CHAPTER 1

BASICS OF R

❖ OBJECTIVES

On completion of this Chapter you will be able to:

- understand how R is different from other languages
- install R in your system
- write a beginners program using R
- get help in R
- assign variables in R
- know the basic mathematical operations in R
- understand various environments and scope of variables
- understand functions in R
- understand program flow control in R
- understand loops in R

1.1. Introducing R

R is a Programming Language and R also refers to the software that is used to run the R programs. Ross Ihaka and Robert Gentleman from University of Auckland created R language in 1990s. R language is based on the S language. S Language was developed at the Bell Laboratories in 1970s. S Language was developed by John Chambers. R Software is a GNU project free and open source software. R (Language and Software) is developed by the R Core Team. R has evolved over the past 3 to 4 decades as its history originated from 1970s.

One can write a new package in R if the existing package is not sufficient for the individual's use. R is a high-level scripting language which need not be compiled, but it is an interpreted language. R is an imperative language and still it supports object-oriented programming.

R is a free open source language that has cross platform compatibility. R is a most advanced statistical programming language and it can produce outstanding graphical outputs. R is extremely flexible and comprehensive even for the beginners. R easily relates to other programming languages such as C, C++, Java, Python, Hadoop, etc. R can handle huge data in flat files even in semi structured or in unstructured form.

The R language allows the user to program loops to successively analyze several data sets. It is also possible to combine in single program different statistical functions to perform more complex analyses. The R users may get benefitted from a large number of programs written and available on the internet. At first R can look very complex for a beginner or non-specialist. But, this is not actually true as the prominent feature of R is its flexibility. R displays the results of the analysis immediately and these results are stored in "objects" so that further analysis can be done on them. The user can also extract a part of the result which is of interest to him.

Looking at the features of R, some users may think that "I can't write programs using R". But, this is not the case for two reasons. First, R is an interpreted language and not a compiled one. This means that all commands typed on the keyboard are directly executed without need to build the complete program like in C, C++ or Java. Second, R's syntax is very simple and intuitive.

In R, a function is always written with parentheses, eg. *ls()*. If only the name of the function is typed, R displays the content of the function. In this book the functions are written with their names followed by parentheses to distinguish them from other objects. When R is running variables, data, functions, results, etc. are stored in the active memory of the computer in the form of *objects* which have a *name*. The user can do actions on these *objects* with *operators* and *functions*.

1.2. Installing R

R is available in several forms, essentially for Unix and Linux machines, or some pre-compiled binaries for Windows, Linux and Macintosh. The files needed to install R, either from the source or from the pre-compiled binaries are distributed from the internet site of the Comprehensive R Archive Network (CRAN) where the instructions for 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 the IDE, the R-Studio using the R-Console. The prerequisite for installing R-Studio is that one should have downloaded and installed any version of R. (Version 3.3.0 of R is used for installation and running scripts used in this book). Follow the instructions on the website to complete installation of R Console.

Once R installation is completed we install R-Studio. For installation of R-Studio in Windows operating system, we download the latest precompiled binary distribution from the CRAN website http://www.rstudio.org. (Version 3.4 of R-Studio is used for installation and running scripts used in this book). Start the installation and follow the steps required by the setup wizard. Once completed, launch RStudio IDE from Start à All Programs à Rstudio à RStudio.exe or from your custom installation directory. The default installation directory for RStudio IDE is "C:\Program Files\RStudio\bin\rstudio.exe.

R Studio is an Integrated Development Environment (IDE) 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 the variables used in the program (Environment / History).

