

Data Analytics with R

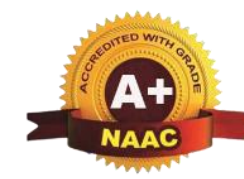
BDS306C

Prepared By,
Dr. Anitha DB
Associate Professor & Head
Department of CSE-Data Science
ATME College of Engineering, Mysuru



ATME

College of Engineering



Module1 : Basics of R

- Introducing R,
- Initiating R,
- Packages in R,
- Environments and Functions,
- Flow Controls, Loops,
- Basic Data Types in R,
- Vectors

1.1 Introducing R

- R is a Programming language
- R also refers to the software that is used to run the R program
- Ross Ihaka and Robert Gentleman from University of Auckland created R language in 1990s.
- R is a free open source software
- It is more advanced statistical programming language and it can produce outstanding graphical outputs.
- It is extremely flexible.

Features of R

- R is an interpreted language and not compiled one. This means that all commands typed on the keyboard are directly executed without need to build the complete program like C,C++ or Java.
- R's Syntax is very simple.

1.2 Installing R

R is available in several forms, essentially for Unix and Linux machines, or some precompiled binaries for windows, Linux and Macintosh.

The files needed to install R, either from the source or from the precompiled binaries are distributed from the internet site of the Comprehensive R Archive Network(CRAN) where the instructions for the installation are also available.

R can be installed from the link <http://www.r-project.org> using internet connection.

Use the “**Download R**” link in web page to download the R Executable.

Choose the version of R that is suitable for your operating system.

R scripts can run without the installation of IDE, the R-Studio using R-Console.

Once R installation completed then install R Studio.



1.2 Installing R

1: Install R

RStudio requires R 3.3.0+. Choose a version of R that matches your computer's operating system.

DOWNLOAD AND INSTALL R

2: Install RStudio

DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS

Size: 214.34 MB | [SHA-256: FE62B784](#) | Version:
2023.09.1+494 | Released: 2023-10-17

1.2 Installing R

R Studio is an Integrated Development Environment that consists of a GUI with **four parts**

1. A text editor,
2. Command line interpreter,
3. Place to display files, plots, packages and help information
4. Place to display the data being used and variables used in the program(Environment/History)

