**MODULE\_1**  
**Introduction to R programming**

* R is an open-source programming language that is widely used as a statistical software and data analysis tool.
* R is a popular programming language used for statistical computing and graphical presentation. Its most common use is to analyze and visualize data.
* R comes with the Command-line interface.
* R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.
* It was designed by **Ross Ihaka and Robert Gentleman** at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.
* R programming language is an implementation of the S programming language.

#### Why R Programming Language is used?

* R programming is used as a tool for machine learning, statistics, and data analysis. Objects, functions, and packages can easily be created by R.
* It’s a platform-independent language. This means it can be applied to all operating system.
* It’s an open-source free language. That means anyone can install it in any organization without purchasing a license.
* R programming language is not only a statistic package but also allows us to integrate with other languages (C, C++). Thus, you can easily interact with many data sources and statistical packages.
* The R programming language has a vast community of users and it’s growing day by day.
* R is currently one of the most requested programming languages in the Data Science job market that makes it the hottest trend nowadays.

#### Features of R Programming Language

1. **Statistical Features of R:**

* **Basic Statistics:** The most common basic statistics terms are the mean, mode, and median. These are all known as “Measures of Central Tendency.” So using the R language we can measure central tendency very easily.
* **Static graphics:** R is rich with facilities for creating and developing interesting static graphics. R contains functionality for many plot types including graphic maps, mosaic plots, biplots, and the list goes on.
* **Probability distributions:** Probability distributions play a vital role in statistics and by using R we can easily handle various types of probability distribution such as Binomial Distribution, Normal Distribution, Chi-squared Distribution and many more.
* **Data analysis:** It provides a large, coherent and integrated collection of tools for data analysis.

1. **Programming Features of R:**

* **R Packages:** One of the major features of R is it has a wide availability of libraries. R has CRAN(Comprehensive R Archive Network), which is a repository holding more than 10, 0000 packages.
* **Distributed Computing:** Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance. Two new packages **ddR and multidplyr** used for distributed programming in R were released in November 2015.

#### Programming in R:

Since R is much similar to other widely used languages syntactically, it is easier to code and learn in R. Programs can be written in R in any of the widely used IDE like **R Studio, Rattle, Tinn-R**, etc. After writing the program save the file with the extension **.r**. To run the program use the following command on the command line:

**Advantages of R:**

1. R is the most comprehensive statistical analysis package.It is a great resource for data analysis, data visualization, data science and machine learning
2. As R programming language is an open source and free. Thus, you can run R anywhere and at any time.
3. R programming language is suitable for GNU/Linux and Windows operating system.
4. R programming is cross-platform which runs on any operating system.
5. In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.
6. It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
7. It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
8. It works on different platforms (Windows, Mac, Linux)
9. It has a large community support
10. It has many packages (libraries of functions) that can be used to solve different problems

**Disadvantages of R:**

* In the R programming language, the standard of some packages is less than perfect.
* Although, R commands give little pressure to memory management. So R programming language may consume all available memory.
* In R basically, nobody to complain if something doesn’t work.
* R programming language is much slower than other programming languages such as Python and MATLAB.

**Applications of R:**

* We use R for Data Science. It gives us a broad variety of libraries related to statistics. It also provides the environment for statistical computing and design.
* R is used by many quantitative analysts as its programming tool. Thus, it helps in data importing and cleaning.
* R is the most prevalent language. So many data analysts and research programmers use it. Hence, it is used as a fundamental tool for finance.
* Tech giants like Google, Facebook, bing, Twitter, Accenture, Wipro and many more using R nowadays.

R and Python both play a major role in data science. It becomes confusing for any newbie to choose the better or the most suitable one among the two, R and Python. So take a look at [R vs Python for Data Science](https://www.geeksforgeeks.org/r-vs-python-datascience/) to choose which language is more suitable for data science.

1. **Installation**:
   * To get started with R, you need to download and install it on your computer. You can obtain the latest version of R from the official website (<https://www.r-project.org/>). Additionally, you may want to install an integrated development environment (IDE) for R, such as RStudio (<https://www.rstudio.com/>), to make your coding experience more convenient.
2. **R Basics**:
   * R is an interpreted language, which means you can execute code interactively in the R console or write scripts.
   * You can perform basic arithmetic operations, assign variables, and create vectors (arrays) easily. For example:

x <- 5 y <- 10 z <- x + y

1. **Data Types**:
   * R supports various data types, including numeric, character, logical, and factors. It also has more advanced data structures like vectors, matrices, data frames, and lists.
2. **Data Manipulation**:
   * R provides extensive functions for data manipulation and transformation. The **dplyr** package, for instance, offers a concise and powerful way to filter, arrange, and summarize data.
3. **Data Visualization**:
   * R excels in data visualization with packages like **ggplot2**. You can create a wide range of high-quality plots and charts to explore and communicate your data effectively.
4. **Statistical Analysis**:
   * R is renowned for its statistical capabilities. It offers numerous built-in statistical functions and libraries, making it a top choice for data analysis and hypothesis testing.
5. **Packages and Libraries**:
   * R's strength lies in its vast ecosystem of packages contributed by the community. You can extend R's functionality by installing and loading packages relevant to your specific task.
6. **Control Structures**:
   * R supports common control structures like loops (for, while), conditional statements (if-else), and functions. You can create custom functions to encapsulate reusable code.
7. **Data Import and Export**:
   * R can read data from various sources, including CSV files, Excel spreadsheets, databases, and web APIs. The **readr** and **readxl** packages are popular choices for data import.
8. **Documentation and Help**:
   * R has extensive documentation available within the language itself, accessible with the **help()** or **?** functions. Additionally, online resources, forums, and communities like Stack Overflow provide ample support.
9. **Reproducibility**:
   * R promotes reproducible research. You can document your analysis steps and create reports using tools like R Markdown, ensuring that others can replicate your work easily.
10. **Version Control**:
    * It's a good practice to use version control systems like Git in combination with R to manage your code, collaborate with others, and track changes in your projects.
11. **Machine Learning**:
    * R has several packages (e.g., **caret**, **randomForest**, **xgboost**) that facilitate machine learning and predictive modeling tasks.
12. **Community and Collaboration**:
    * The R community is active and welcoming. Collaboration with other R users and sharing your own packages and code is encouraged.

**How to Run programs in the R Workspace:**

1. Open R (Double Click on Desktop Icon or Open Program from START)
2. Click on File → Open Script.
3. Select the Program you want to run, it will appear in a R Editor Window.
4. Right Click Select All (or Type Ctrl-A)
5. Right Click Run Line or Selection (or Type Ctrl-R)

R Sessions and Functions

### R Sessions:

1. **R Console**:
   * When you start R, you typically interact with it through the R console. The console allows you to enter and execute R commands and see the results immediately.
2. **Scripts**:
   * In addition to the console, you can write R code in scripts. R scripts are collections of R commands saved in a text file with the ".R" extension. You can run a script by sourcing it, which executes all the commands in the script sequentially.
3. **Interactive Sessions**:
   * R is interactive, meaning you can enter commands one at a time and see the results immediately. This interactivity is beneficial for exploring data and experimenting with code.
4. **Workspace**:
   * R keeps track of your variables, data, and the state of your session in a workspace. You can save and load the workspace to preserve your environment between sessions.
5. **History**:
   * R maintains a history of commands you've executed in the current session. You can access and reuse previous commands from the history.
6. **Session Control**:
   * You can control the R session by saving your workspace, quitting R, or restarting it. This allows you to manage your work effectively.

**R FUNCTIONS:**

**There are mainly three types of function in R programming:**

* **Primitive Functions**

Some core functionality of R is defined using primitive functions, which use a special technique for accessing C-code, for performance reasons. Examples of primitive functions include language elements, like if and for , operators like + and $ , and mathematical functions like exp and sin .

* **Infix Functions**

Infix functions (operators) can help clean up your data model and maintain preferences for an order of operation. Most of the functions in R programming are written with the arguments set within a parenthesis.

* **Replacement Functions.**

The replace() method in R replaces the values in the defined vector or data frame with the indices specified in the list. Replace() is a very simple and straightforward function in R syntax.It also contains the vector, indices vector, and replacement values.

**Basic Math in R:**

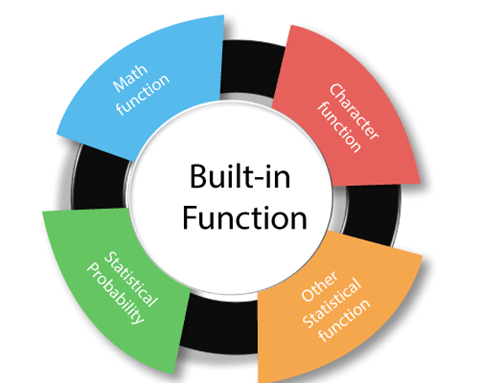
Performing basic math operations in R is straightforward, and it works similarly to most other programming languages. You can use R as a calculator to perform arithmetic operations such as addition, subtraction, multiplication, and division. Here are some basic math operations in R:

1. **Addition (+)**:
2. **Subtraction (-)**:
3. **Multiplication (\*)**:
4. **Division (/)**:
5. **Exponentiation (^)**:
6. **Modulus (remainder) (%%)**:
7. **Integer Division (%/%)**:
8. **Square Root (sqrt())**
9. **Absolute Value (abs())**:
10. **Trigonometric Functions (sin(), cos(), tan())**:
11. **Logarithms (log(), log10())**:
12. **Rounding (round())**:
13. **Floor and Ceiling (floor(), ceiling())**:

You can combine these basic math operations to create more complex expressions. R follows the standard order of operations (PEMDAS/BODMAS), where parentheses have the highest precedence, followed by exponents, multiplication and division (left to right), and addition and subtraction (left to right).