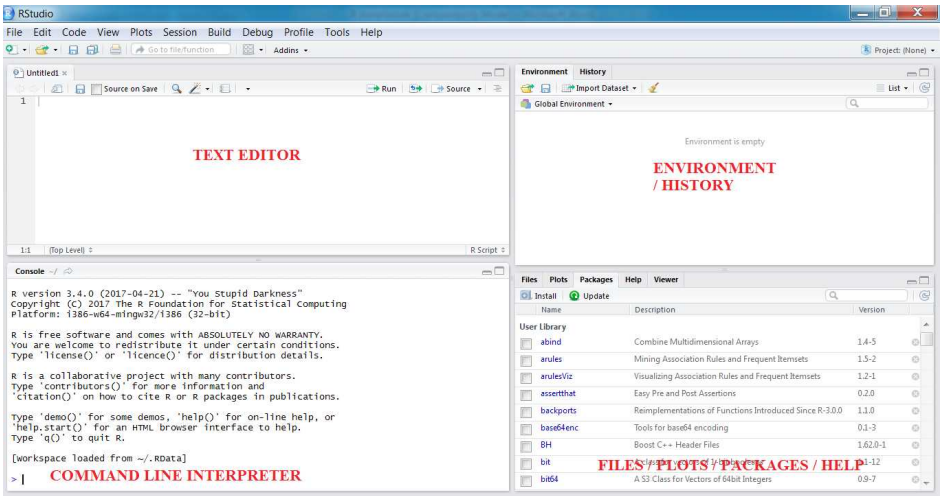

Figure 1.1 R-Studio GUI

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 is 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 a sequence of integers from 1 to 5. Like any other command prompt, R also allows to use the up arrow key to revoke the previous commands.

1.3.2. Help in R

There are many ways to get help from R. If a function name or a dataset name is 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 like any other programming language.

```
> ?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
```

The same help can be obtained by the functions *help()* and *help.search()*. In these functions the arguments has to be enclosed by quotes.

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

1.3.3. Assigning Variables

The results 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 variables declaration in R. The variable type is assumed based on the value assigned.

```
> X <- 1:3
> X
[1] 1 2 3
> Y = 4:6
> Y
[1] 4 5 6
> X + 3 * Y - 2
[1] 11 15 19
```

The variable names consist of letters, numbers, dots and underscores, but a variable name should only start with an alphabet. The variable names should not be reserve words. To create global variables (variables available everywhere) we use the symbol "<<-".

X <<- exp(exp(1))

Assignment operation can also be done using the *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 variable in the command prompt. The same thing can be done using a *print()* function.

```
> assign("F", 3 * 8)
> assign("G", 6 * 9, globalenv())
> F
[1] 24
> print(G)
[1] 54
```

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
```

1.3.4. Basic Mathematical Operations

The "+" plus operator is used to perform the addition operation. 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. Open R Gui, find the command prompt and type the command below and hit enter to run the command.

```
> 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
```

The vectors and the *c()* function in R helps us to avoid loops. The statistical functions in R can take the vectors as input and produce results. The *sum()* function takes vector arguments and produces results. But, the *median()* function when taking the vector arguments shows errors.

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

Similar to the "+" plus operator all other operators in R take vectors as inputs and can produce results. The subtraction and the multiplication operations work as below.

```
> 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
> -1:4 * -2:3
[1]  2  0  0  2  6 12
> -1:4 * 3
[1] -3  0  3  6  9 12
```

The exponentiation operator is represented using the symbol " ^ " or the "**". This can be checked using the function *identical()*.

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

The division operator is of three types. The ordinary division is represented using the "/" symbol, the integer division operator is represented using the "%/%" symbol and the modulo division operator is represented using the "%%" symbol. The below example commands show the results of the division operators.

```
> 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
```

The other *mathematical functions* are the trigonometry functions like, *sin()*, *cos()*, *tan()*, *asin()*, *acos()*, *atan()* and the logarithmic and exponential functions like *log()*, *exp()*, *log1p()*, *expm1()*. All these mathematical functions can operate on vectors as well as individual elements. Few more examples of the mathematical functions are listed below

The operator "==" is used for comparing two values. For checking inequalities of values the operator "!=" is used. These operators are called the *relational operators*. The relational operators also take the vectors as input and operate on them. The other relational operators are the "< ", "> ", "<= " and ">= ".

```
> 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
```

Non-integers cannot be compared using the operator "==" as it produces wrong results due to rounding off error of the float numbers being compared. For overcoming this issue we have the function *all.equal()*. If the value to be compared by the function *all.equal()* is not equal, it returns a report on the difference. To get a TRUE or FALSE reply, the *all.equal()* function has to be wrapped using the function *isTRUE()*. The below example code will help to understand the concepts discussed.

```
> 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
```

The equality operator "==" can also be used to compare strings, but, string comparison is case sensitive. Similarly, the operators "<" and ">" can also be used on strings. The below examples show the results.

