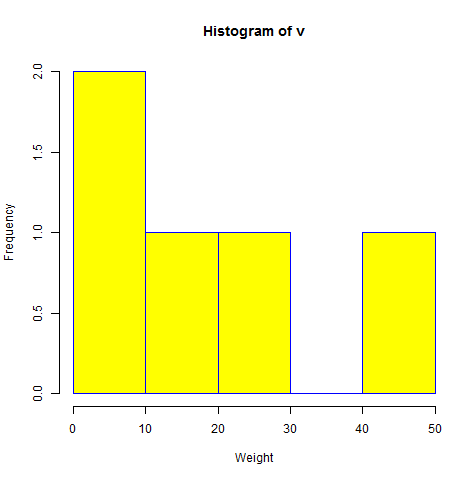
# Create the histogram.

hist(v,xlab = "Weight",col = "yellow",border = "blue")

# Save the file.

dev.off()

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



## Range of X and Y values

To specify the range of values allowed in X axis and Y axis, we can use the xlim and ylim parameters.

The width of each of the bar can be decided by using breaks.

# Create data for the graph.

v <- c(9,13,21,8,36,22,12,41,31,33,19)

# Give the chart file a name.

png(file = "histogram\_lim\_breaks.png")

# Create the histogram.

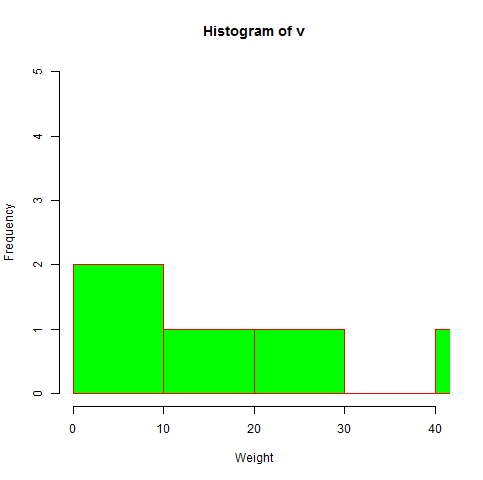
hist(v,xlab = "Weight",col = "green",border = "red", xlim = c(0,40), ylim = c(0,5),

breaks = 5)

# Save the file.

dev.off()

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



# R - Mean, Median & Mode

Statistical analysis in R is performed by using many in-built functions. Most of these functions are part of the R base package. These functions take R vector as an input along with the arguments and give the result.

The functions we are discussing in this chapter are mean, median and mode.

## Mean

It is calculated by taking the sum of the values and dividing with the number of values in a data series.

The function **mean()** is used to calculate this in R.

## Syntax

The basic syntax for calculating mean in R is −

mean(x, trim = 0, na.rm = FALSE, ...)

Following is the description of the parameters used −

* **x** is the input vector.
* **trim** is used to drop some observations from both end of the sorted vector.
* **na.rm** is used to remove the missing values from the input vector.

### Example

# Create a vector.

x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# Find Mean.

result.mean <- mean(x)

print(result.mean)

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

[1] 8.22

## Applying Trim Option

When trim parameter is supplied, the values in the vector get sorted and then the required numbers of observations are dropped from calculating the mean.

When trim = 0.3, 3 values from each end will be dropped from the calculations to find mean.

In this case the sorted vector is (−21, −5, 2, 3, 4.2, 7, 8, 12, 18, 54) and the values removed from the vector for calculating mean are (−21,−5,2) from left and (12,18,54) from right.

# Create a vector.

x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# Find Mean.

result.mean <- mean(x,trim = 0.3)

print(result.mean)

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

[1] 5.55

## Applying NA Option

If there are missing values, then the mean function returns NA.

To drop the missing values from the calculation use na.rm = TRUE. which means remove the NA values.

# Create a vector.

x <- c(12,7,3,4.2,18,2,54,-21,8,-5,NA)

# Find mean.

result.mean <- mean(x)

print(result.mean)

# Find mean dropping NA values.

result.mean <- mean(x,na.rm = TRUE)

print(result.mean)

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

[1] NA

[1] 8.22

## Median

The middle most value in a data series is called the median. The **median()**function is used in R to calculate this value.