# R Built-in Functions

The functions which are already created or defined in the programming framework are known as a built-in function. R has a rich set of functions that can be used to perform almost every task for the user. These built-in functions are divided into the following categories based on their functionality.



## Math Functions

R provides the various mathematical functions to perform the mathematical calculation. These mathematical functions are very helpful to find absolute value, square value and much more calculations. In R, there are the following functions which are used:

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **Function** | **Description** | **Example** |
| **1.** | abs(x) | It returns the absolute value of input x. | x<- -4  print(abs(x))  **Output**  [1] 4 |
| **2.** | sqrt(x) | It returns the square root of input x. | x<- 4  print(sqrt(x))  **Output**  [1] 2 |
| **3.** | ceiling(x) | It returns the smallest integer which is larger than or equal to x. | x<- 4.5  print(ceiling(x))  **Output**  [1] 5 |
| **4.** | floor(x) | It returns the largest integer, which is smaller than or equal to x. | x<- 2.5  print(floor(x))  **Output**  [1] 2 |
| **5.** | trunc(x) | It returns the truncate value of input x. | x<- c(1.2,2.5,8.1)  print(trunc(x))  **Output**  [1] 1 2 8 |
| **6.** | round(x, digits=n) | It returns round value of input x. | x<- -4  print(abs(x))  **Output**  4 |
| **7.** | cos(x), sin(x), tan(x) | It returns cos(x), sin(x) value of input x. | x<- 4  print(cos(x))  print(sin(x))  print(tan(x))  **Output**  [1] -06536436  [2] -0.7568025  [3] 1.157821 |
| **8.** | log(x) | It returns natural logarithm of input x. | x<- 4  print(log(x))  **Output**  [1] 1.386294 |
| **9.** | log10(x) | It returns common logarithm of input x. | x<- 4  print(log10(x))  **Output**  [1] 0.60206 |
| **10.** | exp(x) | It returns exponent. | x<- 4  print(exp(x))  **Output**  [1] 54.59815 |

## String Function

R provides various string functions to perform tasks. These string functions allow us to extract sub string from string, search pattern etc. There are the following string functions in R:

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **Function** | **Description** | **Example** |
| **1.** | substr(x, start=n1,stop=n2) | It is used to extract substrings in a character vector. | a <- "987654321"  substr(a, 3, 3)  **Output**  [1] "3" |
| **2.** | grep(pattern, x , ignore.case=FALSE, fixed=FALSE) | It searches for pattern in x. | st1 <- c('abcd','bdcd','abcdabcd')  pattern<- '^abc'  print(grep(pattern, st1))  **Output**  [1] 1 3 |
| **3.** | sub(pattern, replacement, x, ignore.case =FALSE, fixed=FALSE) | It finds pattern in x and replaces it with replacement (new) text. | st1<- "England is beautiful but no the part of EU"  sub("England', "UK", st1)  **Output**  [1] "UK is beautiful but not a part of EU" |
| **4.** | paste(..., sep="") | It concatenates strings after using sep string to separate them. | paste('one',2,'three',4,'five')  **Output**  [1] one 2 three 4 five |
| **5.** | strsplit(x, split) | It splits the elements of character vector x at split point. | a<-"Split all the character"  print(strsplit(a, ""))  **Output**  [[1]]  [1] "split" "all" "the" "character" |
| **6.** | tolower(x) | It is used to convert the string into lower case. | st1<- "shuBHAm"  print(tolower(st1))  **Output**  [1] shubham |
| **7.** | toupper(x) | It is used to convert the string into upper case. | st1<- "shuBHAm"  print(toupper(st1))  **Output**  [1] SHUBHAM |

## Other Statistical Function

Apart from the functions mentioned above, there are some other useful functions which helps for statistical purpose. There are the following functions:

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **Function** | **Description** | **Example** |
| **1.** | mean(x, trim=0, na.rm=FALSE) | It is used to find the mean for x object | a<-c(0:10, 40)  xm<-mean(a)  print(xm)  **Output**  [1] 7.916667 |
| **2.** | sd(x) | It returns standard deviation of an object. | a<-c(0:10, 40)  xm<-sd(a)  print(xm)  **Output**  [1] 10.58694 |
| **3.** | median(x) | It returns median. | a<-c(0:10, 40)  xm<-meadian(a)  print(xm)  **Output**  [1] 5.5 |
| **4.** | quantilie(x, probs) | It returns quantile where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0, 1] |  |
| **5.** | range(x) | It returns range. | a<-c(0:10, 40)  xm<-range(a)  print(xm)  **Output**  [1] 0 40 |
| **6.** | sum(x) | It returns sum. | a<-c(0:10, 40)  xm<-sum(a)  print(xm)  **Output**  [1] 95 |
| **7.** | diff(x, lag=1) | It returns differences with lag indicating which lag to use. | a<-c(0:10, 40)  xm<-diff(a)  print(xm)  **Output**  [1] 1 1 1 1 1 1 1 1 1 1 30 |
| **8.** | min(x) | It returns minimum value. | a<-c(0:10, 40)  xm<-min(a)  print(xm)  **Output**  [1] 0 |
| **9.** | max(x) | It returns maximum value | a<-c(0:10, 40)  xm<-max(a)  print(xm)  **Output**  [1] 40 |
| **10.** | scale(x, center=TRUE, scale=TRUE) | Column center or standardize a matrix. | a <- matrix(1:9,3,3)  scale(x)  **Output**  [,1]  [1,] -0.747776547  [2,] -0.653320562  [3,] -0.558864577  [4,] -0.464408592  [5,] -0.369952608  [6,] -0.275496623  [7,] -0.181040638  [8,] -0.086584653  [9,] 0.007871332  [10,] 0.102327317  [11,] 0.196783302  [12,] 3.030462849  attr(,"scaled:center")  [1] 7.916667  attr(,"scaled:scale")  [1] 10.58694 |

# R - Variables

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many Robjects.

A variable is a memory allocated for the storage of specific data and the name associated with the variable is used to work around this reserved block.

The name given to a variable is known as its variable name. Usually a single variable stores only the data belonging to a certain data type.

The name is so given to them because when the program executes there is subject to change hence it varies from time to time.

A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

### Declaring and Initializing Variables in R Language

**R supports three ways of variable assignment:**

* Using equal operator- operators use an arrow or an equal sign to assign values to variables.
* Using the leftward operator- data is copied from right to left.
* Using the rightward operator- data is copied from left to right.

### R Variables Syntax

Types of Variable Creation in R:

* *Using equal to operators  
    variable\_name = value*
* *using leftward operator  
   variable\_name <- value*
* *using rightward operator   
   value -> variable\_name*

### Creating Variables in R

|  |
| --- |
| R program to illustrate Initialization of variable:  # **using equal to operator**  var1 = "hello"  print(var1)  # **using leftward operator**  var2 <- "hello"  print(var2)  # **using rightward operator**  "hello" -> var3  print(var3) |

**Output**

[1] "hello"

[1] "hello"

[1] "hello"

## Nomenclature of R Variables

The following rules need to be kept in mind while naming a R variable:

* A valid variable name consists of a combination of alphabets, numbers, dot(.), and underscore(\_) characters. Example: var.1\_ is valid
* Apart from the dot and underscore operators, no other special character is allowed. Example: var$1 or var#1 both are invalid
* Variables can start with alphabets or dot characters. Example: .var or var is valid
* The variable should not start with numbers or underscore. Example: 2var or \_var is invalid.
* If a variable starts with a dot the next thing after the dot cannot be a number. Example: .3var is invalid
* The variable name should not be a reserved keyword in R. Example: TRUE, FALSE,etc.

## Important Methods for R Variables

R provides some useful methods to perform operations on variables. These methods are used to determine the data type of the variable, finding a variable, deleting a variable, etc. Following are some of the methods used to work on variables:

### class() function

This built-in function is used to determine the data type of the variable provided to it. The R variable to be checked is passed to this as an argument and it prints the data type in return.

**Syntax**

class(variable)

**Example**

|  |
| --- |
| var1 = "hello"  print(class(var1)) |

**Output**

[1] "character"

### ls() function

This built-in function is used to know all the present variables in the workspace. This is generally helpful when dealing with a large number of variables at once and helps prevents overwriting any of them.

**Syntax**

ls()

**Example**

|  |
| --- |
| # using equal to operator  var1 = "hello"  # using leftward operator  var2 <- "hello"  # using rightward operator  "hello" -> var3  print(ls()) |

**Output:**

[1] "var1" "var2" "var3"

### rm() function

This is again a built-in function used to delete an unwanted variable within your workspace. This helps clear the memory space allocated to certain variables that are not in use thereby creating more space for others. The name of the variable to be deleted is passed as an argument to it.