```
> 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*". 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.

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.

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"        "package:cluster"   "tools:rstudio"     "package:stats"
[5] "package:graphics"  "package:grDevices" "package:utils"     "package:datasets"
[9] "package:methods"   "Autoloads"         "package:base"
```

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.

```
> View(installed.packages())
```

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-33~1.0/library"
> .Library
[1] "C:/PROGRA~1/R/R-33~1.0/library"
> path.expand("~")
[1] "C:/Users/admin/Documents"
> Sys.getenv("HOME")
[1] "C:/Users/admin/Documents"
```

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/admin/Documents/R/win-library/3.3"
[2] "C:/Program Files/R/R-3.3.0/library"
```

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:   BioC extra
 6:   CRAN (extras)
 7:   Omegahat
 8:   R-Forge
 9:   rforge.net
10: + CRANextra
Enter one or more numbers separated by spaces, or an empty line to cancel
1:
```

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()
```

We can assign variables into a newly created environment using the *double square brackets* or the *dollar operator* as below.

```
> newenvironment[["variable1"]] <- c(4, 7, 9)
> newenvironment$variable2 <- TRUE
> assign("variable3", "Value for variable3", newenvironment)
```

The *assign()* function can also be used to assign variables to an environment. Retrieving values stored in an environment is like list indexing or we can use the *get()* function.

```
> newenvironment[["variable1"]]
[1] 4 7 9
> newenvironment$variable2
[1] TRUE
> get("variable3", newenvironment)
[1] "Value for variable3"
```

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(envir = newenvironment)
[1] "variable1" "variable2" "variable3"
> ls.str(envir = newenvironment)
variable1 :  num [1:3] 4 7 9
variable2 :  logi TRUE
variable3 :  chr "Value for variable3"
> exists("variable2", newenvironment)
[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()*.

```
> newlist <- as.list(newenvironment)
> newlist
$variable3
[1] "Value for variable3"
$variable1
[1] 4 7 9
$variable2
[1] TRUE
> as.environment(newlist)
<environment: 0x124730a8>
> list2env(newlist)
<environment: 0x12edf3e8>
> anotherenv <- as.environment(newlist)
> anotherenv[["variable3"]]
[1] "Value for variable3"
```

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.

1.5.2. 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. A function in R is a verb and not a noun as it does things with its data. 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 function *readLines()*.

```
> 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))
}
```

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. The below code is an example of how to create a user defined function. 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.

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

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" "}"
```

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
```

When using functions as arguments to the *do.call()* function, it is not necessary to assign them first. We can pass a function anonymously as below.

```
> do.call(function(x,y) x * y, list(1:3, 4:6))
[1]  4 10 18
```

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)
}
> x(5)
[1] 0.2
```

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 1 1 5 1 5 5 5 5 5 5 5 5 5 1 1 5 1 5
```

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.

```
> if(TRUE) message("TRUE Statement")
TRUE Statement
> if(FALSE) message("FALSE Statement")
```

It is not necessary to pass the logical values as *TRUE* or *FALSE* directly, instead a variable or expression that returns a logical value can be used. If there are several statements to execute after the condition, they can be wrapped in curly braces.

```
a <- 5
if(a < 7)
{
   b <- a * 5
   c <- b * 3
   message("b is ", b)
   message("c is ", c)
}
b is 25
c is 75
```

In the *if* and *else* construct the code that follows the *if* statement is executed if the condition is TRUE and the code that follows the *else* statement is executed if the condition is FALSE. It is important to note that the *else* statement must occur on the same line as the closing curly brace of the *if* statement and otherwise it will throw an error message.

```
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
```

The *if* and *else* statements can be used repeatedly to code multiple conditions and this respective actions. In this case it is important to note that the *if* and the *else* statements are separated and they are not one word as *ifelse*. The *ifelse* function is of different use which will be covered shortly.

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

a is negative
```

The *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 the 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] "a is less than b"
```

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

1.7. Loops

There are three kinds of loops in R namely, *repeat*, *while* and *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 execute the next iteration and this is done using the *next* statement.