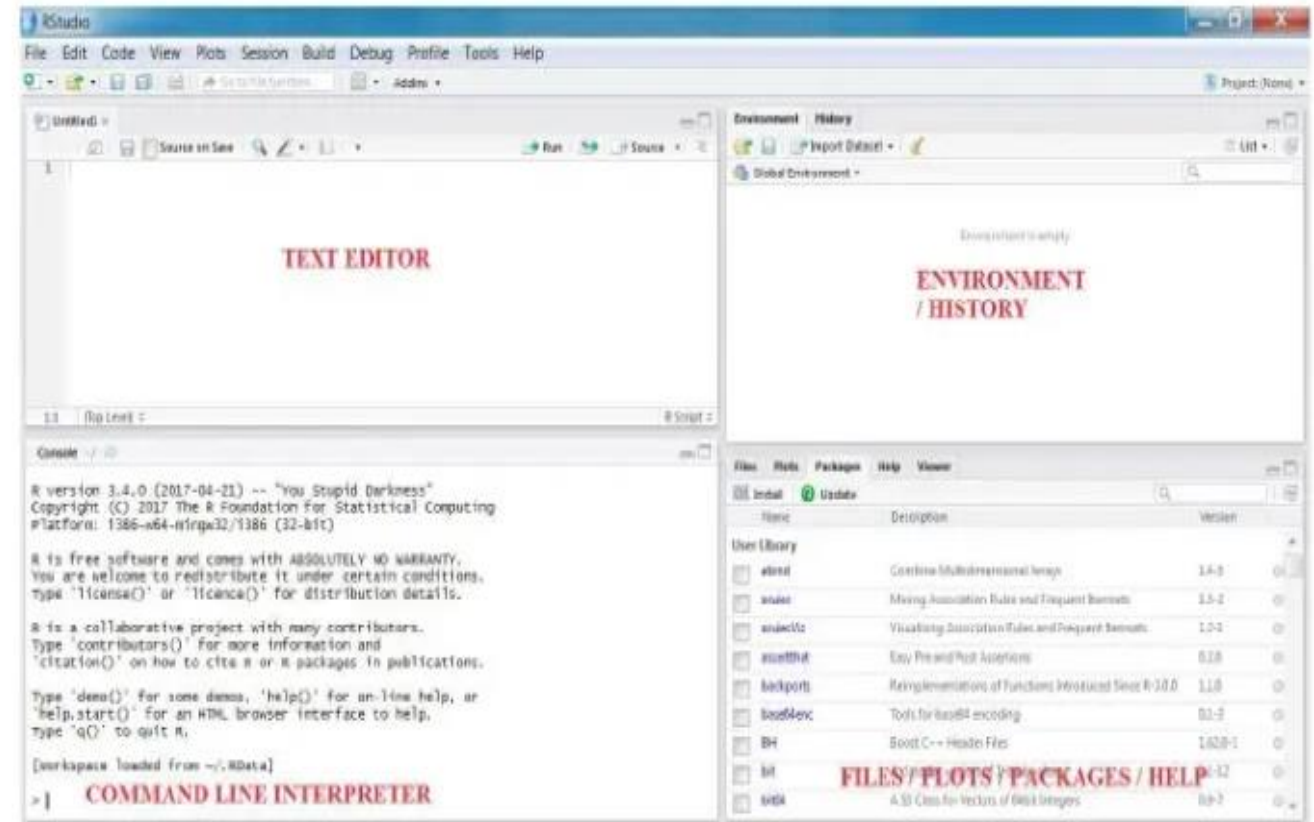


Figure 1.1 R-Studio GUI

1.3 Initiating R

1.3.1 First Program

Open R GUI, find the command prompt and type the command below and hit enter to run the command

```
>sum(1:5)
```

```
[1] 15
```

The result above shows that the command gives the result 15. That the command has taken the input of integers from 1 to 5 and has performed the sum operation on them.

In the above command `sum()` is a function that takes the argument `1:5` which means a vector that consists of sequence of integers from 1 to 5.

1.3 Initiating R

1.3.2 Help in R

- There are many ways to get help from R.
- If a function name or dataset name known then we can type ? followed by the name.
- If name is not known then we need to type ?? Followed by a term that is related to the search function.
- Keywords, special characters and two separate terms of search need to be enclosed in double or single quotes.
- The symbol # is used to comment a line in R program.

? name or **help()** or **help.search()**

```
> ?mean           # help page for mean function opens
> ?"+"            # help page for addition function opens
> ?"if"           # help page for if opens
> ??plotting      # searches for the help pages containing the word "plotting"
> ??"regression model" # searches for "regression model" phrase
```

```
> help("mean")
> help("+")
> help("if")
> help.search("plotting")
> help.search("regression model")
```

1.3 Initiating R

1.3.3 Assigning Variables

- The result of the operations in R can be stored for reuse.
- The values can be assigned to the variables using the symbol “< -” or “=” of which the symbol “<-” is preferred.
- There is no concept of variable declaration in R.
- The variable type is assumed based on the value assigned

Examples

```
> x<-1:3
```

```
> x
```

```
[1] 1 2 3
```

```
> Y=4:6
```

```
> Y
```

```
[1] 4 5 6
```

```
> x+3*Y-2
```

```
[1] 11 15 19
```

1.3 Initiating R

1.3.3 Assigning Variables

- The **variable names** consists of letters, numbers, dots and underscore, but a variable name should only starts with an alphabet.
- The variable name should not be reserve words.
- To create a global variable (Variable available everywhere) we use the symbol “<<-”.

Example

```
> x<<-exp(exp(1))  
> x  
[1] 15.15426
```

```
> assign("F",3*8)  
> F  
[1] 24
```

```
> assign("G",6*9,globalenv())  
> G  
[1] 54  
> print(G)  
[1] 54
```

- **Assignment operation** can also be done using assign() function.
- For global assignment the same function assign() can be used, but ,by including an extra attribute globalenv().
- To see the value of the variable, simply type the name of variable in the command prompt.
- The same thing can be done using a print() function

1.3 Initiating R

1.3.3 Assigning Variables

- If assignment and printing of a value has to be done in one line we can do the same in two ways.
- First method, by separating the two statements by a semicolon and
- The second method is by wrapping the assignment in parenthesis() as below.

```
> L<-sum(4:8);L  
[1] 30  
> (M<-sum(5:9))  
[1] 35
```

1.3 Initiating R

1.3.4 Basic Mathematical Operations

The “+” operator is used to perform the addition.

It can be used to add two numbers or add two vectors.

Vector represents an ordered set of values.

Vectors are mainly used to analyse statistical data.

The “:” colon operator creates a sequence .

Sequence is a series of numbers within the given limits.

The “c()” function concatenates the values given within the brackets “(“ and”)”.

Variable names in R are case sensitive.

Examples

```
> 7:12+12:17
```

```
[1] 19 21 23 25 27 29
```

```
> c(3,1,8,6,7)+c(9,2,5,7,1)
```

```
[1] 12 3 13 13 8
```

1.3 Initiating R

1.3.4 Basic Mathematical Operations

The vectors and c() function in R help us to **avoid loops**.

The statistical function in R can take the vectors as input and produce results.

The sum() function takes vector arguments and produces results.

Similar to the “+” operator all other operators in R take vectors as inputs and can produce the results.

The **subtraction and multiplication** operations work as below.