**Syntax**

rm(variable)

**Example**

|  |
| --- |
| # using equal to operator  var1 = "hello"  # using leftward operator  var2 <- "hello"  # using rightward operator  "hello" -> var3  # Removing variable  rm(var3)  print(var3) |

**Output**

Error in print(var3) : object 'var3' not found

Execution halted

## Variable Assignment

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

Assignment using equal operator.

var.1 = c(0,1,2,3)

# Assignment using leftward operator.

var.2 <- c("learn","R")

# Assignment using rightward operator.

c(TRUE,1) -> var.3

print(var.1)

cat ("var.1 is ", var.1 ,"\n")

cat ("var.2 is ", var.2 ,"\n")

cat ("var.3 is ", var.3 ,"\n")

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

[1] 0 1 2 3

var.1 is 0 1 2 3

var.2 is learn R

var.3 is 1 1

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

## Data Type of a Variable

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

var\_x <- "Hello"

cat("The class of var\_x is ",class(var\_x),"\n")

var\_x <- 34.5

cat(" Now the class of var\_x is ",class(var\_x),"\n")

var\_x <- 27L

cat(" Next the class of var\_x becomes ",class(var\_x),"\n")

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

The class of var\_x is character

Now the class of var\_x is numeric

Next the class of var\_x becomes integer

## Finding Variables

To know all the variables currently available in the workspace we use the **ls()** function. Also the ls() function can use patterns to match the variable names.

print(ls())

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

[1] "my var" "my\_new\_var" "my\_var" "var.1"

[5] "var.2" "var.3" "var.name" "var\_name2."

[9] "var\_x" "varname"

**Note** − It is a sample output depending on what variables are declared in your environment.

The ls() function can use patterns to match the variable names.

# List the variables starting with the pattern "var".

print(ls(pattern = "var"))

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

[1] "my var" "my\_new\_var" "my\_var" "var.1"

[5] "var.2" "var.3" "var.name" "var\_name2."

[9] "var\_x" "varname"

The variables starting with **dot(.)** are hidden, they can be listed using "all.names = TRUE" argument to ls() function.

print(ls(all.name = TRUE))

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

[1] ".cars" ".Random.seed" ".var\_name" ".varname" ".varname2"

[6] "my var" "my\_new\_var" "my\_var" "var.1" "var.2"

[11]"var.3" "var.name" "var\_name2." "var\_x"

## Deleting Variables

Variables can be deleted by using the **rm()** function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

rm(var.3)

print(var.3)

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

[1] "var.3"

Error in print(var.3) : object 'var.3' not found

All the variables can be deleted by using the **rm()** and **ls()** function together.

rm(list = ls())

print(ls())

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

character(0)

## Scope of Variables in R programming

The location where we can find a variable and also access it if required is called the [scope of a variable](https://www.geeksforgeeks.org/scope-of-variable-in-r/). There are mainly two types of variable scopes:

### 1.Global Variables

Global variables are those variables that exist throughout the execution of a program. It can be changed and accessed from any part of the program.

As the name suggests, Global Variables can be accessed from any part of the program.

* They are available throughout the lifetime of a program.
* They are declared anywhere in the program outside all of the functions or blocks.

**Declaring global variables**

Global variables are usually declared outside of all of the functions and blocks. They can be accessed from any portion of the program

|  |
| --- |
| # R program to illustrate  # usage of global variables  # global variable  global = 5  # global variable accessed from  # within a function  display = function(){  print(global)  }  display()  # changing value of global variable  global = 10  display() |

**Output**

[1] 5

[1] 10

In the above code, the variable ‘**global’** is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

### Local Variables

Local variables are those variables that exist only within a certain part of a program like a function and are released when the function call ends. Local variables do not exist outside the block in which they are declared, i.e. they can not be accessed or used outside that block.

**Declaring local variables**

Local variables are declared inside a block.

|  |
| --- |
| # R program to illustrate usage of local variables  func = function(){  # this variable is local to the  # function func() and cannot be  # accessed outside this function  age = 18  print(age)  }    cat("Age is:\n")  func() |

**Output**

Age is:

[1] 18

**Difference between local and global variables in R**

1. **Scope** A global variable is defined outside of any function and may be accessed from anywhere in the program, as opposed to a local variable.
2. **Lifetime** A local variable’s lifetime is constrained by the function in which it is defined. The local variable is destroyed once the function has finished running. A global variable, on the other hand, doesn’t leave memory until the program is finished running or the variable is explicitly deleted.
3. **Naming conflicts** If the same variable name is used in different portions of the program, they may occur since a global variable can be accessed from anywhere in the program. Contrarily, local variables are solely applicable to the function in which they are defined, reducing the likelihood of naming conflicts.
4. **Memory usage**Because global variables are kept in memory throughout program execution, they can eat up more memory than local variables. Local variables, on the other hand, are created and destroyed only when necessary, therefore they normally use less memory.

### ****What are R Data types?****

R Data types are used in computer programming to specify the kind of data that can be stored in a variable. For effective memory consumption and precise computation, the right data type must be selected. Each R data type has its own set of regulations and restrictions.

### ****Data Types in R Programming Language:****

Each variable in R has an associated data type. Each R-Data Type requires different amounts of memory and has some specific operations which can be performed over it. [R Programming language](https://www.geeksforgeeks.org/r-programming-language-introduction/) has the following basic R-data types and the following table shows the data type and the values that each data type can take.

| **Basic Data Types** | **Values** | **Examples** |
| --- | --- | --- |
| Numeric | Set of all real numbers | "numeric\_value <- 3.14" |
| Integer | Set of all integers, Z | "integer\_value <- 42L" |
| Logical | TRUE and FALSE | "logical\_value <- TRUE" |
| Complex | Set of complex numbers | "complex\_value <- 1 + 2i" |
| Character | “a”, “b”, “c”, …, “@”, “#”, “$”, …., “1”, “2”, …etc | "character\_value <- "Hello Geeks" |
| raw | as.raw() | "single\_raw <- as.raw(255)" |

### Numeric Data type in R

Decimal values are called numerics in R. It is the default  R data type for numbers in R. If you assign a decimal value to a variable x as follows, x will be of numeric type. Real numbers with a decimal point are represented using this data type in R. it uses a format for double-precision floating-point numbers to represent numerical values

|  |
| --- |
| # A simple R program  # to illustrate Numeric data type    # Assign a decimal value to x  x = 5.6  # print the class name of variable  print(class(x))  # print the type of variable  print(typeof(x)) |

**Output**

[1] "numeric"

[1] "double"

Even if an integer is assigned to a variable y, it is still saved as a numeric value.

* R

|  |
| --- |
| # A simple R program  # to illustrate Numeric data type  # Assign an integer value to y  y = 5  # print the class name of variable  print(class(y))  # print the type of variable  print(typeof(y)) |

**Output**

[1] "numeric"

[1] "double"

When R stores a number in a variable, it converts the number into a “double” value or a decimal type with at least two decimal places. This means that a value such as “5” here, is stored as 5.00 with a type of double and a class of numeric. And also y is not an integer here can be confirmed with the **is.integer()** function.

* R

|  |
| --- |
| # A simple R program  # to illustrate Numeric data type  # Assign a integer value to y  y = 5  # is y an integer?  print(is.integer(y)) |

**Output**

[1] FALSE

### Integer Data type in R

R supports integer data types which are the set of all integers. You can create as well as convert a value into an integer type using the **as.integer()** function. You can also use the capital ‘L’ notation as a suffix to denote that a particular value is of the integer R data type.

|  |
| --- |
| # A simple R program to illustrate integer data type  # Create an integer value  x = as.integer(5)    # print the class name of x  print(class(x))    # print the type of x  print(typeof(x))    # Declare an integer by appending an L suffix.  y = 5L    # print the class name of y  print(class(y))    # print the type of y  print(typeof(y)) |

**Output**

[1] "integer"

[1] "integer"

[1] "integer"

[1] "integer"

### Logical Data type in R

R has logical data types that take either a value of true or false. A logical value is often created via a comparison between variables. Boolean values, which have two possible values, are represented by this R data type: FALSE or TRUE

* R

|  |
| --- |
| # A simple R program to illustrate logical data type    # Sample values  x = 4  y = 3    # Comparing two values  z = x > y  # print the logical value  print(z)    # print the class name of z  print(class(z))    # print the type of z  print(typeof(z)) |

**Output**

[1] TRUE

[1] "logical"

[1] "logical"

### Complex Data type in R

R supports complex data types that are set of all the complex numbers. The complex data type is to store numbers with an imaginary component.

* R

|  |
| --- |
| # A simple R program  # to illustrate complex data type    # Assign a complex value to x  x = 4 + 3i  # print the class name of x  print(class(x))    # print the type of x  print(typeof(x)) |

**Output**

[1] "complex"

[1] "complex"

### Character Data type in R

R supports character data types where you have all the alphabets and special characters. It stores character values or strings. Strings in R can contain alphabets, numbers, and symbols. The easiest way to denote that a value is of character type in R data type is to wrap the value inside single or double inverted commas.

* R

|  |
| --- |
| # A simple R program to illustrate character data type  # Assign a character value to char  char = "Geeksforgeeks"    # print the class name of char  print(class(char))    # print the type of char  print(typeof(char)) |

**Output**

[1] "character"

[1] "character"

There are several tasks that can be done using R data types. Let’s understand each task with its action and the syntax for doing the task along with an R code to illustrate the task.

### ****Raw data type in R****

To save and work with data at the byte level in R, use the raw data type. By displaying a series of unprocessed bytes, it enables low-level operations on binary data. Here are some speculative data on R’s raw data types:

* R

|  |
| --- |
| # Create a raw vector  x <- as.raw(c(0x1, 0x2, 0x3, 0x4, 0x5))  print(x) |

**Output:**

[1] 01 02 03 04 05

Five elements make up this raw vector x, each of which represents a raw byte value.

### Find data type of an object in R

To find the data type of an object you have to use **class()** function. The syntax for doing that is you need to pass the object as an argument to the function **class()** to find the data type of an object.