```
a <- 1
repeat {
 message("Inside the loop")
 if(a == 3)
 {
    a = a + 1
    next
 }
 message("The value of a is ", a)
 a = a + 1
 if(a > 5)
 {
    message("Exiting the loop")
    break
 }
}
Inside the loop
The value of a is 1
Inside the loop
The value of a is 2
Inside the loop
Inside the loop
```

```
The value of a is 4
Inside the loop
The value of a is 5
Exiting the loop
```

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. The same example above can be written using the while statement.

```
a <- 1
while (a <= 5)
{
   message("Inside the loop")
   if(a == 3)
  {
     a = a + 1
     next
  }
  message("The value of a is ", a)
  a = a + 1
}
Inside the loop
The value of a is 1
Inside the loop
The value of a is 2
Inside the loop
Inside the loop
```

```
The value of a is 4
Inside the loop
The value of a is 5
```

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
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 the list is ", i)
  }
```

The value of the list is 123

The value of the list is abcd

The value of the list is TRUEFALSE

❖ HIGHLIGHTS

- R is a free open source language that has cross platform compatibility.
- R's syntax is very simple and intuitive.
- R's installation software can be downloaded from the *CRAN* Website.
- Help in R can be obtained by using, for eg. *?mean() / help("mean")*
- Variables can be assigned using the symbol ß or the *assign()* function.
- The basic functions are *c()*, *sum()*, *mean()*, *median()*, *exp()*, *sqrt()* etc.
- The basic operators are "+", "*", ":", "/", "**", "^", "%%", "%/%", "==", "!=", "<", ">", "<=", ">=" etc.
- Currently, the *CRAN* package repository features *10756* available packages.
- A Package can be newly installed using the function *install.packages()* and it can be invoked using the function *library()*.
- When a variable is assigned in the command prompt, it goes by default into the global environment.
- To create a new environment we use the function *new.env()*.
- Typing the function name in the command prompt lists the code associated with the function.
- The *if* and the *else* statements are separated and they are not one word as *ifelse*.
- The *ifelse()* function takes three arguments.
- If there are many else statements, the *switch()* function is required.

- The function *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.
- To skip the rest of the statements in a loop we use the *next* statement.
- The *while* loops are backward *repeat* loops.
- The *for* loops are used when we know how many times the code needs to be repeated.

CHAPTER 2

DATA TYPES IN R

❖ OBJECTIVES

On completion of this Chapter you will be able to:

- know the basic data types in R of which the other complex data types are made of
- know how to create, access and perform basic operations on the vector data types in R
- know how to create, access and perform basic operations on matrices and arrays in R
- know how to create, access and perform basic operations on the list data types
- know how to create, access and perform basic operations on the factor data types in R
- know how to create, access and perform basic operations on strings in R
- understand the various date and time classes in R
- convert between various date formats
- setup various time zones
- perform calculations on dates and times

2.1. Basic Data Types in R

In contrast to other programming languages like C and Java, in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-objects becomes the data type of the variables. There are many

types of R-objects. The frequently used ones are – Vectors, Arrays, Matrices, Lists, Data Frames, Strings and Factors.

The simplest of these objects is the Vector object and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors. Hence, the basic data types in R are Numeric, Integer, Complex, Logical and 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.

```
> x = 10.5
> x
[1] 10.5
> class(x)      # print the class name of x
[1] "numeric"
```

Further more, even if we assign an integer to a variable *k*, it is still being saved as a numeric value. The fact that if *k* is an integer can be confirmed with the *is.integer()* function.

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

2.1.2. Integer

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

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

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")        # force a decimal string
[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
```

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

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

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
```

2.1.4. Logical

When two variables are compared, the logical values are created. The logical operators are "&" (and), "|" (or), and "!" (negation).

```
> 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
```

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 = "Adam"
> lname = "Smith"
> paste(fname, lname)
[1] "Adam Smith"
```

However, a readable string can be created using the *sprint()* 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. The *sub()* function is used to replace the first occurrence of a string with another string as below.

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

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 below commands 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] "" ""
```

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
```

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.

