R Language

**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.

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.

This programming language was named **R**, based on the initals letter of the two R authors (Robert Gentleman and Ross Ihaka).

Audience

* Software programmers
* Statisticians and
* Data miners

who are looking forward for developing statistical software using R programming. If you are trying to understand the R programming language as a beginner, this tutorial will give you enough understanding on almost all the concepts of the language from where you can take yourself to higher levels of expertise.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages will help you in understanding the R programming concepts and move fast on the learning track.

# 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 copy left, 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. The following are the 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 on 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.

# R - Environment Setup

## Local Environment Setup

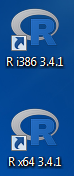
If you are still willing to set up your environment for R, you can follow the steps given below.

### 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 "C:\Program Files\R\R-3.4.1\bin\x64\Rgui.exe" under the Windows Program Files. Clicking this icon brings up the R-GUI which is the R console to do R Programming or double click R desktop icon.



### Linux Installation

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

The instruction to install Linux varies from flavor to flavor. These steps are mentioned under each type of Linux version in the mentioned link. However, if you are in a 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 the required package. For example, the following command will install **plotrix** package which is required for 3D charts.

> install.packages("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 an R script file to write your program. Let's check both one by one.

## R Command Prompt

Once you have R environment setup, then it’s easy to start your R command prompt by just typing 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!"

str <- "Hari Yadav"

print (str)

[1] "Hari Yadav"

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.

> 24+7+11

[1] 42

The answer is 42. R gives you one other piece of information: The [1] preceding 42 indicates that the value 42 is the first element in your answer.

**HOW R DOES INDEXING**

Every time R shows you a vector (learn later on), it displays a number such as [1] in front of the output. In this example, [1] tells you where the first position in your vector is.

This number is called the index of that value. If you make a longer vector — say, with the numbers from 1 to 30 — you see more indices. Consider this example:

numbers <- 30:1

numbers

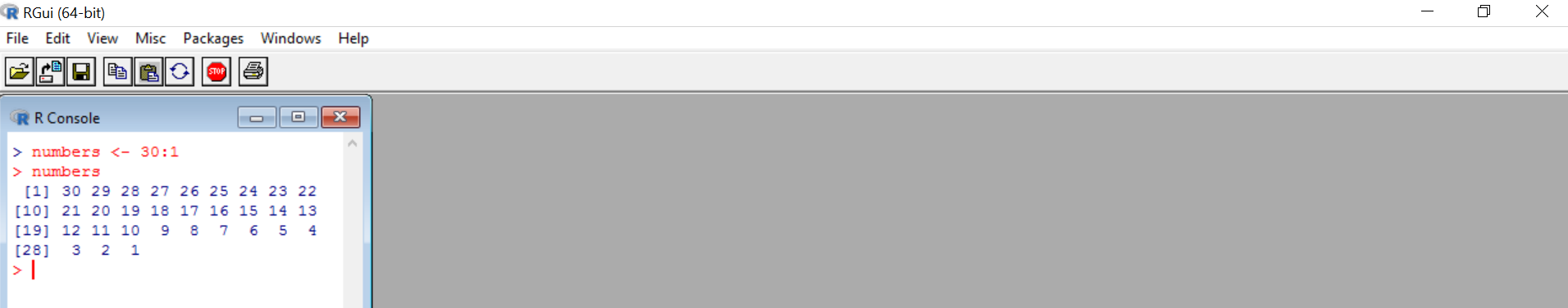
[1] 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14

[18] 13 12 11 10 9 8 7 6 5 4 3 2 1

Here, you see that R counts 13 as the 18th value in the vector. At the beginning of every line, R tells you the index of the first value in that line.