**Syntax**

class(object)

**Example**

|  |
| --- |
| # A simple R program to find data type of an object    # Logical  print(class(TRUE))    # Integer  print(class(3L))    # Numeric  print(class(10.5))    # Complex  print(class(1+2i))    # Character  print(class("12-04-2020")) |

**Output**

[1] "logical"

[1] "integer"

[1] "numeric"

[1] "complex"

[1] "character"

### Type verification

To do that, you need to use the prefix “is.” before the data type as a command. The syntax for that is, **is.data\_type()** of the object you have to verify.

**Syntax:**

is.data\_type(object)

**Example**

|  |
| --- |
| A simple R program Verify if an object is of a certain datatype  **# Logical**  print(is.logical(TRUE))  **# Integer**  print(is.integer(3L))  **# Numeric**  print(is.numeric(10.5))  **# Complex**  print(is.complex(1+2i))  **# Character**  print(is.character("12-04-2020"))  print(is.integer("a"))  print(is.numeric(2+3i)) |

**Output**

[1] TRUE

[1] TRUE

[1] TRUE

[1] TRUE

[1] TRUE

[1] FALSE

[1] FALSE

### Coerce or convert the data type of an object to another

The process of altering the data type of an object to another type is referred to as coercion or data type conversion. This is a common operation in many programming languages that is used to alter data and perform various computations. When coercion is required, the language normally performs it automatically, whereas conversion is performed directly by the programmer.

Coercion can manifest itself in a variety of ways, depending on the [R programming language](https://www.geeksforgeeks.org/r-programming-language-introduction/) and the context in which it is employed. In some circumstances, the coercion is implicit, which means that the language will change one type to another without the programmer having to expressly request it.

* **Syntax**

as.data\_type(object)

**Note:**All the coercions are not possible and if attempted will be returning an “NA” value.

**Example**

|  |
| --- |
| #A simple R program convert data type of an object to another  # Logical  print(as.numeric(TRUE))  # Integer  print(as.complex(3L))    # Numeric  print(as.logical(10.5))    # Complex  print(as.character(1+2i))    # Can't possible  print(as.numeric("12-04-2020")) |

**Output**

[1] 1

[1] 3+0i

[1] TRUE

[1] "1+2i"

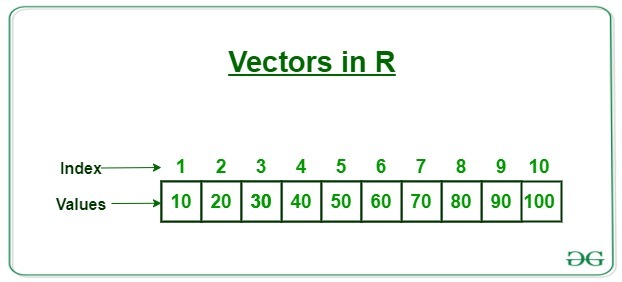
[1] NA

Warning message:

In print(as.numeric("12-04-2020")) : NAs introduced by coercion

**R vectors**

R vectors are the same as the arrays in C language which are used to hold multiple data values of the same type. One major key point is that in R the indexing of the vector will start from ‘1’ and not from ‘0’. We can create numeric vectors and character vectors as well.



*R – Vector*

## Types of R vectors

Vectors are of different types which are used in R. Following are some of the types of vectors:

**1.Numeric vectors:** Numeric vectors are those which contain numeric values such as integer, float, etc.

|  |
| --- |
| # R program to create numeric Vectors creation of vectors using c() function.  v1<- c(4, 5, 6, 7)  # display type of vector  typeof(v1)  # by using 'L' we can specify that we want integer values.  v2<- c(1L, 4L, 2L, 5L)  # display type of vector  typeof(v2) |

**Output:**

[1] "double"

[1] "integer"

**2.Character vectors:**Character vectors in R contain alphanumeric values and special characters.

|  |
| --- |
| # R program to create Character Vectorsby default numeric values are converted into characters  v1<- c('geeks', '2', 'hello', 57)  # Displaying type of vector  typeof(v1) |

**Output:**

[1] "character"

**Logical vectors:** Logical vectors in R contain Boolean values such as TRUE, FALSE and NA for Null values.

|  |
| --- |
| # R program to create Logical Vectors Creating logical vector  # using c() function  v1<- c(TRUE, FALSE, TRUE, NA)  # Displaying type of vector  typeof(v1) |

**Output:**

[1] "logical"

### Creating a vector

There are different ways of creating R vectors. Generally, we use ‘c’ to combine different elements together.

|  |
| --- |
| # R program to create Vectors# we can use the c function# to combine the values as a vector. By default the type will be double  X<- c(61, 4, 21, 67, 89, 2)  cat('using c function', X, '\n')  # seq() function for creating  # a sequence of continuous values.  # length.out defines the length of vector.  Y<- seq(1, 10, length.out = 5)  cat('using seq() function', Y, '\n')  # use':' to create a vector  # of continuous values.  Z<- 2:7  cat('using colon', Z) |

**Output:**

using c function 61 4 21 67 89 2

using seq() function 1 3.25 5.5 7.75 10

using colon 2 3 4 5 6 7

### Length of R vector

* R

|  |
| --- |
| # Create a numeric vector  x <- c(1, 2, 3, 4, 5)  # Find the length of the vector  length(x)  # Create a character vector  y <- c("apple", "banana", "cherry")  # Find the length of the vector  length(y)  # Create a logical vector  z <- c(TRUE, FALSE, TRUE, TRUE)  # Find the length of the vector  length(z) |

**Output:**

> length(x)

[1] 5

> length(y)

[1] 3

> length(z)

[1] 4

### Accessing R vector elements

Accessing elements in a vector is the process of performing operation on an individual element of a vector. There are many ways through which we can access the elements of the vector. The most common is using the ‘[]’, symbol.

|  |
| --- |
| # R program to access elements of a Vector accessing elements with an index number.  X<- c(2, 5, 18, 1, 12)  cat('Using Subscript operator', X[2], '\n')  # by passing a range of values  # inside the vector index.  Y<- c(4, 8, 2, 1, 17)  cat('Using combine() function', Y[c(4, 1)], '\n') |

**Output:**

Using Subscript operator 5

Using combine() function 1 4

### Modifying a R vector

Modification of a Vector is the process of applying some operation on an individual element of a vector to change its value in the vector. There are different ways through which we can modify a vector:

|  |
| --- |
| # R program to modify elements of a Vector  # Creating a vector  X<- c(2, 7, 9, 7, 8, 2)    # modify a specific element  X[3] <- 1  X[2] <-9  cat('subscript operator', X, '\n')    # Modify using different logics.  X[1:5]<- 0  cat('Logical indexing', X, '\n')    # Modify by specifying  # the position or elements.  X<- X[c(3, 2, 1)]  cat('combine() function', X) |

**Output:**

subscript operator 2 9 1 7 8 2

Logical indexing 0 0 0 0 0 2

combine() function 0 0 0

### Deleting a R vector

Deletion of a Vector is the process of deleting all of the elements of the vector. This can be done by assigning it to a NULL value.

|  |
| --- |
| # R program to delete a Vector  # Creating a Vector  M<- c(8, 10, 2, 5)  # set NULL to the vector  M<- NULL  cat('Output vector', M) |

**Output:**

Output vector NULL

### Sorting elements of a R Vector

**sort()** function is used with the help of which we can sort the values in ascending or descending order.

|  |
| --- |
| # R program to sort elements of a Vector Creation of Vector  X<- c(8, 2, 7, 1, 11, 2)  # Sort in ascending order  A<- sort(X)  cat('ascending order', A, '\n')  # sort in descending order  # by setting decreasing as TRUE  B<- sort(X, decreasing = TRUE)  cat('descending order', B) |

**Output:**

ascending order 1 2 2 7 8 11

descending order 11 8 7 2 2 1

**CONCLUSION**

In R, the concept of a "conclusion" is not a built-in feature like in a formal report or document. However, in the context of data analysis or statistical research, you often draw conclusions based on your analysis and findings. These conclusions are typically documented in your analysis report, presentation, or code comments to communicate the results of your work effectively. Here's how you can summarize your analysis and draw conclusions in R:

1. **Summarize Findings**:
   * After performing data analysis or running statistical tests, summarize your findings in a clear and concise manner. This may include key statistics, insights, and observations.
2. **Visualization**:
   * Use data visualization tools like ggplot2 to create plots and charts that visually represent your findings. Visualizations can help convey complex information more effectively.
3. **Interpret Results**:
   * Interpret the results of your analysis in the context of the research or problem you're addressing. Explain the implications of your findings and why they are relevant.
4. **Statistical Significance**:
   * If you conducted statistical tests, report the significance level (e.g., p-value) and explain whether the results are statistically significant. This helps assess the reliability of your conclusions.
5. **Provide Context**:
   * Consider providing context for your conclusions. Explain how your findings relate to the broader research question, hypothesis, or problem statement.
6. **Limitations and Assumptions**:
   * Acknowledge any limitations or assumptions made during your analysis. Discuss potential sources of bias or uncertainty that might affect your conclusions.
7. **Recommendations**:
   * If appropriate, offer recommendations or actions based on your conclusions. What should be done next, or what decisions can be made based on your analysis?
8. **Code Comments**:
   * If you're documenting your analysis in an R script, use comments to explain your code, especially parts related to data processing, calculations, and findings. This makes your code more understandable to others (and your future self).