Examples for subtraction

```
> c(5,6,1,9)-2  
[1] 3 4 -1 7  
> c(5,6,1,9)-c(4,2,0,7)  
[1] 1 4 1 2
```

Examples for Multiplication

```
> -1.4*-2.3  
[1] 3.22  
> -1:4*3  
[1] -3 0 3 6 9 12
```

Examples

```
> sum(7:10)  
[1] 34  
> mean(7:10)  
[1] 8.5  
> median(7:10)  
[1] 8.5  
> sum(7,8,9,10)  
[1] 34
```

1.3 Initiating R

1.3.4 Basic Mathematical Operations

The **exponentiation** operator is represented using the symbol “^” or the “**”.

This can be checked using the function **identical()**

Examples

```
> identical(2^3,2**3)
[1] TRUE
```

The **division** operator is of **three types**.

1. The ordinary division is represented using the “/” symbol.
2. The integer division operator is represented using the “%/%” symbol.
3. The modulo division operator is represented using the “%%” symbol

Examples

```
> 5:9/2
[1] 2.5 3.0 3.5 4.0 4.5
> 5:9%/%2
[1] 2 3 3 4 4
> 5:9%%2
[1] 1 0 1 0 1
```

1.3 Initiating R

1.3.4 Basic Mathematical Operations

The **other mathematical function** are the trigonometry functions like `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()` and the logarithmic and exponential functions like `log()`, `exp()`, `loglp()`, `expml()`.

All these mathematical functions can operate on vectors as well as individual elements.

Comparison or relational operators

The operator “`=`” is used for comparing two values.

For checking inequalities of values the operator “`!=`” is used.

The other relations operators are the “`<`”, “`>`”, “`<=`”, “`>=`”.

The relational operator also take the vectors as input and operate on them.

Examples

```
> c(2,4-2,1+1)==2
```

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

```
> 1:5!=5:1
```

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

```
> exp(1:3)<20
```

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

```
> (1:10)^2>=50
```

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

1.3 Initiating R

- Non-integers cannot be compared using the operator "==" because it produces wrong results due to rounding off error of the floating numbers being compared.
- To overcome this issue we have the function `all.equal()`.

Examples

```
> sqrt(2)^2==2  
[1] FALSE
```

```
> sqrt(2)^2-2  
[1] 4.440892e-16
```

```
> all.equal(sqrt(2)^2,2)  
[1] TRUE
```

```
> all.equal(sqrt(2)^2,3)  
[1] "Mean relative difference: 0.5"
```

```
> isTRUE(all.equal(sqrt(2)^2,3))  
[1] FALSE
```

```
> c("Week","WEEK","week","weak")==="week"  
[1] FALSE FALSE TRUE FALSE
```

```
> c("A","B","C")<"B"  
[1] TRUE FALSE FALSE
```

```
> c("a","b","c")<"B"  
[1] TRUE TRUE FALSE
```

1.4 Packages in R

- R Packages are installed in an online repository called CRAN (Comprehensive R Archive Network).
- A Package is a collection of R functions and datasets.
- Currently, the CRAN package repository features 10756 available packages.
- The list of all available packages in the CRAN repository can be viewed from the web site ["https:// cran.r-project.org/web/packages/available_packages_by_name.html"](https://cran.r-project.org/web/packages/available_packages_by_name.html).
- To find the list of functions available in a package (say the package is "stats") we can use the command `ls("package:stats")` or the command `library(help = stats)` in the command prompt.

1.4 Packages in R

- A **library** is a folder in the machine that stores the files for a package.
- If a package is already installed on a machine we can load the same using the **library()** function.
- The name of the package to be loaded is passed to the library() function as argument without enclosing in quotes.
- If the package name has to be programmatically passed to the library() function, then we need to set the argument **character.only = TRUE**.
- If a package is not installed and if the library() function is used to load the package, it will throw an **error** message.
- Alternatively if the **require()** function is used to load a package, it returns **TRUE** if the package is already installed or it returns **FALSE** if the package is not already installed.

1.4 Packages in R

- We can list and see all the packages that are already loaded using the **search()** function.
- This list shows the global environment as the first one followed by the recently loaded packages.
- The last two are special environments, namely, "Autoloads" and "base" package.

```
> search()
[1] ".GlobalEnv"          "tools:rstudio"       "package:stats"
[4] "package:graphics"    "package:grDevices"   "package:utils"
[7] "package:datasets"    "package:methods"     "Autoloads"
[10] "package:base"
.
```

1.4 Packages in R

- The function **installed.packages()** returns a data frame with information about all the packages installed in a machine.
- It is safe to view the results of this using the **View()** function as it may list hundreds of packages.
- This list of packages also shows the version of the package installed, location on the machine and dependent packages.