**NOTE:**

**If you try this example on your computer, you may see a different index at the beginning of the line, depending on the width of your console.**

****

**Extract Values From A Vector**

Brackets ([]) illustrate another strong point of R. They represent a function that you can use to extract a value from that vector. You can get the fifth value of the preceding number vector:

numbers[5]

[1] 26

numbers[c(5,11,3)]

[1] 26 20 28

You also can store the indices you want to retrieve in another vector and give that vector as an argument.

indices <- c(5,11,3)

numbers[indices]

[1] 26 20 28

Retrive all values except 3

numbers[-3]

[1] 30 29 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

If you want to expel the first 20 numbers, use this code:

numbers[-(1:20)]

[1] 10 9 8 7 6 5 4 3 2 1

**COMBINE LOGICAL STATEMENTS**

v1 <- c(**12**,4,4,6,9,3)

v2 <- c(5,3,2,2,12,9)

min.vector <- v1 == min(v1)

max.vector <- v1 == max(v1)

min.vector

[1] FALSE FALSE FALSE FALSE FALSE TRUE

max.vector

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

**Combine both vectors with the OR operator (|)**

min.vector | max.vector

[1] TRUE FALSE FALSE FALSE FALSE TRUE

## 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 perform a trick which is something as follows −

if(FALSE) {

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

print("Multi line Comment Section")

}

myString <- "Hello, World!"

print ( myString)

# To see the list of objects created.

ls()

[1] "myString"

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

## R Script File run on OS prompt

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** as under −

# My first program in R Programming

myString <- "Hello, World!"

print ( myString)

Save the 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.

On Linux

$ Rscript test.R

On window

Open command prompt and switch to directory where R language is installed.

Microsoft Windows [Version 6.1.7600]

Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>**chdir** C:\Program Files\R\R-3.4.1\bin\x64

List files from this directory

C:\Program Files\R\R-3.4.1\bin\x64>dir

Volume in drive C has no label.

Volume Serial Number is 789E-9B6B

Directory of C:\Program Files\R\R-3.4.1\bin\x64

08/27/2017 03:42 PM <DIR> .

08/27/2017 03:42 PM <DIR> ..

06/30/2017 01:46 PM 18,944 open.exe

06/30/2017 01:45 PM 30,778,880 R.dll

06/30/2017 01:46 PM 104,448 R.exe

06/30/2017 01:46 PM 315,866 Rblas.dll

06/30/2017 01:46 PM 104,448 Rcmd.exe

06/30/2017 01:46 PM 88,576 Rfe.exe

06/30/2017 01:45 PM 321,501 Rgraphapp.dll

06/30/2017 01:46 PM 88,576 Rgui.exe

06/30/2017 01:45 PM 66,785 Riconv.dll

06/30/2017 01:48 PM 2,734,592 Rlapack.dll

06/30/2017 01:46 PM 92,160 Rscript.exe

06/30/2017 01:46 PM 89,600 RSetReg.exe

06/30/2017 01:46 PM 88,576 Rterm.exe

08/27/2017 10:04 AM 87 Test.txt

14 File(s) 34,893,039 bytes

2 Dir(s) 19,192,541,184 bytes free

C:\Program Files\R\R-3.4.1\bin\x64>Rscript test.txt

[1] "Hello, World!"

OR absolute path

C:\Program Files\R\R-3.4.1\bin\x64>Rscript "C:\Users\Administrator\Desktop\R\_Language\Test.txt"

[1] "Hello, World!"

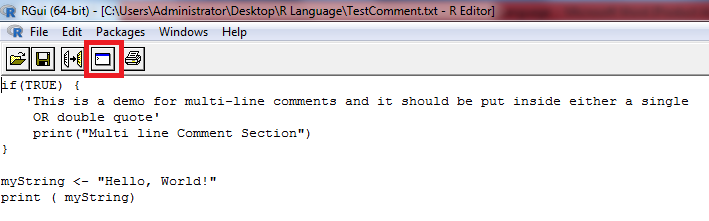
C:\Program Files\R\R-3.4.1\bin\x64>

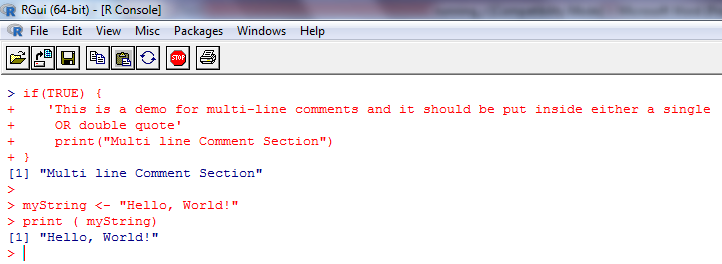
When we run the above program, it produces the following result.