Here's an example of how you might summarize your findings and draw conclusions in an R analysis script or report:

R # Load necessary libraries and data

library(ggplot2)

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

# Data analysis and visualization

# ...

# Conclusion

cat("In this analysis, we examined the data from a survey of 500 participants.\n")

cat("Our analysis revealed that:\n")

cat("- The average age of participants is 35 years.\n")

cat("- There is a significant positive correlation between income and education level (p < 0.05).\n")

cat("- 80% of respondents prefer product A over product B.\n")

# Recommendations

cat("Based on these findings, we recommend:\n")

cat("- Targeting younger demographics in marketing campaigns to increase product A sales.\n")

cat("- Investing in educational programs to boost income levels among our target audience.\

**Advanced Data Structures in R**

R offers a variety of advanced data structures beyond the basic ones like vectors and matrices. These advanced data structures are useful for handling more complex data and solving more advanced data analysis tasks. Here are some of the advanced data structures in R:

1. **Lists**:
   * Lists are versatile data structures that can store elements of different data types, including vectors, matrices, other lists, and even functions. They are particularly useful when you need to store heterogeneous data.

my\_list <- list(name = "John", age = 30, scores = c(90, 85, 78))

1. **Data Frames**:

* Data frames are two-dimensional tables that resemble a spreadsheet or a SQL table. They are used to store and manipulate tabular data, where columns can have different data types.

# Creating a data frame

df <- data.frame(

Name = c("Alice", "Bob", "Charlie"),

Age = c(25, 30, 22),

Score = c(85, 92, 78)

)

1. **Matrices**:

* While matrices are a basic data structure, they are still considered advanced because they store data in a two-dimensional format, which is essential for many numerical and statistical operations.

my\_matrix <- matrix(1:9, nrow = 3, ncol = 3)

1. **Factors**:

* Factors are used for categorical data, and they represent the different levels or categories within a variable. Factors are essential for statistical analysis.

gender <- factor(c("Male", "Female", "Male", "Female"))

1. **Arrays**:

* Arrays are multi-dimensional data structures, similar to matrices but with more than two dimensions. They are less commonly used than matrices but can be essential for certain advanced applications.

my\_array <- array(1:24, dim = c(2, 3, 4))

1. **Data Tables**:

* Data tables, provided by the **data.table** package, are an enhanced version of data frames optimized for handling large datasets. They offer fast data manipulation and subsetting.

library(data.table)

dt <- data.table(

Name = c("Alice", "Bob", "Charlie"),

Age = c(25, 30, 22),

Score = c(85, 92, 78)

)

1. **Lists of Data Frames**:

* Sometimes, you may need to work with lists where each element is a data frame. This structure is useful for handling multiple datasets or tables.

list\_of\_dfs <- list(df1, df2, df3)

1. **Sparse Matrices**:

* Sparse matrices are matrices that contain mostly zero values. They are used to efficiently store and manipulate large matrices with many zero entries. The **Matrix** package provides tools for working with sparse matrices.

library(Matrix)

sparse\_mat <- sparseMatrix(i = c(1, 2, 3), j = c(2, 1, 3), x = c(2, 3, 1))

**9.Time Series Objects**:

* R provides specialized data structures for time series data. The **ts** (time series) and **xts** (extensible time series) objects are commonly used to represent and analyze time series data, which includes timestamped observations.

library(xts)

timestamp <- seq(from = as.Date("2021-01-01"), by = "month", length.out = 12)

values <- c(100, 120, 110, 130, 115, 125, 105, 140, 130, 150, 120, 135)

ts\_data <- xts(values, order.by = timestamp)

**10.Spatial Data Structures**:

* If you work with geographic or spatial data, R offers advanced data structures like **SpatialPoints**, **SpatialLines**, and **SpatialPolygons** from the **sp** package. These are used to represent and analyze spatial information.

library(sp)

coordinates <- cbind(c(1, 2, 3), c(4, 5, 6))

spatial\_points <- SpatialPoints(coordinates)

**11.Networks and Graphs**:

* When dealing with network data, you can use data structures from packages like **igraph** to create and analyze graphs and networks. These structures include vertices, edges, and various graph algorithms.

library(igraph)

vertices <- c("A", "B", "C", "D")

edges <- data.frame(from = c("A", "A", "B", "C"), to = c("B", "C", "C", "D"))

graph <- graph\_from\_data\_frame(edges, vertices = vertices)

1. **Dictionaries and Named Lists**:
   * R allows you to create dictionaries and named lists, which are useful for associating values with specific keys or labels. This can be handy when you need to store data in a key-value pair format.

R  
# Creating a named list (dictionary)

my\_dict <- list(apple = 3, banana = 5, orange = 2)

1. **Environments**:
   * Environments are special data structures that store named objects (like variables and functions) and their values. They are used for managing and isolating variables in different scopes.

R

my\_env <- new.env()

my\_env$x <- 10

1. **S4 and Reference Classes**:

* R supports object-oriented programming (OOP) with advanced data structures like S4 classes and Reference Classes. These structures allow you to create custom classes and methods for your specific needs.

setClass("Person", representation(name = "character", age = "numeric"))

person <- new("Person", name = "Alice", age = 30)

1. **Ragged Arrays**:

* Ragged arrays are arrays where each element can have a different length. They are useful for representing irregular or nested data structures.

**ragged\_array <- list(c(1, 2, 3), c(4, 5), c(6, 7, 8, 9))**

## Data Frames

Data Frames are data displayed in a format as a table.

Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.

Use the data.frame() function to create a data frame:

### Example

# Create a data frame  
Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
# Print the data frame  
Data\_Frame

**Summarize the Data:**

Use the summary() function to summarize the data from a Data Frame:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
Data\_Frame  
summary(Data\_Frame)

## Access Items:

We can use single brackets [ ], double brackets [[ ]] or $ to access columns from a data frame:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
Data\_Frame[1]  
  
Data\_Frame[["Training"]]  
  
Data\_Frame$Training

**Add Rows:**

Use the rbind() function to add new rows in a Data Frame:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
**# Add a new row:**  
New\_row\_DF <- rbind(Data\_Frame, c("Strength", 110, 110))  
  
**# Print the new row:**  
New\_row\_DF

## Add Columns:

Use the cbind() function to add new columns in a Data Frame:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
**# Add a new column**:  
New\_col\_DF <- cbind(Data\_Frame, Steps = c(1000, 6000, 2000))  
  
**# Print the new column**:  
New\_col\_DF

## Remove Rows and Columns:

Use the c() function to remove rows and columns in a Data Frame:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
**# Remove the first row and column:**  
Data\_Frame\_New <- Data\_Frame[-c(1), -c(1)]  
 **# Print the new data frame:**  
Data\_Frame\_New

## Amount of Rows and Columns:

Use the dim() function to find the amount of rows and columns in a Data Frame:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
dim(Data\_Frame)

You can also use the ncol() function to find the number of columns and nrow() to find the number of rows:

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
ncol(Data\_Frame)  
nrow(Data\_Frame)

**Data Frame Length:**

Use the length() function to find the number of columns in a Data Frame (similar to ncol()):

### Example

Data\_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
length(Data\_Frame)

**Combining Data Frames**

Use the rbind() function to combine two or more data frames in R vertically:

### Example

Data\_Frame1 <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
Data\_Frame2 <- data.frame (  
  Training = c("Stamina", "Stamina", "Strength"),  
  Pulse = c(140, 150, 160),  
  Duration = c(30, 30, 20)  
)  
  
New\_Data\_Frame <- rbind(Data\_Frame1, Data\_Frame2)  
New\_Data\_Frame

And use the cbind() function to combine two or more data frames in R horizontally:

### Example

Data\_Frame3 <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
Data\_Frame4 <- data.frame (  
  Steps = c(3000, 6000, 2000),  
  Calories = c(300, 400, 300)  
)  
  
New\_Data\_Frame1 <- cbind(Data\_Frame3, Data\_Frame4)  
New\_Data\_Frame1

**R -LIST**

A list in [R](https://www.geeksforgeeks.org/r-programming-language-introduction/)is a generic object consisting of an ordered collection of objects. Lists are one-dimensional, heterogeneous [data structures](https://www.geeksforgeeks.org/data-structures-in-r-programming/). The list can be a list of [vectors](https://www.geeksforgeeks.org/r-vector/), a list of matrices, a list of characters and a list of [functions](https://www.geeksforgeeks.org/functions-in-r-programming/), and so on.

A list is a vector but with heterogeneous data elements. A list in R is created with the use of **list()** function. R allows accessing elements of an R list with the use of the index value. In R, the indexing of a list starts with 1 instead of 0 like in other programming languages.

## ****Creating a List****

To create a List in R you need to use the function called “list()”. In other words, a list is a generic vector containing other objects. To illustrate how a list looks, we take an example here. We want to build a list of employees with the details. So for this, we want attributes such as ID, employee name, and the number of employees.

**Example:**

|  |
| --- |
| # R program to create a List. The first attributes is a numeric vector  # containing the employee IDs which is created  # using the command here  empId = c(1, 2, 3, 4)  # The second attribute is the employee name  # which is created using this line of code here  # which is the character vector  empName = c("Debi", "Sandeep", "Subham", "Shiba")  # The third attribute is the number of employees  # which is a single numeric variable.  numberOfEmp = 4  # We can combine all these three different  # data types into a list  # containing the details of employees  # which can be done using a list command  empList = list(empId, empName, numberOfEmp)  print(empList) |

**Output:**

[[1]]

[1] 1 2 3 4

[[2]]

[1] "Debi" "Sandeep" "Subham" "Shiba"

[[3]]

[1] 4

## ****Accessing components of a list:****

We can access components of an R list in two ways.

* **Access components by names:** All the components of a list can be named and we can use those names to access the components of the R list using the dollar command.