Data: installed.packages()

	row.names	Package	LibPath	Version
1	base	base	C:/Program Files/R/R-4.3.1/library	4.3.1
2	boot	boot	C:/Program Files/R/R-4.3.1/library	1.3-28.1
3	class	class	C:/Program Files/R/R-4.3.1/library	7.3-22
4	cluster	cluster	C:/Program Files/R/R-4.3.1/library	2.1.4
5	codetools	codetools	C:/Program Files/R/R-4.3.1/library	0.2-19
6	compiler	compiler	C:/Program Files/R/R-4.3.1/library	4.3.1
7	datasets	datasets	C:/Program Files/R/R-4.3.1/library	4.3.1
8	foreign	foreign	C:/Program Files/R/R-4.3.1/library	0.8-84
9	graphics	graphics	C:/Program Files/R/R-4.3.1/library	4.3.1
10	grDevices	grDevices	C:/Program Files/R/R-4.3.1/library	4.3.1
11	grid	grid	C:/Program Files/R/R-4.3.1/library	4.3.1
12	KernSmooth	KernSmooth	C:/Program Files/R/R-4.3.1/library	2.23-21
13	lattice	lattice	C:/Program Files/R/R-4.3.1/library	0.21-8
14	MASS	MASS	C:/Program Files/R/R-4.3.1/library	7.3-60
15	Matrix	Matrix	C:/Program Files/R/R-4.3.1/library	1.5-4.1
16	methods	methods	C:/Program Files/R/R-4.3.1/library	4.3.1
17	mgcv	mgcv	C:/Program Files/R/R-4.3.1/library	1.8-42
18	nlme	nlme	C:/Program Files/R/R-4.3.1/library	3.1-162
19	nnet	nnet	C:/Program Files/R/R-4.3.1/library	7.3-19

1.4 Packages in R

The function **R.home("library")** retrieves the location on the machine that stores all R default packages.

The same result can be accomplished using the **.Library** command.

The home directory can be listed using the **path.expand("~/")** and **Sys.getenv("HOME")** functions.

```
> R.home("library")  
[1] "C:/PROGRA~1/R/R-43~1.2/library"
```

```
> .Library  
[1] "C:/PROGRA~1/R/R-43~1.2/library"
```

```
> path.expand("~/")  
[1] "C:/Users/ANITHA D B/Documents"  
> Sys.getenv("HOME")  
[1] "C:\\Users\\ANITHA D B\\Documents"
```

1.4 Packages in R

When R is upgraded, it is required to reinstall all the packages as different versions of R needs different versions of the packages.

The function **.libPaths()** lists all the R libraries in the installed machine.

The first value listed is the place where the packages will be installed by default.

```
> .libPaths()  
[1] "C:/Users/ANITHA D B/AppData/Local/R/win-library/4.3"  
[2] "C:/Program Files/R/R-4.3.2/library"
```

1.4 Packages in R

- The CRAN package repository contains handful of packages that needs special attention.
- To access additional repositories, type **setRepositories()** and select the repository required.
- The repositories R-Forge and rforge.net contains the development versions of the packages that appear on the CRAN repository.
- The function **available.packages()** lists thousands of packages in each of the selected repository. (Note: can use the View() function to restrict fetching of thousands of the packages at one go)

```
> setRepositories()
--- Please select repositories for use in this session ---

1: + CRAN
2:   BioC software
3:   BioC annotation
4:   BioC experiment
5:   CRAN (extras)
6:   R-Forge
7:   rforge.net

Enter one or more numbers separated by spaces and then ENTER, or 0 to cancel
1:
```

1.4 Packages in R

There are many online repositories like GitHub, Bitbucket, and Google Code from where many R Packages can be retrieved.

The packages can be installed using the function **install.packages()** function by mentioning the name of the package as argument to this function.

But, it is necessary to have internet connection to install any package and write permission to the hard drive.

To update the latest version of the installed packages, we use the function **update.packages()** with the argument `ask = FALSE` which disallows prompting before updating each package.

To delete a package already installed, we use the function **remove.packages()** by passing the name of the package to be removed as argument.

```
> install.packages("chron")
```

1.5 Environments and Functions

1.5.1. Environments

- In R the variables that we create need to be stored in an environment.
- Environments are another type of variables.
- We can assign them, manipulate them and pass them as arguments to functions.
- They are like lists that are used to store different types of variables.
- When a variable is assigned in the command prompt, it goes by default into the global environment.
- When a function is called, an environment is automatically created to store the function-related variables.
- A new environment is created using the function **new.env()**.

```
>newenvironment <-new.env()
```