[1] "Hello, World!"

## R Script File run on R Workspace prompt

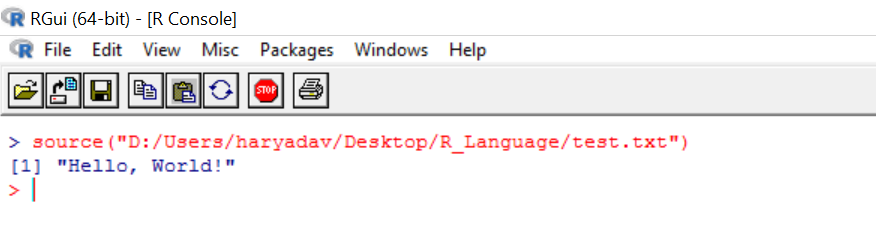
* Open R (Double Click on Desktop Icon or From Start menu > All Programms > R
* Click on **File → Open Script**
* Select the Program you want to run, it will appear in a **R Editor Window**
* Right Click **Select All** (or Type **Ctrl-A)**
* Right Click **Run Line or Selection** (or Type **Ctrl-R)**
* Output will appear in **R Console Window**





* Graphics will be routed to file you create in **pdf** statement

Alternatively you can execute from RGui



## GETTING HELP WITH R

The R Help files provide detailed information about the use of different functions and their peculiarities. R has excellent built-in help for every function that explains with examples how to use that function.

To search through the Help files, you’ll use one of the following functions:

1. ? Displays the Help file for a specific function. For example:

**?data.frame** displays the Help file for the data.frame() function.

1. ?? Searches for a word (or pattern) in the Help files. For example:

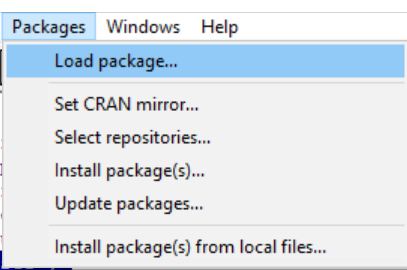
**??list** returns the names of functions that contain the word list in either the function names or their descriptions.

1. **RSiteSearch()** Performs an online search of RSiteSearch. This search engine allows you to perform a search of the R functions, package etc. For example:

**RSiteSearch("linear models")** It will take you to the R Site Search website for the search term “linear models.”

1. You aren’t limited to the R Help files if you’re looking for help with R. The add-on package sos, available for download from CRAN [**here**](http://cran.r-project.org/web/packages/sos/index.html), has some neat functions to search all the Help files on RSiteSearch. It displays results in a web browser window, making it easy to work with.

To use the package sos, you need to install the package by typing **install.packages("sos")** in your R console, and then load the package with library("sos").



Then you can use the findFn() function to do your search. For example: findFn("regression") into your R console, you get a web page with the names, descriptions and links to several hundred functions that contain the word regression in the function name or Help text description.

findFn("regression")

findFn("array")

# R - Data Types

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.

V\_name varchar(10)

In R, the variables are not declared as some data type like C, C++ or java. 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 these atomic vectors.

A vector is a sequence of data elements of the same basic type. Members in a vector are officially called components.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Example** | **Verify** |
| Logical | TRUE, FALSE | v <- TRUE  print(v)  print(class(v))  v <- T  print(v)  print(class(v))  it produces the following result −  [1] TRUE  [1] "logical" |
| Numeric | 12.3, 5, 999 | v <- 23.5  print(v)  print(class(v))  it produces the following result −  [1] 23.5  [1] "numeric" |
| Integer | 2L, 34L, 0L | v <- 2L  print(v)  print(class(v))  it produces the following result −  [1] 2  [1] "integer" |
| Integer | Integer do not contain decimal | v <- 2.89L  Warning message:  integer literal 2.89L contains decimal; using numeric value  print(v)  print(class(v))  [1] 2.89  [1] "numeric" |
| Complex | 3 + 2i | v <- 2+5i  print(v)  print(class(v))  it produces the following result −  [1] 2+5i  [1] "complex" |
| Character | 'a' , "good", "TRUE", '23.4' | v <- "TRUE"  print(v)  print(class(v))  v <- c('a','good','TRUE','23.3')  v <- "'a','good','TRUE','23.3'"  it produces the following result −  [1] "TRUE"  [1] "character" |
| Character | 'a' , "good", "TRUE", '23.4' | v <- "'TRUE'  + 'FALSE'  + 'NULL'  + 'UNKNOWN'"  > print(v)  > print(class(v))  [1] "'TRUE'\n'FALSE'\n'NULL'\n'UNKNOWN'"  [1] "character"  Here \n means new line |

