[R Fundamentals]

2.0 Introduction

This Unit highlights the fundamentals of the R programming language. It starts with the R syntax, discusses about variables, provides an in-depth insight on the R data structures, identifies the common control structures and ends with an overview of functions.

Study skills	• *To better grasp the concepts, copy and paste the contents of the examples into an R Script.* • *Run it and observe its output.* • *Make modifications to the source code and observe its output again.*

2.1 Learning Outcomes

Outcomes

Upon completion of this unit, you will be able to:

- Use the console window and the script editor.
- Have an overview of the arithmetic, relational and logical operators.
- Work with variables.
- Examine the different data structures that exist in R.
- Familiarise yourself with the two main control structures: decisions and loops.
- Work with in-built and user-defined functions.

2.2 R Syntax

The basic syntax of R will be illustrated by writing a “Hello World!” program, a program that displays “Hello World” to the user.

There are two main ways of interacting with R:

- Using the console
- Using R scripts (plain text files that contain your code)

2.2.1. Using the Console

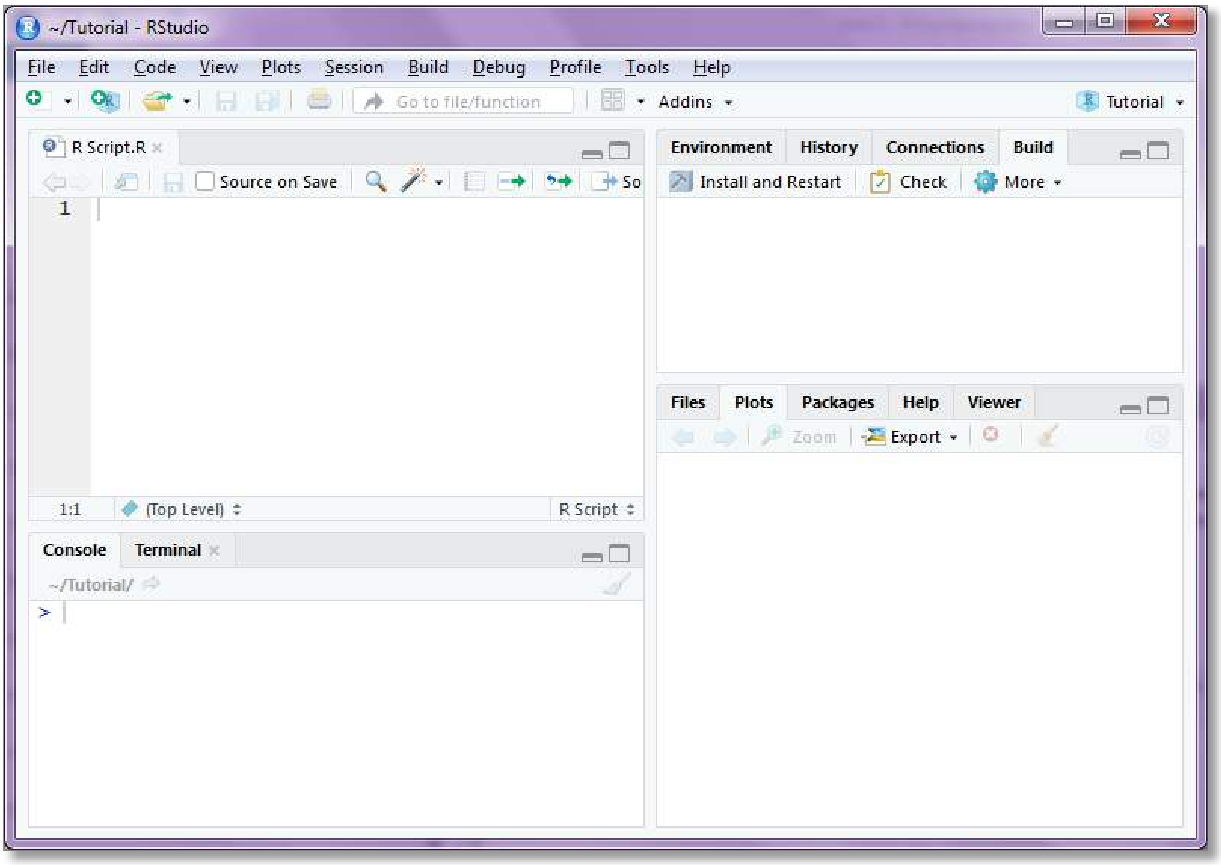
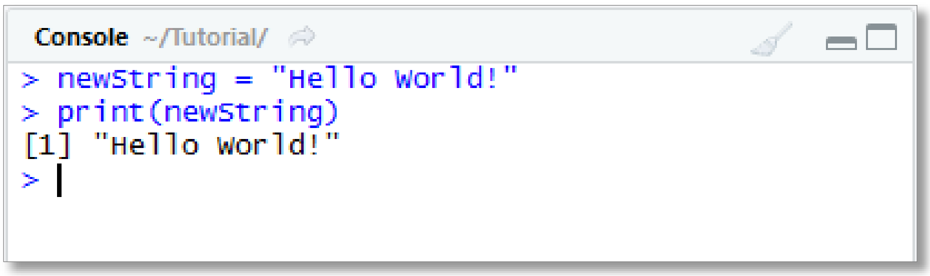