1.5 Environments and Functions

1.5.1. Environments

- We can assign variables into a newly created environment using the double square brackets or the dollar operator as below.
- The assign function can also be used to assign variables to an environment.
- We can use get function to retrieve the values stored in environment

```
> newenvironment[["variable1"]]<-c(4,5,7)
> newenvironment$"variable2"<-TRUE
> assign("variable3", c(1:5),newenvironment)
> newenvironment[["variable1"]]
[1] 4 5 7
> newenvironment$variable2
[1] TRUE
> get("variable3",newenvironment)
[1] 1 2 3 4 5
```

Name	Type	Value
newenvironment	environment [3]	<environment: 0x000002122bc19d00>
variable1	double [3]	4 5 7
variable2	logical [1]	TRUE
variable3	integer [5]	1 2 3 4 5

1.5 Environments and Functions

1.5.1. Environments

The functions **ls()** and **ls.str()** take an environment argument and lists its contents.
We can test if a variable exists in an environment using the **exists()** function.

```
> ls.str(envir=newenvironment)
variable1 :  num [1:3] 4 5 7
variable2 :  logi TRUE
variable3 :  int [1:5] 1 2 3 4 5
> exists("variable3",newenvironment)
[1] TRUE
```

```
> newlist<-as.list(newenvironment)
> newlist
$variable3
[1] 1 2 3 4 5

$variable1
[1] 4 5 7

$variable2
[1] TRUE
```

An environment can be converted into a list using the function **as.list()** and a list can be converted into an environment using the function **as.environment()** or the function **list2env()**.

1.5 Environments and Functions

1.5.1. Environments

All environments are nested and so every environment has a parent environment. The empty environment sits at the top of the hierarchy without any parent. The `exists()` and the `get()` function also looks for the variables in the parent environment. To change this behaviour we need to pass the argument `inherits = FALSE`.

```
> subenv <- new.env(parent = newenvironment)
> exists("variable1", subenv)
[1] TRUE
> exists("variable1", subenv, inherits = FALSE)
[1] FALSE
```

The word **frame** is used interchangeably with the word **environment**. The function to refer to parent environment is denoted as `parent.frame()`. The variables assigned from the command prompt are stored in the global environment. The functions and the variables from the R's base package are stored in the base environment.

1.5 Environments and Functions

1.5.2. Functions

- A function and its environment together is called a **closure**.
- When we load a package, the functions in that package are stored in the environment on the search path where the package is installed.
- Functions are also another data types and hence we can assign and manipulate and pass them as arguments to other functions.
- **Typing the function name in the command prompt lists the code associated with the function.**
- Below is the code listed for the functions **readLines()** and **matrix()**.

```
> readLines
function (con = stdin(), n = -1L, ok = TRUE, warn = TRUE, encoding = "unknown",
  skipNul = FALSE)
{
  if (is.character(con)) {
    con <- file(con, "r")
    on.exit(close(con))
  }
  .Internal(readLines(con, n, ok, warn, encoding, skipNul))
}
<bytecode: 0x000002009f3be198>
<environment: namespace:base>
```

```
> matrix
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
{
  if (is.object(data) || !is.atomic(data))
    data <- as.vector(data)
  .Internal(matrix(data, nrow, ncol, byrow, dimnames, missing(nrow),
    missing(ncol)))
}
<bytecode: 0x000002009c8e7ea8>
<environment: namespace:base>
```

1.5 Environments and Functions

1.5.2. Functions

- When we call a function by passing values to it, the values are called as **arguments**.
- The lines of code of the function can be seen between the **curly braces** as **body** of the function.
- In R, there is no **explicit return statement** to return values.
- The last value that is calculated in a function is returned by default in R.

To create user defined functions

It is required to just assign the function as we do for other variables.

Example

```
> cube<-function(x)
+ {cu<-x^3}
> z<-cube(5)
> z
[1] 125
```

In this cube is the name of the function and x is the argument passed to this function. The content within the curly braces is the body of the function. (Note: If it is a one line code we can omit the curly braces). Once a function is defined, it can be called like any other function in R by passing its arguments.

1.5 Environments and Functions

1.5.2. Functions

The functions **formals()**, **args()** and **formalArgs()** can fetch the arguments defined for a function. The body of the function can be retrieved using the **body()** and **deparse()** functions.

```
> formals(cube)
$x

> args(cube)
function (x)
NULL
> formalArgs(cube)
[1] "x"
> body(cube)
{
  cu <- x^3
}
> deparse(cube)
[1] "function (x) " "{ " "cu <- x^3" "}"
```

1.5 Environments and Functions

1.5.2. Functions

Functions can be passed as arguments to other functions and they can be returned from other functions.

For calling a function, there is another function called **do.call()** in which we can pass the function name and its arguments as arguments.