|  |  |  |
| --- | --- | --- |
| Character | 'a' , '"good", "TRUE", '23.4' | v <- '"good", "TRUE", "23.4"'  print(v)  print(class(v))  [1] "\"good\", \"TRUE\", \"23.4\""  [1] "character" |
| Character | 'a' , '"good", "TRUE", '23.4'  c function used to combine the elements | v <- **c**("good", "TRUE", "23.4")  print(v)  print(class(v))  [1] "good" "TRUE" "23.4"  [1] "character" |
| Raw | "Hello" is stored as 48 65 6c 6c 6f | v <- charToRaw("Hello")  print(v)  print(class(v))  it produces the following result −  [1] 48 65 6c 6c 6f  [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.

## CALCULATES INFINITE, UNDEFINED, AND MISSING VALUES

R can deal with data anomalies that confound some other statistical platforms. For instance, in some cases, you don’t have real values to calculate with. In most real-life data sets, in fact, at least a few values are missing. Also, some calculations have infinity as a result (such as dividing by zero) or can’t be carried out at all (such as taking the logarithm of a negative value).

## DEFINES INFINITY

2/0

[1] Inf

4 - Inf

[1] –Inf

Inf/0

[1] Inf

R correctly tells you the result is Inf, or infinity. Negative infinity is shown as -Inf.

R considers everything larger than the largest number a computer can hold to be infinity — on most machines, that’s approximately 1.8 × 10308. This definition of infinity can lead to unexpected results, as shown in the following example:

is.finite(10^(305:310))

[1] TRUE TRUE TRUE TRUE FALSE FALSE

is.infinite(10^(305:310))

[1] FALSE FALSE FALSE FALSE TRUE TRUE

## DEALS WITH UNDEFINED OUTCOMES

Your math teacher probably explained that if you divide any real number by infinity, you get zero. But what if you divide infinity by infinity?

Inf / Inf

[1] NaN

R tells you that the outcome is NaN. That result simply means Not a Number. This is R’s way of telling you that the outcome of that calculation is not defined.

The funny thing is that R actually considers NaN to be numeric, so you can use NaN in calculations. The outcome of those calculations is always NaN

NaN + 4

[1] NaN

You can test whether a calculation results in NaN by using the is.nan() function. Note that both is.finite() and is.infinite() return FALSE when you’re testing on a NaN value.

is.nan(NaN + 4)

[1] TRUE

is.finite(NaN + 4)

[1] FALSE

is.infinite(NaN + 4)

[1] FALSE

## DEALS WITH MISSING VALUES

One of the most common problems in statistics is incomplete data sets. To deal with missing values, R uses the reserved keyword NA, which stands for Not Available. You can use NA as a valid value, so you can assign it as a value as well:

x <- NA

x

[1] NA

You have to take into account, however, that calculations with a value of NA also generally return NA as a result:

x + 4

[1] NA

log(x)

[1] NA

If you want to test whether a value is NA, you can use the is.na() function, as follows:

is.na(x)

[1] TRUE

Note that the is.na() function also returns TRUE if the value is NaN. The functions is.finite(), is.infinite(), and is.nan() return FALSE for NA values.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Inf** | **–Inf** | **NaN** | **NA** |
| **is.finite()** | **FALSE** | **FALSE** | **FALSE** | **FALSE** |
| **is.infinite()** | **TRUE** | **TRUE** | **FALSE** | **FALSE** |
| **is.nan()** | **FALSE** | **FALSE** | **TRUE** | **FALSE** |
| **is.na()** | **FALSE** | **FALSE** | **TRUE** | **TRUE** |

## Vectors

A vector is a sequence of data elements of the same basic type. Members in a vector are officially called components.