**Example:**

|  |
| --- |
| # R program to access  # components of a list  # Creating a list by naming all its components  empId = c(1, 2, 3, 4)  empName = c("Debi", "Sandeep", "Subham", "Shiba")  numberOfEmp = 4  empList = list(    "ID" = empId,    "Names" = empName,    "Total Staff" = numberOfEmp    )print(empList)  # Accessing components by names  cat("Accessing name components using $ command\n")  print(empList$Names) |

**Output:**

$ID

[1] 1 2 3 4

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba"

$`Total Staff`

[1] 4

**Accessing name components using $ command::**

[1] "Debi" "Sandeep" "Subham" "Shiba"

* **Access components by indices:** We can also access the components of the R list using indices. To access the top-level components of a R list we have to use a double slicing operator “**[[ ]]**” which is two square brackets and if we want to access the lower or inner-level components of a R list we have to use another square bracket “**[ ]**” along with the double slicing operator “**[[ ]]**“.

**Example:**

|  |
| --- |
| # R program to access  # components of a list  # Creating a list by naming all its components  empId = c(1, 2, 3, 4)  empName = c("Debi", "Sandeep", "Subham", "Shiba")  numberOfEmp = 4  empList = list(    "ID" = empId,    "Names" = empName,    "Total Staff" = numberOfEmp    )  print(empList)  # Accessing a top level components by indices  cat("Accessing name components using indices\n")  print(empList[[2]])  # Accessing a inner level components by indices  cat("Accessing Sandeep from name using indices\n")  print(empList[[2]][2])  # Accessing another inner level components by indices  cat("Accessing 4 from ID using indices\n")  print(empList[[1]][4]) |

**Output:**

$ID

[1] 1 2 3 4

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba"

$`Total Staff`

[1] 4

**Accessing name components using indices:**

[1] "Debi" "Sandeep" "Subham" "Shiba"

**Accessing Sandeep from name using indices:**

[1] "Sandeep"

**Accessing 4 from ID using indices:**

[1] 4

## ****Modifying components of a list:****

A R list can also be modified by accessing the components and replacing them with the ones which you want.

**Example:**

|  |
| --- |
| # R program to edit  # components of a list  # Creating a list by naming all its components  empId = c(1, 2, 3, 4)  empName = c("Debi", "Sandeep", "Subham", "Shiba")  numberOfEmp = 4  empList = list(    "ID" = empId,    "Names" = empName,    "Total Staff" = numberOfEmp  )  cat("Before modifying the list\n")  print(empList)  # Modifying the top-level component  empList$`Total Staff` = 5  # Modifying inner level component  empList[[1]][5] = 5  empList[[2]][5] = "Kamala"  cat("After modified the list\n")  print(empList) |

**Output:**

Before modifying the list

$ID

[1] 1 2 3 4

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba"

$`Total Staff`

[1] 4

After modified the list

$ID

[1] 1 2 3 4 5

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba" "Kamala"

$`Total Staff`

[1] 5

## ****Concatenation of lists:****

Two R lists can be concatenated using the concatenation function. So, when we want to concatenate two lists we have to use the concatenation operator.

**Syntax:**

*list = c(list, list1)  
list = the original list   
list1 = the new list*

**Example:**

|  |
| --- |
| # R program to edit components of a list.Creating a list by naming all its components  empId = c(1, 2, 3, 4)  empName = c("Debi", "Sandeep", "Subham", "Shiba")  numberOfEmp = 4  empList = list(    "ID" = empId,    "Names" = empName,    "Total Staff" = numberOfEmp  )  cat("Before concatenation of the new list\n")  print(empList)  # Creating another list  empAge = c(34, 23, 18, 45)  # Concatenation of list using concatenation operator  empList = c(empName, empAge)  cat("After concatenation of the new list\n")  print(empList) |

**Output:**

Before concatenation of the new list

$ID

[1] 1 2 3 4

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba"

$`Total Staff`

[1] 4

After concatenation of the new list

[1] "Debi" "Sandeep" "Subham" "Shiba" "34" "23" "18" "45"

## ****Deleting components of a list:****

To delete components of a R list, first of all, we need to access those components and then insert a negative sign before those components. It indicates that we had to delete that component.

**Example:**

|  |
| --- |
| # R program to access components of a list    # Creating a list by naming all its components  empId = c(1, 2, 3, 4)  empName = c("Debi", "Sandeep", "Subham", "Shiba")  numberOfEmp = 4  empList = list(    "ID" = empId,    "Names" = empName,    "Total Staff" = numberOfEmp  )  cat("Before deletion the list is\n")  print(empList)  # Deleting a top level components  cat("After Deleting Total staff components\n")  print(empList[-3])  # Deleting a inner level components  cat("After Deleting sandeep from name\n")  print(empList[[2]][-2]) |

**Output:**

Before deletion the list is

$ID

[1] 1 2 3 4

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba"

$`Total Staff`

[1] 4

After Deleting Total staff components

$ID

[1] 1 2 3 4

$Names

[1] "Debi" "Sandeep" "Subham" "Shiba"

After Deleting sandeep from name

[1] "Debi" "Subham" "Shiba"

## Merging list:

We can merge the R list by placing all the lists into a single list

|  |
| --- |
| # Create two lists.  lst1 <- list(1,2,3)  lst2 <- list("Sun","Mon","Tue")  # Merge the two lists.  new\_list <- c(lst1,lst2)  # Print the merged list.  print(new\_list) |

**Output:**

[[1]]

[1] 1

[[2]]

[1] 2

[[3]]

[1] 3

[[4]]

[1] "Sun"

[[5]]

[1] "Mon"

[[6]]

[1] "Tue"

## Converting List to Vector:

Here we are going to convert the R list to vector, for this we will create a list first and then unlist the list into the vector.

|  |
| --- |
| # Create lists.  lst <- list(1:5)  print(lst)  # Convert the lists to vectors.  vec <- unlist(lst)  print(vec) |

**Output:**

[[1]]

[1] 1 2 3 4 5

[1] 1 2 3 4 5

## R List to matrix:

We will create matrices using matrix() function in R programming. Another function that will be used is unlist() function to convert the lists into a vector.

|  |
| --- |
| # Defining list  lst1 <- list(list(1, 2, 3),              list(4, 5, 6))  # Print list  cat("The list is:\n")  print(lst1)  cat("Class:", class(lst1), "\n")  # Convert list to matrix  mat <- matrix(unlist(lst1), nrow = 2, byrow = TRUE)  # Print matrix  cat("\nAfter conversion to matrix:\n")  print(mat)  cat("Class:", class(mat), "\n") |

**Output:**

The list is:

[[1]]

[[1]][[1]]

[1] 1

[[1]][[2]]

[1] 2

[[1]][[3]]

[1] 3

[[2]]

[[2]][[1]]

[1] 4

[[2]][[2]]

[1] 5

[[2]][[3]]

[1] 6

Class: list

After conversion to matrix:

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

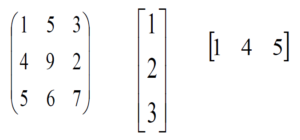
[1,] 1 2 3

[2,] 4 5 6

Class: matrix

**Matrix**

**Matrix** is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. In [R programming](https://www.geeksforgeeks.org/r-programming-language-introduction/), matrices are two-dimensional, homogeneous [data structures](https://www.geeksforgeeks.org/data-structures-in-r-programming/). These are some examples of matrices:



*R – Matrices*

#### Creating a Matrix

To create a matrix in R you need to use the function called **matrix()**. The arguments to this **matrix()** are the set of elements in the vector. You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix.

***Note:****By default, matrices are in column-wise order.*

|  |
| --- |
| # R program to create a matrix  A = matrix( # Taking sequence of elements    c(1, 2, 3, 4, 5, 6, 7, 8, 9),      # No of rows    nrow = 3,    # No of columns    ncol = 3,    # By default matrices are in column-wise order    # So this parameter decides how to arrange the matrix    byrow = TRUE  )  # Naming rows  rownames(A) = c("a", "b", "c")  # Naming columns  colnames(A) = c("c", "d", "e")  cat("The 3x3 matrix:\n")  print(A) |

**Output:**

The 3x3 matrix:

c d e

a 1 2 3

b 4 5 6

c 7 8 9

#### Creating special matrices:

R allows the creation of various different types of matrices with the use of arguments passed to the matrix() function.

* **Matrix where all rows and columns are filled by a single constant ‘k’:**   
  To create such a R matrix the syntax is given below:

***Syntax:****matrix(k, m, n)****Parameters:******k:****the constant****m:****no of rows****n:****no of columns*

**Example:**

|  |
| --- |
| # R program to illustrate special matrices Matrix having 3 rows and 3 columns,filled by a single constant 5  print(matrix(5, 3, 3)) |

**Output:**

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

[1,] 5 5 5

[2,] 5 5 5

[3,] 5 5 5

* **Diagonal matrix:**   
  A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero. To create such a R matrix the syntax is given below:

***Syntax:****diag(k, m, n)****Parameters:******k:****the constants/array****m:****no of rows****n:****no of columns*

**Example:**

|  |
| --- |
| # R program to illustrate special matrices  # Diagonal matrix having 3 rows and 3 columns filled by array of elements (5, 3, 3)  print(diag(c(5, 3, 3), 3, 3)) |

**Output:**

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

[1,] 5 0 0

[2,] 0 3 0

[3,] 0 0 3

* **Identity matrix:**   
  An identity matrix in which all the elements of the principal diagonal are ones and all other elements are zeros. To create such a R matrix the syntax is given below:

***Syntax:****diag(k, m, n)****Parameters:******k:****1****m:****no of rows****n:****no of columns*

**Example:**

|  |
| --- |
| # R program to illustrate special matrices Identity matrix having 3 rows and 3 columns  print(diag(1, 3, 3)) |

**Output:**

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

[1,] 1 0 0

[2,] 0 1 0

[3,] 0 0 1

#### ****Matrix metrics****

Matrix metrics mean once a matrix is created then

* How can you know the dimension of the matrix?
* How can you know how many rows are there in the matrix?
* How many columns are in the matrix?
* How many elements are there in the matrix? are the questions we generally wanted to answer.