Figure 2.1 – R Studio

Figure 2.1 shows the main interface of RStudio and the Console window is located in the bottom left panel. Commands can be typed directly into the Console. Figure 2.2 shows the “Hello World” application written in the Console window.

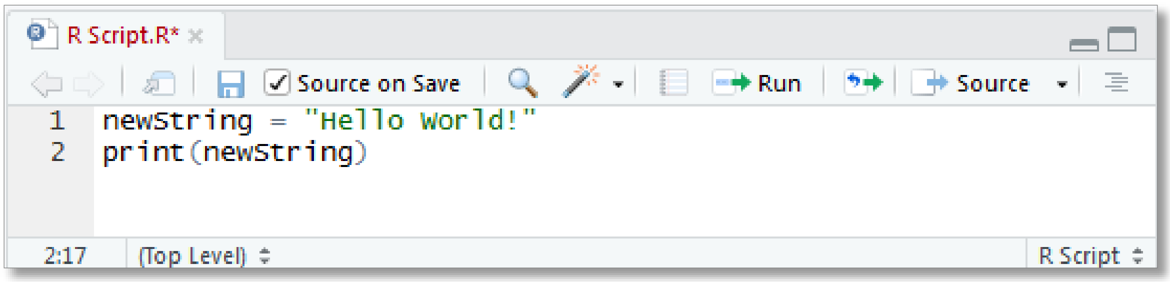


```
> newString = "Hello World!"
> print(newString)
[1] "Hello World!"
> 
```

Figure 2.2 – Hello World in Console

2.2.2. Using R Scripts

Commands written in the Console are forgotten once the session is closed. Therefore, to have a complete record of all the commands, it is best to enter the commands in the script editor. The R Script editor window is located in the top left panel of RStudio and is shown in Figure 2.3.



```
newString = "Hello World!"
print(newString)
```

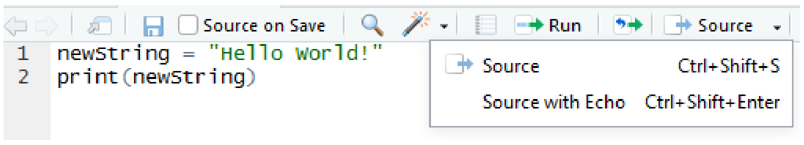
Figure 2.3 – The R Script Editor Window

The R Script Editor window contains the following options for running the commands:

- Run: Executes the selected line or lines of code
- Re-Run: Re-runs the previous code region
- Source: Executes the entire active document
- Source with Echo: Automatically prints (echo) all expressions inside the sourced script

Note it!