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

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.

# Create a vector.

apple <- c('red','green',"yellow")

print(apple)

print(class(apple))

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

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

[1] "character"

apple <- c('red','green',150L)

print(apple)

print(class(apple))

[1] "red" "green" "150"

[1] "character"

v\_num <- c(1:10)

print(v\_num)

print(class(v\_num))

[1] 1 2 3 4 5 6 7 8 9 10

[1] "integer"

## Creating Integer and Double Vectors

By default, when you create a numeric vector using the c() function it will produce a vector of double precision numeric values. To create a vector of integers using c() function you must specify explicity by placing an L directly after each number.

'mode' is a mutually exclusive classification of objects according to their basic structure. The 'atomic' modes are numeric, complex, character and logical. Recursive objects have modes such as 'list' or 'function' or a few others. An object has one and only one mode.

'class' is a property assigned to an object that determines how generic functions operate with it. It is not a mutually exclusive classification. If an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention.

The mode of an object can change without necessarily changing the class.

According to the R documentation for typeof and class. class is an attribute of an object that can be assigned regardless of its internal storage mode, while "typeof determines the (R internal) type or storage mode of any object." One describes a logical characteristic while the other is a physical characteristic of an object.

# create a double-precision vector

dbl\_var <- c(10, 30.5, 40.5)

dbl\_var

print(class(dbl\_var))

typeof(dbl\_var)

mode(dbl\_var)

# placing an L after the values creates a string of integers

int\_var <- c(10L, 60L, 100L)

int\_var

print(class(int\_var))

typeof(int\_var)

mode(int\_var)

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

[1] 10.0 30.5 40.5

[1] "numeric"

[1] "double"

[1] "numeric"

[1] 10 60 100

[1] "integer"

[1] "integer"

[1] "numeric"

# converts integers to double-precision values

as.double(int\_var)

[1] 10 60 100

# identical to as.double()

as.numeric(int\_var)

[1] 10 60 100

# converts doubles to integers

dbl\_var

[1] 10.0 30.5 40.5

as.integer(dbl\_var)

[1] 10 30 40

The main difference between class and typeof is that the first can be defined by the user, but the type cannot.

X <- list("a",c(1,2))

# x is a list

class(x)

[1] "list"

# class can be user defined

class(x)<-"CapgeminiClass"

class(x)

[1] "CapgeminiClass"

typeof(x)

[1] "list"

# you cannot assign a different type using typeof()

typeof(x)<-"newclass"

Error in typeof(x) <- "newclass" : could not find function "typeof<-"

## BASIC OPERATORS IN R

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **x + y** | **y added to x** | **2 + 3 = 5** |
| **x – y** | **y subtracted from x** | **8 – 2 = 6** |
| **x \* y** | **x multiplied by y** | **3 \* 2 = 6** |
| **x / y** | **x divided by y** | **10 / 5 = 2** |
| **x ^ y (or x \*\* y)** | **x raised to the power y** | **2 ^ 5 = 32** |
| **x %% y** | **remainder of x divided by y (x mod y)** | **7 %% 3 = 1 (remainder value)** |
| **x %/% y** | **quotient of x divided by y** | **7 %/% 3 = 2 (quotient value)** |

## 

Most of the basic arithmetic operators are very familiar to programmers.