```
> length(1:7)
[1] 7
> length(c("aa", "ccc", "eeee"))
[1] 3
> nchar(c("aa", "ccc", "eeee"))
[1] 2 3 4
> s <- c(1,2,3,4,5)
> length(s) <- 3
> s
[1] 1 2 3
> length(s) <- 8
> s
[1]  1  2  3 NA NA NA NA NA
```

Each element of a vector can be given a name during the vector creation itself. If there are space or special characters in the name, it needs to be enclosed in quotes. The *names()* function can be used to give names to the vector elements after its creation.

```
> c(a = 1, b = 2, c = 3)
a b c
1 2 3
> s <- 1:3
> s
[1] 1 2 3
> names(s) <- c("a", "b", "c")
> s
a b c
1 2 3
```

Elements of a vector can be accessed using its indexes which are specified in a square bracket. The index number starts from 1 and not 0. Specifying a negative number as index to a vector means, it returns all the elements except the one specified. The name of the vector element can also be specified as index to fetch it.

```
> x <- c(1:5)
> x
[1] 1 2 3 4 5
> x[c(2,3)]
[1] 2 3
> x[c(-1,-4)]
[1] 2 3 5
> s <- 1:3
> s
[1] 1 2 3
> names(s) <- c("a", "b", "c")
> s["b"]
b
2
```

If an incorrect index is specified to access a vector element, the result is *NA*. Non integer indices are rounded off. Not passing any index to a vector will return all the elements of the vector.

```
> x
[1] 1 2 3 4 5
> x[7]
[1] NA
```

The *which()* function returns the elements of the vector which satisfies the condition specified within this function. The functions *which.min()* and *which.max()* can be used to display the minimum and the maximum elements in the vector.

```
> x
[1] 1 2 3 4 5
> which.min(x)
[1] 1
> which.max(x)
[1] 5
> which(x>3)
[1] 4 5
```

Vectors can be combined using the *c()* function. When the two vectors are combined the numeric values are forced into character values. This shows that all the members of a vector should be of the same basic data type.

```
> f = c(7, 5, 9)
> g = c("aaa", "bbb", "ccc")
> c(f, g)
[1] "7"   "5"   "9"   "aaa" "bbb" "ccc"
```

Arithmetic operations in a vector will be performed member-wise. If two vectors are of unequal length, the shorter vector will be recycled in order to match the longer vector.

```
> x = c(5, 8, 9)
> y = c(2, 6, 9)
> 4 * y
[1]  8 24 36
> x + y
[1]  7 14 18
> x - y
[1] 3 2 0
> x * y
[1] 10 48 81
> x / y
[1] 2.500000 1.333333 1.000000
> v = c(1, 2, 3, 4, 5, 6)
> x + v
[1]  6 10 12  9 13 15
```

The *rep()* function creates a vector with repeated elements. This function has its other variants such as *rep.int()* and *rep_len()* whose usage is as given below.

```
> rep(1:3, 4)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
> rep(1:3, each = 4)
 [1] 1 1 1 1 2 2 2 2 3 3 3 3
> rep(1:3, times = 1:3)
[1] 1 2 2 3 3 3
> rep(1:3, length.out = 9)
[1] 1 2 3 1 2 3 1 2 3
> rep.int(1:3, 4)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

```
> rep_len(1:3, 9)
[1] 1 2 3 1 2 3 1 2 3
```

2.3. Matrices and Arrays

A matrix is a collection of data elements with the same basic type arranged in a two-dimensional rectangular layout. An array consists of multidimensional rectangular data. Matrices are special cases of two-dimensional arrays. To create an array the *array()* function can be used and a vector of values and vector of dimensions are passed to it.

```
> x <- array(1:24, dim = c(4, 3, 2),
            dimnames = list(c("a", "b", "c", "d"), c("e", "f", "g"), c("h", "i")))
> x
, , h
  e f  g
a 1 5  9
b 2 6 10
c 3 7 11
d 4 8 12
, , i
   e  f  g
a 13 17 21
b 14 18 22
c 15 19 23
d 16 20 24
```

The syntax for creating matrices is using the function *matrix()* and passing the *nrow* or *ncol* argument instead of the *dim* argument in the arrays. A matrix can also be created using the *array()* function where the dimension of the array is two.

```
> m <- matrix(1:12, nrow = 3, dimnames = list(c("a", "b", "c"), c("d", "e", "f", "g")))
> m
```