**Example:**

|  |
| --- |
| # R program to illustrate matrix metrics Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    cat("Dimension of the matrix:\n")  print(dim(A))    cat("Number of rows:\n")  print(nrow(A))    cat("Number of columns:\n")  print(ncol(A))    cat("Number of elements:\n")  print(length(A))  # OR  print(prod(dim(A))) |

**Output:**

The 3x3 matrix:

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

[1,] 1 2 3

[2,] 4 5 6

[3,] 7 8 9

Dimension of the matrix:

[1] 3 3

Number of rows:

[1] 3

Number of columns:

[1] 3

Number of elements:

[1] 9

[1] 9

#### Accessing elements of a Matrix

We can access elements in the R matrices using the same convention that is followed in data frames. So, you will have a matrix and followed by a square bracket with a comma in between array. Value before the comma is used to access rows and value that is after the comma is used to access columns. Let’s illustrate this by taking a simple R code.

**Accessing rows:**

|  |
| --- |
| # R program to illustrate access rows in metrics. Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Accessing first and second row  cat("Accessing first and second row\n")  print(A[1:2, ]) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

Accessing first and second row

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

[1, ] 1 2 3

[2, ] 4 5 6

**Accessing columns:**

|  |
| --- |
| # R program to illustrate access columns in metrics Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Accessing first and second column  cat("Accessing first and second column\n")  print(A[, 1:2]) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

Accessing first and second column

[, 1] [, 2]

[1, ] 1 2

[2, ] 4 5

[3, ] 7 8

**Accessing elements of a R matrix:**

|  |
| --- |
| # R program to illustrate access an entry in metrics.Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Accessing 2  print(A[1, 2])    # Accessing 6  print(A[2, 3]) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

[1] 2

[1] 6

**Accessing Submatrices:**

We can access the submatrix in a matrix using the **colon(:)** operator.

|  |
| --- |
| # R program to illustrateaccess submatrices in a matrix. Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    cat("Accessing the first three rows and the first two columns\n")  print(A[1:3, 1:2]) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

Accessing the first three rows and the first two columns

[, 1] [, 2]

[1, ] 1 2

[2, ] 4 5

[3, ] 7 8

#### ****Modifying Elements of a Matrix****

In R you can modify the elements of the matrices by a direct assignment.

**Example:**

|  |
| --- |
| # R program to illustrateediting elements in metrics Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Editing the 3rd rows and 3rd column element  # from 9 to 30  # by direct assignments  A[3, 3] = 30    cat("After edited the matrix\n")  print(A) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

After edited the matrix

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 30

#### Matrix Concatenation

Matrix concatenation refers to the merging of rows or columns of an existing R matrix.

**Concatenation of a row:**

The concatenation of a row to a matrix is done using **rbind()**.

|  |
| --- |
| # R program to illustrate concatenation of a row in metrics Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Creating another 1x3 matrix  B = matrix(    c(10, 11, 12),    nrow = 1,    ncol = 3  )  cat("The 1x3 matrix:\n")  print(B)    # Add a new row using rbind()  C = rbind(A, B)    cat("After concatenation of a row:\n")  print(C) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

The 1x3 matrix:

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

[1, ] 10 11 12

After concatenation of a row:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

[4, ] 10 11 12

**Concatenation of a column:**

The concatenation of a column to a matrix is done using **cbind()**.

|  |
| --- |
| # R program to illustrateconcatenation of a column in metricsCreate a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Creating another 3x1 matrix  B = matrix(    c(10, 11, 12),    nrow = 3,    ncol = 1,    byrow = TRUE  )  cat("The 3x1 matrix:\n")  print(B)    # Add a new column using cbind()  C = cbind(A, B)    cat("After concatenation of a column:\n")  print(C) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

The 3x1 matrix:

[, 1]

[1, ] 10

[2, ] 11

[3, ] 12

After concatenation of a column:

[, 1] [, 2] [, 3] [, 4]

[1, ] 1 2 3 10

[2, ] 4 5 6 11

[3, ] 7 8 9 12

**Dimension inconsistency:**Note that you have to make sure the consistency of dimensions between the matrix before you do this matrix concatenation.

|  |
| --- |
| # R program to illustrateDimension inconsistency in metrics concatenation Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("The 3x3 matrix:\n")  print(A)    # Creating another 1x3 matrix  B = matrix(    c(10, 11, 12),    nrow = 1,    ncol = 3,  )  cat("The 1x3 matrix:\n")  print(B)    # This will give an error  # because of dimension inconsistency  C = cbind(A, B)    cat("After concatenation of a column:\n")  print(C) |

**Output:**

The 3x3 matrix:

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

The 1x3 matrix:

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

[1, ] 10 11 12

Error in cbind(A, B) : number of rows of matrices must match (see arg 2)

#### Deleting rows and columns of a Matrix

To delete a row or a column, first of all, you need to access that row or column and then insert a negative sign before that row or column. It indicates that you had to delete that row or column.

**Row deletion:**

|  |
| --- |
| R program to illustrate row deletion in metrics.Create a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("Before deleting the 2nd row\n")  print(A)    # 2nd-row deletion  A = A[-2, ]    cat("After deleted the 2nd row\n")  print(A) |

**Output:**

Before deleting the 2nd row

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

After deleted the 2nd row

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

[1, ] 1 2 3

[2, ] 7 8 9

**Column deletion:**

|  |
| --- |
| # R program to illustrate column deletion in metricsCreate a 3x3 matrix  A = matrix(    c(1, 2, 3, 4, 5, 6, 7, 8, 9),    nrow = 3,    ncol = 3,    byrow = TRUE  )  cat("Before deleting the 2nd column\n")  print(A)    # 2nd-row deletion  A = A[, -2]    cat("After deleted the 2nd column\n")  print(A) |

**Output:**

Before deleting the 2nd column

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

[1, ] 1 2 3

[2, ] 4 5 6

[3, ] 7 8 9

After deleted the 2nd column

[, 1] [, 2]

[1, ] 1 3

[2, ] 4 6

[3, ] 7 9

### Example

# Create a matrix:  
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)  
  
# Print the matrix:  
thismatrix

**Note:** Remember the c() function is used to concatenate items together.

You can also create a matrix with strings:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
thismatrix

## Access Matrix Items:

You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
**thismatrix[1, 2]**

The whole row can be accessed if you specify a comma **after** the number in the bracket:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
**thismatrix[2,]**

The whole column can be accessed if you specify a comma **before** the number in the bracket:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
**thismatrix[,2]**

**Access More Than One Row:**

More than one row can be accessed if you use the c() function:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
thismatrix[c(1,2),]

## Access More Than One Column:

More than one column can be accessed if you use the c() function:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
thismatrix[, c(1,2)]

## Add Rows and Columns

Use the cbind() function to add additional columns in a Matrix:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
newmatrix <- cbind(thismatrix, c("strawberry", "blueberry", "raspberry"))  
  
# Print the new matrix  
newmatrix

**Note:** The cells in the new column must be of the same length as the existing matrix.

Use the rbind() function to add additional rows in a Matrix:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
newmatrix <- rbind(thismatrix, c("strawberry", "blueberry", "raspberry"))  
  
# Print the new matrix  
newmatrix

**Note:** The cells in the new row must be of the same length as the existing matrix.

## Remove Rows and Columns:

Use the c() function to remove rows and columns in a Matrix:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "mango", "pineapple"), nrow = 3, ncol =2)  
  
#Remove the first row and the first column  
thismatrix <- thismatrix[-c(1), -c(1)]  
  
thismatrix

## Check if an Item Exists:

To find out if a specified item is present in a matrix, use the %in% operator:

### Example

Check if "apple" is present in the matrix:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
"apple" %in% thismatrix

**Number of Rows and Columns:**

Use the dim() function to find the number of rows and columns in a Matrix:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
dim(thismatrix)

## Matrix Length:

Use the length() function to find the dimension of a Matrix:

### Example

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
length(thismatrix)

Total cells in the matrix is the number of rows multiplied by number of columns.

In the example above: Dimension = 2\*2 = **4**.

## Loop Through a Matrix

You can loop through a Matrix using a for loop. The loop will start at the first row, moving right:

### Example

Loop through the matrix items and print them:

thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
for (rows in 1:nrow(thismatrix)) {  
  for (columns in 1:ncol(thismatrix)) {  
    print(thismatrix[rows, columns])  
  }  
}

## Combine two Matrices

Again, you can use the rbind() or cbind() function to combine two or more matrices together:

### Example

# Combine matrices  
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol = 2)  
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2, ncol = 2)  
  
# Adding it as a rows  
Matrix\_Combined <- rbind(Matrix1, Matrix2)  
Matrix\_Combined  
  
# Adding it as a columns  
Matrix\_Combined <- cbind(Matrix1, Matrix2)  
Matrix\_Combined

## ARRAYS

Compared to matrices, arrays can have more than two dimensions.

We can use the array() function to create an array, and the dim parameter to specify the dimensions:

### Example

# An array with one dimension with values ranging from 1 to 24  
thisarray <- c(1:24)  
thisarray  
  
# An array with more than one dimension  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
multiarray

#### Example Explained

In the example above we create an array with the values 1 to 24.