basketball.score <- c(12,4,4,6,9,3)

football.score <- c(5,3,2,2,12,9)

print(basketball.score)

print(football.score)

or

basketball.score <- c(12,4,4,6,9,3); basketball.score

[1] 12 4 4 6 9 3

football.score <- c(5,3,2,2,12,9); football.score

[1] 5 3 2 2 12 9

bb.score <- basketball.score \* 10 ; bb.score

[1] 120 40 40 60 90 30

fb.score <- football.score \* 20 ; fb.score

[1] 100 60 40 40 240 180

bb.score + fb.score

[1] 220 100 80 100 330 210

x <- c(10,20,31,39,44,45,48,60,77,55,75,90,85,87)

x/2

[1] 5.0 10.0 15.5 19.5 22.0 22.5 24.0 30.0 38.5 27.5 37.5 45.0 42.5 43.5

x\*2

[1] 20 40 62 78 88 90 96 120 154 110 150 180 170 174

x%%10

[1] 0 0 1 9 4 5 8 0 7 5 5 0 5 7

x%/%10

[1] 1 2 3 3 4 4 4 6 7 5 7 9 8 8

x <- c(1:10)

x

[1] 1 2 3 4 5 6 7 8 9 10

x\*\*2

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

or

x^2

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

### Another sets of examples

## 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 or c()**

# Creating a sequence from 5 to 13.

v <- 5:13

print(v)

v <- c(5,6,7,8,9,10,11,12,13)

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] 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))

print(seq(5, 9, 0.4))

print(seq(5, 50, 5))

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

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

[1] 5 10 15 20 25 30 35 40 45 50

**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 or 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 do not have date data types.

dates <- c("06/25/18", "10/05/18")

dates

print(typeof(dates))

dob <- as.Date(dates,format = "%m/%d/%y")

dob

dates <- c("May 27 1984", "July 7 2005")

dates

[1] "May 27 1984" "July 7 2005"

class(dates)

[1] "character"

doj <- as.Date(dates, format = "%B %d %Y")

doj

[1] "1984-05-27" "2005-07-07"

### Date Formats

|  |  |  |
| --- | --- | --- |
| Conversion specification | Description | Example |
| %a | Abbreviated weekday | Sun, Thu |
| %A | Full weekday | Sunday, Thursday |
| %b or %h | Abbreviated month | May, Jul |
| %B | Full month | May, July |
| %d | Day of the month 01-31 | 27, 07 |
| %j | Day of the year 001-366 | 148, 188 |
| %m | Month 01-12 | 05, 07 |
| %U | Week 01-53 with Sunday as first day of the week | 22, 27 |
| %w | Weekday 0-6 Sunday is 0 | 0, 4 |
| %W | Week 00-53 with Monday as first day of the week | 21, 27 |
| %x | Date, locale-specific |  |
| %y | Year without century 00-99 | 84, 05 |
| %Y | Year with century on input: 00 to 68 prefixed by 20 69 to 99 prefixed by 19 | 1984, 2005 |
| %C | Century | 19, 20 |
| %D | Date formatted %m/%d/%y | 05/27/84, 07/07/05 |
| %u | Weekday 1-7 Monday is 1 | 7, 4 |
|  |  |  |
| %n | Newline on output or Arbitrary whitespace on input |  |
| %t | Tab on output or Arbitrary whitespace on input |  |

### References

* help(as.Date)
* help(strptime)

## 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.

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.

# Create a list.

list1 <- list(c(2,5,3),21.3,sin,cos)

print(list1)

print(class(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")

[[4]]

function (x) .Primitive("cos")

[1] "list"

list2 <- list(c(2,5,3),21.3,sin,cos, list(c("East","West","North","South")), list(c("Sun","Moon","Earth"),c("x","y","z")))

print(list2)

print(class(list2))

[[1]]

[1] 2 5 3

[[2]]

[1] 21.3

[[3]]

function (x) .Primitive("sin")

[[4]]

function (x) .Primitive("cos")

[[5]]

[[5]][[1]]

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

[[6]]

[[6]][[1]]

[1] "Sun" "Moon" "Earth"

[[6]][[2]]

[1] "x" "y" "z"

[1] "list"

## 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)

list\_data[5]

[[1]]

[1] 51.23

list\_data[6]

[[1]]

[1] 119.1

list\_data[[5]] \* list\_data[[6]]

[1] 6101.493

typeof(list\_data[[5]])

[1] "double"

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))

print(list\_data)

# 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 −

[[1]]

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

[[2]]

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

[1,] 3 5 -2

[2,] 9 1 8

[[3]]

[[3]][[1]]

[1] "green"

[[3]][[2]]

[1] 12.3

$`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