The use of this function can be seen below when using the **rbind()** function to concatenate two data frames.

```
> f1<-data.frame(x=1:4,y=5:8)
> f2<-data.frame(x=9:12,y=13:16)
> do.call(rbind,list(f1,f2))
```

	x	y
1	1	5
2	2	6
3	3	7
4	4	8
5	9	13
6	10	14
7	11	15
8	12	16

```
> do.call(cbind,list(f1,f2))
```

	x	y	x	y
1	1	5	9	13
2	2	6	10	14
3	3	7	11	15
4	4	8	12	16

1.5 Environments and Functions

1.5.3. Variable Scope

- Variable's scope is the place where we can see the variable.
- If a variable is defined within a function, the variable can be accessed from any statement in the function.
- Also the sub-functions will have access to the variables defined in the parent function.

```
x<-function(a1)
{
  a2<-1
  y<-function(a1)
  {
    a2/a1
  }
  y(a1)
}

print(x(5))
```

1.5 Environments and Functions

1.5.3. Variable Scope

- Thus R will search for a variable in the current environment and if it could not find it, it will check the same in its parent environment. This search will proceed upwards until the variable is searched in the global environment. The variables defined in the global environment are called the global variables, which can be accessed from anywhere else.
- The replicate() function can be used to run a function several times as below. In this the user defined function random() returns 1 if the value returned by the rnorm() function is a positive value and otherwise it returns the value of the argument passed to the function random().
- This function random() is called 20 times using the replicate() function.

```
random<-function(x)
{
  if(rnorm(1)>0)
  { r<-1 }
  else
  { r<-x }}
replicate(20,random(5))
```

```
[1] 5 5 5 1 1 5 5 5 1 5 1 5 5 1 5 1 5 1 1 1
```

1.6 Flow Control

In Some situations it may be required to execute some code only if a condition is satisfied

1.6.1. If and Else Statement

The **if statement** takes a logical value and executes the next statement only if the value is TRUE

```
a <- 33
```

```
b <- 200
```

```
if (b > a) {  
  message("b is greater than a")  
}
```

```
b is greater than a
```

In this example we use two variables, **a** and **b**, which are used as a part of the if statement to test whether **b** is greater than **a**. As **a** is 33, and **b** is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

1.6 Flow Control

1.6.1. If and Else Statement

Else If: The **else if** keyword is R's way of saying "if the previous conditions were not true, then try this condition"

```
a <- 33
```

```
b <- 33
```

```
if (b > a) {  
  print("b is greater than a")  
} else if (a == b) {  
  print("a and b are equal")  
}
```

In this example **a** is equal to **b**, so the first condition is not true, but the **else if** condition is true, so we print to screen that "a and b are equal".

You can use as many **else if** statements as you want in R.

1.6 Flow Control

1.6.1. If and Else Statement

If Else: The **else** keyword catches anything which isn't caught by the preceding conditions:

```
a <- 200
```

```
b <- 33
```

```
if (b > a) {  
  print("b is greater than a")  
} else if (a == b) {  
  print("a and b are equal")  
} else {  
  print("a is greater than b")  
}
```

In this example, **a** is greater than **b**, so the first condition is not true, also the **else if** condition is not true, so we go to the **else** condition and print to screen that "a is greater than b".

1.6 Flow Control

1.6.1. If and Else Statement

```
a<-5
if(a<7)
{
  b<-a*3
  c<-b*3
  message("b is",b)
  message("c is",c)
}
```

```
b is15
c is45
```

```
a<-8
if(a<7)
{
  b<-a*5
  c<-b*3
  message("b is",b)
  message("c is",c)
}else
{
  message("a is greater than 7")
}
```

```
a is greater than 7
```

```
a<--8
if(a<0)
{
  message("a is negative")
}else if(a==0)
{
  message("a is 0")
}else if(a>0)
{
  message("a is positive")
}
```

```
a is negative
```

1.6 Flow Control

1.6.1. If and Else Statement

Ifelse() function takes three arguments of which the first is logical condition, the second is the value that is returned when the first vector is TRUE and third is the value that is returned when the first vector is FALSE.

```
a<-3  
b<-5  
ifelse(a<b,"a is less than b","a is greater than b")
```

1.6 Flow Control

1.6.2. Switch Statement

- If there are many else statements, it looks confusing and in such cases the **switch()** function is required.
- The first argument of the switch statement is an expression that can return a string value or an integer.
- This is followed by several named arguments that provide the results when the name matches the value of the first argument.
- Here also we can execute multiple statements enclosed by curly braces.
- If there is no match the switch statement returns NULL. So, in this case, it is safe to mention a default value if none matches.

```
switch("color","color"="red","shape"="circle","radius"=10)
switch("position","color"="red","shape"="circle","radius"=10)
switch("position","color"="red","shape"="circle","radius"=10,"default")
switch(2,"red","green","blue")
```

```
> switch("color","color"="red","shape"="circle","radius"=10)
[1] "red"
> switch("position","color"="red","shape"="circle","radius"=10)
> switch("position","color"="red","shape"="circle","radius"=10,"default")
[1] "default"
> switch(2,"red","green","blue")
[1] "green"
> switch(1,"red","green","blue")
[1] "red"
> switch(3,"red","green","blue")
[1] "blue"
```

1.7 Loops

There are three kinds of loops in R namely

- Repeat
- While
- For

1.7.1 Repeat Loops

The **repeat** is the easiest loop in R that executes the same code until it is forced to stop.