### Syntax

The basic syntax for calculating median in R is −

median(x, na.rm = FALSE)

Following is the description of the parameters used −

* **x** is the input vector.
* **na.rm** is used to remove the missing values from the input vector.

### Example

# Create the vector.

x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# Find the median.

median.result <- median(x)

print(median.result)

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

[1] 5.6

## Mode

The mode is the value that has highest number of occurrences in a set of data. Unike mean and median, mode can have both numeric and character data.

R does not have a standard in-built function to calculate mode. So we create a user function to calculate mode of a data set in R. This function takes the vector as input and gives the mode value as output.

### Example

# Create the function.

getmode <- function(v) {

uniqv <- unique(v)

uniqv[which.max(tabulate(match(v, uniqv)))]

}

# Create the vector with numbers.

v <- c(2,1,2,3,1,2,3,4,1,5,5,3,2,3)

# Calculate the mode using the user function.

result <- getmode(v)

print(result)

# Create the vector with characters.

charv <- c("o","it","the","it","it")

# Calculate the mode using the user function.

result <- getmode(charv)

print(result)

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

[1] 2

[1] "it"

# R - Linear Regression

Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called predictor variable whose value is gathered through experiments. The other variable is called response variable whose value is derived from the predictor variable.

In Linear Regression these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

The general mathematical equation for a linear regression is −

y = ax + b

Following is the description of the parameters used −

* **y** is the response variable.
* **x** is the predictor variable.
* **a** and **b** are constants which are called the coefficients.

## Steps to Establish a Regression

A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

* Carry out the experiment of gathering a sample of observed values of height and corresponding weight.
* Create a relationship model using the **lm()** functions in R.
* Find the coefficients from the model created and create the mathematical equation using these
* Get a summary of the relationship model to know the average error in prediction. Also called **residuals**.
* To predict the weight of new persons, use the **predict()** function in R.

### Input Data

Below is the sample data representing the observations −

# Values of height

151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight.

63, 81, 56, 91, 47, 57, 76, 72, 62, 48

## lm() Function

This function creates the relationship model between the predictor and the response variable.

## Syntax

The basic syntax for **lm()** function in linear regression is −

lm(formula,data)

Following is the description of the parameters used −

* **formula** is a symbol presenting the relation between x and y.
* **data** is the vector on which the formula will be applied.

## Create Relationship Model & get the Coefficients

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

print(relation)

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

Call:

lm(formula = y ~ x)

Coefficients:

(Intercept) x

-38.4551 0.6746

## Get the Summary of the Relationship

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

print(summary(relation))

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

Call:

lm(formula = y ~ x)

Residuals:

Min 1Q Median 3Q Max

-6.3002 -1.6629 0.0412 1.8944 3.9775

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -38.45509 8.04901 -4.778 0.00139 \*\*

x 0.67461 0.05191 12.997 1.16e-06 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.253 on 8 degrees of freedom

Multiple R-squared: 0.9548, Adjusted R-squared: 0.9491

F-statistic: 168.9 on 1 and 8 DF, p-value: 1.164e-06

## predict() Function

### Syntax

The basic syntax for predict() in linear regression is −

predict(object, newdata)

Following is the description of the parameters used −

* **object** is the formula which is already created using the lm() function.
* **newdata** is the vector containing the new value for predictor variable.

### Predict the weight of new persons

# The predictor vector.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

# The resposne vector.

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

# Find weight of a person with height 170.

a <- data.frame(x = 170)

result <- predict(relation,a)

print(result)

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

1

76.22869

### Visualize the Regression Graphically

# Create the predictor and response variable.

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

relation <- lm(y~x)

# Give the chart file a name.

png(file = "linearregression.png")

# Plot the chart.

plot(y,x,col = "blue",main = "Height & Weight Regression",

abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")

# Save the file.

dev.off()

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