How does dim=c(4,3,2) work?  
The first and second number in the bracket specifies the amount of rows and columns.  
The last number in the bracket specifies how many dimensions we want.

**Note:** Arrays can only have one data type.

## Access Array Items:

You can access the array elements by referring to the index position. You can use the [] brackets to access the desired elements from an array:

### Example

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
multiarray[2, 3, 2]

The syntax is as follow: array[row position, column position, matrix level]

You can also access the whole row or column from a matrix in an array, by using the c() function:

### Example

thisarray <- c(1:24)  
  
# Access all the items from the first row from matrix one  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
multiarray[c(1),,1]  
  
# Access all the items from the first column from matrix one  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
multiarray[,c(1),1]

A comma (,) before c() means that we want to access the column.

A comma (,) after c() means that we want to access the row.

## Check if an Item Exists:

To find out if a specified item is present in an array, use the %in% operator:

### Example

Check if the value "2" is present in the array:

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
2 %in% multiarray

**Amount of Rows and Columns**

Use the dim() function to find the amount of rows and columns in an array:

### Example

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
dim(multiarray)

## Array Length:

Use the length() function to find the dimension of an array:

### Example

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
length(multiarray)

## Loop Through an Array:

You can loop through the array items by using a for loop:

### Example

thisarray <- c(1:24)  
multiarray <- array(thisarray, dim = c(4, 3, 2))  
  
for(x in multiarray){  
  print(x)  
}

# Classes in R Programming

Classes and Objects are basic concepts of Object-Oriented Programming that revolve around the real-life entities. Everything in R is an object. An **object** is simply a data structure that has some methods and attributes. A **class** is just a blueprint or a sketch of these objects. It represents the set of properties or methods that are common to all objects of one type.

Unlike most other programming languages, R has a three-class system. These are S3, S4, and Reference Classes.

#### S3 Class:

S3 is the simplest yet the most popular OOP system and it lacks formal definition and structure. An object of this type can be created by just adding an attribute to it. Following is an example to make things more clear:

**Example:**

|  |
| --- |
| # create a list with required components  movieList <- list(name = "Iron man", leadActor = "Robert Downey Jr")    # give a name to your class  class(movieList) <- "movie"    movieList |

**Output:**

$name

[1] "Iron man"

$leadActor

[1] "Robert Downey Jr"

In S3 systems, methods don’t belong to the class. They belong to generic functions. It means that we can’t create our own methods here, as we do in other programming languages like C++ or Java. But we can define what a generic method (for example print) does when applied to our objects.

|  |
| --- |
| print(movieList) |

**Output:**

$name

[1] "Iron man"

$leadActor

[1] "Robert Downey Jr"

**Example: Creating a user-defined print function**

|  |
| --- |
| # now let us write our method  print.movie <- function(obj)  {      cat("The name of the movie is", obj$name,".\n")      cat(obj$leadActor, "is the lead actor.\n")  } |

**Output:**

The name of the movie is Iron man .

Robert Downey Jr is the lead actor.

#### S4 Class:

Programmers of other languages like C++, Java might find S3 to be very much different than their normal idea of classes as it lacks the structure that classes are supposed to provide. S4 is a slight improvement over S3 as its objects have a proper definition and it gives a proper structure to its objects.

**Example:**

|  |
| --- |
| library(methods)    # definition of S4 class  setClass("movies", slots=list(name="character", leadActor = "character"))    # creating an object using new() by passing class name and slot values  movieList <- new("movies", name="Iron man", leadActor = "Robert Downey Jr")  movieList |

**Output:**

An object of class "movies"

Slot "name":

[1] "Iron man"

Slot "leadActor":

[1] "Robert Downey Jr"

As shown in the above example, **setClass()** is used to define a class and **new()** is used to create the objects.

The concept of methods in S4 is similar to S3, i.e., they belong to generic functions. The following example shows how to create a method:

**Output:**

An object of class "movies"

Slot "name":

[1] "Iron man"

Slot "leadActor":

[1] "Robert Downey Jr"

**Example:**

|  |
| --- |
| # using setMethod to set a method  setMethod("show", "movies",  function(object)  {      cat("The name of the movie is ", object@name, ".\n")      cat(object@leadActor, "is the lead actor.\n")  }  )  movieList |

**Output:**

[1] "show"

The name of the movie is Iron man .

Robert Downey Jr is the lead actor.

#### Reference Class

Reference Class is an improvement over S4 Class. Here the methods belong to the classes. These are much similar to object-oriented classes of other languages.

Defining a Reference class is similar to defining S4 classes. We use **setRefClass()** instead of **setClass()** and “fields” instead of “slots”.

**Example:**

|  |
| --- |
| library(methods)  # setRefClass returns a generator  movies <- setRefClass("movies", fields = list(name = "character",                         leadActor = "character", rating = "numeric"))    #now we can use the generator to create objects  movieList <- movies(name = "Iron Man",                      leadActor = "Robert downey Jr", rating = 7)  movieList |

**Output:**

Reference class object of class "movies"

Field "name":

[1] "Iron Man"

Field "leadActor":

[1] "Robert downey Jr"

Field "rating":

[1] 7

Now let us see how to add some methods to our class with an example.

**Example**

|  |
| --- |
| library(methods)    movies <- setRefClass("movies", fields = list(name = "character",                         leadActor = "character", rating = "numeric"), methods = list(                         increment\_rating = function()                         {                             rating <<- rating + 1                         },                         decrement\_rating = function()                         {                             rating <<- rating - 1                         }                       ))  movieList <- movies(name = "Iron Man",                      leadActor = "Robert downey Jr", rating = 7)    # print the value of rating  movieList$rating    # increment and then print the rating  movieList$increment\_rating()  movieList$rating    # decrement and print the rating  movieList$decrement\_rating()  movieList$rating |

**Output:**

[1] 7

[1] 8

[1] 7

# R Objects and Classes

R is a functional language that uses concepts of objects and classes.

An object is simply a collection of data (variables) and methods (functions). Similarly, a class is a blueprint for that object.

Let's take a real life example,

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

## Class System in R

While most programming languages have a single class system, R has three class systems:

* [S3 Class](https://www.programiz.com/r/s3)
* [S4 Class](https://www.programiz.com/r/s4)
* Reference Class

**S3 Class in R**

S3 class is the most popular class in the R programming language. Most of the classes that come predefined in R are of this type.

First we create a list with various components then we create a class using the class() function. For example,

# create a list with required components

student1 <- list(name = "John", age = 21, GPA = 3.5)

# name the class appropriately

class(student1) <- "Student\_Info"

# create and call an object

student1

**Output**

$name

[1] "John"

$age

[1] 21

$GPA

[1] 3.5

attr(,"class")

[1] "student"

In the above example, we have created a list named student1 with three components. Notice the creation of class,

class(student1) <- "Student\_Info"

Here, Student\_Info is the name of the class. And to create an object of this class, we have passed the student1 list inside class().

Finally, we have created an object of the Student\_Info class and called the object student1.

## S4 Class in R

S4 class is an improvement over the S3 class. They have a formally defined structure which helps in making objects of the same class look more or less similar.

In R, we use the setClass() function to define a class. For example,

setClass("Student\_Info", slots=list(name="character", age="numeric", GPA="numeric"))

Here, we have created a class named Student\_Info with three slots (member variables): name, age, and GPA.

Now to create an object, we use the new() function. For example,

student1 <- new("Student\_Info", name = "John", age = 21, GPA = 3.5)

Here, inside new(), we have provided the name of the class "Student\_Info" and value for all three slots.

We have successfully created the object named student1.

## Example: S4 Class in R

# create a class "Student\_Info" with three member variables

setClass("Student\_Info", slots=list(name="character", age="numeric", GPA="numeric"))

# create an object of class

student1 <- new("Student\_Info", name = "John", age = 21, GPA = 3.5)

# call student1 object

student1

**Output**

An object of class "Student\_Info"

Slot "name":

[1] "John"

Slot "age":

[1] 21

Slot "GPA":

[1] 3.5

Here, we have created an S4 class named Student\_Info using the setClass() function and an object named student1 using the new() function.

## Reference Class in R:

Reference classes were introduced later, compared to the other two. It is more similar to the object oriented programming we are used to seeing in other major programming languages.

Defining a reference class is similar to defining a S4 class. Instead of setClass() we use the setRefClass() function. For example,

# create a class "Student\_Info" with three member variables

Student\_Info <- setRefClass("Student\_Info",

fields = list(name = "character", age = "numeric", GPA = "numeric"))

# Student\_Info() is our generator function which can be used to create new objects

student1 <- Student\_Info(name = "John", age = 21, GPA = 3.5)

# call student1 object

student1

**Output**

Reference class object of class "Student\_Info"

Field "name":

[1] "John"

Field "age":

[1] 21

Field "GPA":

[1] 3.5

In the above example, we have created a reference class named Student\_Info using the setRefClass() function.

And we have used our generator function Student\_Info() to create a new object student1.

## Comparison Between S3 vs S4 vs Reference Class

|  |  |  |
| --- | --- | --- |
| S3 Class | S4 Class | Reference Class |
| Lacks formal definition | Class defined using setClass() | Class defined using setRefClass() |
| Objects are created by setting the class attribute | Objects are created using new() | Objects are created using generator functions |
| Attributes are accessed using $ | Attributes are accessed using @ | Attributes are accessed using $ |
| Methods belong to generic function | Methods belong to generic function | Methods belong to the class |