This **repeat** is similar to the **do while** statement in other languages.

A **break** statement can be given when it is required to break the looping.

Also it is possible to skip the rest of the statements in a loop and executes the next iteration and this is done by using the **next** statement.

```
a<-1
repeat{
  print(a)
  a<-a+1
  if(a==4){
    break }}

```

1.7 Loops

1.7.1 Repeat Loops

Repeat with break statement

```
a<-1
repeat{
  print(a)
  a<-a+1
  if(a==4){
    break
  }
}
```

```
[1] 1
[1] 2
[1] 3
```

Repeat with break and next statement

```
a<-0
repeat{
  a<-a+1
  if(a==4){
    next
  }
  print(a)
  if(a==6){
    break
  }
}
```

```
[1] 1
[1] 2
[1] 3
[1] 5
[1] 6
```

1.7 Loops

1.7.2 While Loops

The while loops are backward repeat loops.

The **repeat** loop executes the code and then checks for the condition, but in while loops the condition is first checked and then the code is executed.

So, in this case it is possible that the code may not be executed even once when the condition fails at the entry itself during the first iteration.

A **break** statement can be given when it is required to break the looping.

Also it is possible to skip the rest of the statements in a loop and executes the next iteration and this is done by using the **next** statement.

1.7 Loops

1.7.1 Repeat Loops

While without any break and next statement

```
a<-0
while(a<6){
  print(a)
  a<-a+1
}
```

```
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

while with next statement

```
a<-0
while(a<6){
  a<-a+1
  if(a==4){
    next
  }
  print(a)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 5
[1] 6
```

Next: With the next statement, we can skip an iteration without terminating the loop.

When the loop passes the value 4, it will skip it and continue to loop.

1.7 Loops

1.7.2 While Loops

While with break statement

```
a<-0
while(a<6){
a<-a+1
if(a==4){
break
}
print(a)
}
```

```
[1] 1
[1] 2
[1] 3
```

Break: With the break statement, we can stop the loop even if the while condition is TRUE
The loop will stop at 3 because we have chosen to finish the loop by using the break statement when a is equal to 4 (a == 4).

While with break and next statement

```
a<-0
while(1){
a<-a+1
if(a==4){
next
}
print(a)
if(a==8){
break
}
}
```

```
[1] 1
[1] 2
[1] 3
[1] 5
[1] 6
[1] 7
```

1.7 Loops

1.7.3 For Loops

- The **for** loops are used when we know how many times the code needs to be repeated.
- The **for** loop accepts an iterating variable and a vector.
- It repeats the loop giving the iterating each element from the vector in turn.
- In this case also if there are multiple statements to execute, we can use the curly braces.
- The iterating variable can be an integer, number, character or logical vectors and they can be even lists.

```
for(i in 1:5)
{
  j<-i*i
  message("The square value of ",i," is ",j)
}
```

```
The square value of 1 is 1
The square value of 2 is 4
The square value of 3 is 9
The square value of 4 is 16
The square value of 5 is 25
```

1.7 Loops

1.7.3 For Loops

```
for(i in c(TRUE,FALSE,NA))  
{  
  message("This statement is ",i)  
}
```

```
This statement is TRUE  
This statement is FALSE  
This statement is NA
```

```
a<-c(1,2,3)  
b<-c("a","b","c","d")  
d<-c(TRUE,FALSE)  
l<-list(a,b,d)  
for(i in l)  
{  
  message("The value of list is ",i)  
}
```

```
The value of list is 123  
The value of list is abcd  
The value of list is TRUEFALSE
```

Data Types in R

2.1 Basic data types in R

1. Numeric
2. Integer
3. Complex
4. Logical
5. Character

2.1.1 Numeric

- Decimal values are called numeric in R.
- It is the default computational data type.
- If we assign a decimal value to a variable x as follows, x will be of numeric type.
- If we assign an integer to a variable k, it is still being saved as numeric value.
- The fact that if k is an integer can be confirmed with the **is.integer()** function

```
> x=10.5  
> x  
[1] 10.5  
> class(x)  
[1] "numeric"  
> typeof(x)  
[1] "double"
```

```
> k=1  
> class(k)  
[1] "numeric"  
> is.integer(k)  
[1] FALSE
```

Data Types in R

2.1 Basic data types in R

2.1.2 Integer

In order to create an integer variable in R, the `as.integer()` function is involved as below

```
> y=as.integer(3)
> y
[1] 3
> class(y)
[1] "integer"
> is.integer(y)
[1] TRUE
```

```
> y=3L
> y
[1] 3
> class(y)
[1] "integer"
> is.integer(y)
[1] TRUE
```

Data Types in R

2.1 Basic data types in R

2.1.2 Integer

We can force a numeric value into an integer with the same `as.integer()` function as below.

```
> as.integer(3.14)
[1] 3
```

Similarly we can parse a string for a decimal value as below

```
> as.integer("5.27")
[1] 5
```

But, if a non decimal string is forced, it is an error and it returns NA.

```
> as.integer("abc")
[1] NA
Warning message:
NAs introduced by coercion
```

Data Types in R

2.1 Basic data types in R

2.1.2 Integer

We can force a numeric value into an integer with the same `as.integer()` function as below.

```
> as.integer(3.14)
[1] 3
```

Similarly we can parse a string for a decimal value as below

```
> as.integer("5.27")
[1] 5
```

But, if a non decimal string is forced, it is an error and it returns NA.

```
> as.integer("abc")
[1] NA
Warning message:
NAs introduced by coercion
```

```
> as.integer(TRUE)
[1] 1
> as.integer(FALSE)
[1] 0
```

The integer values of the logical values TRUE and FALSE are 1 and 0 respectively.

Data Types in R

2.1 Basic data types in R

2.1.3 Complex

A complex number is expressed as an imaginary value i

```
> z=3+4i
> z
[1] 3+4i
> class(z)
[1] "complex"
```

If we find the square root of -1, it gives an error. But if it is converted into a complex number and then square root is applied, it produces the necessary result as another complex number.

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> sqrt(as.complex(-1))
[1] 0+1i
```

Data Types in R

2.1 Basic data types in R

2.1.4 Logical

- When two variable are compared, the logical values are created.
- The logical operators are “&”(and), “|”(or) and “!”(negation/not).

```
> a=4;b=7  
> p=a>b  
> p  
[1] FALSE  
> class(p)  
[1] "logical"
```

```
> a=TRUE; b=FALSE  
> a & b  
[1] FALSE  
> a|b  
[1] TRUE  
> !a  
[1] FALSE
```

Data Types in R

2.1 Basic data types in R

2.1.5 Character

The Character object is used to represent string values in R.

Objects can be converted into character values using the **as.character()** function.

A **paste()** function can be used to **concatenate** two character values

```
> s=as.character("7.48")
> s
[1] "7.48"
> class(s)
[1] "character"
> fname= "Puneeth"
> lname="Rajkumar"
> paste(fname,lname)
[1] "Puneeth Rajkumar"
```

Data Types in R

2.1 Basic data types in R

2.1.5 Character

However, a readable string can be created using the **sprintf()** function and this is similar to the C Language syntax.

```
> sprintf("%s has %d rupees","Sundar",1000)
[1] "Sundar has 1000 rupees"
```

The **substr()** function can be used to extract a substring from a given string.

```
> substr("Twinkle Twinkle Little Star",start=9,stop=15)
[1] "Twinkle"
```

The **sub()** function is used to replace the first occurrence of a string with another string as below.

```
> sub("Twinkle","Wrinkle","Twinkle Twinkle Little Star")
[1] "Wrinkle Twinkle Little Star"
```

Data Types in R

2.2 Vectors

A sequence of data elements of the same basic type is called a Vector.

Members in a Vector are called as components or members.

The `vector()` function creates a vector of a specified type and length.

The result is a zero or FALSE or empty string.

```
> vector("numeric",3)
[1] 0 0 0
> vector("logical",5)
[1] FALSE FALSE FALSE FALSE FALSE
> vector("character",2)
[1] "" ""
```

The command also produces the same result as the above commands.

```
> numeric(3)
[1] 0 0 0
> logical(5)
[1] FALSE FALSE FALSE FALSE FALSE
> character(2)
[1] "" ""
```

Data Types in R

2.2 Vectors

The **seq()** function allows to generate sequences.

The function **seq.int()** also creates sequence from one number to another, but this function provides more options for splitting the sequence

```
> seq(1:5)
[1] 1 2 3 4 5
> seq.int(5,12)
[1] 5 6 7 8 9 10 11 12
> seq.int(10,5,-1.5)
[1] 10.0 8.5 7.0 5.5
```

The function **seq_len()** creates a sequence from 1 to the input value.

The function **seq_along()** creates a sequence from 1 to the length of the input.

```
> seq_len(7)
[1] 1 2 3 4 5 6 7
> p<-c(3,4,5,6)
> seq_along(p)
[1] 1 2 3 4
```

Data Types in R

2.2 Vectors

The function **length()** can be used to find the length of the vector, that is the number of elements in a vector. Using this function, it is possible to assign new length to a vector. If the vector length is extended NA(s) will be added to the end.