- *To run an entire script, the **Source** or the **Source with Echo** should be used.*



```
newString = "Hello World!"
print(newString)
```

2.2.3. R Comments

Comments are ignored by the interpreter but are inserted in programs to document the code for future maintenance. A # in the beginning of a line denotes a comment. R currently does not support multi-line comments.

In the following code listing, the first line is a comment and is ignored by the interpreter.

Example

```
# This program displays Hello World
newString = "Hello World!"
print(newString)
```

2.3. R Operators

Like other programming languages, R has a number of operators to perform arithmetic, logical and bitwise operations.

2.3.1. Arithmetic Operators

These operators are used to carry out mathematical operations such as addition and multiplication. The Table 2.1 below shows a list of arithmetic operators available in R.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or **	Exponentiation. E.g. 2^3 is 8
x %% y	Modulus - x mod y. E.g. 5%%2 is 1
x %/% y	Integer division. E.g. 5%/%2 is 2

Table 2.1 – Arithmetic Operators

2.3.2. Relational Operators

Relational operators are used to compare between values. The Table 2.2 below shows a list of relational operators available in R.

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Table 2.2 – Relational Operators

2.3.3. Logical Operators

Logical operators are used to carry out Boolean operations like AND and OR. The Table 2.3 below shows a list of logical operators available in R.

Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR
Operator	Description

Table 2.3 – Control Structures

2.4. Variables

A variable is a reserved memory location to store values and therefore provides a named storage that programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R-Objects. A valid variable name consists of letters, numbers and the dot or underline characters but cannot start with a number or an underscore. The variable name starts with a letter or the dot not followed by a number. The following shows examples of valid and invalid variable names.

Example

```
Valid variable names: var, var1, var.1, var_1, .var1

Invalid variable names:2var, .2var, _var2, var2%
```

2.4.1. Variable Assignment

Unlike other programming languages, variables in R can be assigned values using leftward, rightward and equal to operators. The *print()* or *cat()* functions can be used to print the values of the variables. The *cat()* function is used to print multiple items as a continuous print output.

The following code listing shows the different assignment operators.

Example

```
# Assignment using equal operator.
var.1 = 22

# Assignment using leftward operator.
var.2 <- "Learning R is fantastic"

# Assignment using rightward operator.
TRUE -> 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")
cat("var.1 is", var.1, "var.2 is ", var.2 ,"var.3 is ", var.3)
```

The above code produces the following output.

Program Output

```
[1] 22
var.1 is  22
var.2 is  Learning R is fantastic
var.3 is  TRUE
var.1 is 22 , var.2 is  Learning R is fantastic and var.3 is  TRUE
```

Activity

- *Copy the above example and paste the contents into an R Script.*
- *Run it by clicking on the Source or Source with Echo button.*
- *Assign different values to `var.1`, `var.2` and `var.3` and run the script again.*

Tip

- *Save All your activities as a script with an appropriate numbering.*

  *E.g. Save the above script as **Activity 2_4_1***

2.4.2. Variable Data Type

In many other programming languages such as C and Java, variables are used to store information of various data types such integer, floating point, double, string, Boolean etc… Unlike these programming languages, the variables in R are not declared with some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. So R is called a dynamically typed language where the data type of a variable can change again and again in a program.

In the following code listing, the variable `var_x` takes a number of successive values. The *class()* function is used to identify the class of `var_x` at those different points in time.

Example

```
var_x <- 22.5
cat("Initially the class of var_x is",class(var_x),"\n\n")

var_x <- "Hello World!"
cat("Now the class of var_x changes to",class(var_x),"\n\n")

var_x <- 45L
cat("Next the class of var_x becomes",class(var_x),"\n\n")
```

The above code produces the following output.

Program Output

```
Initially the class of var_x is numeric

Now the class of var_x changes to character

Next the class of var_x becomes integer
```

2.4.3. Locating Variables and Objects

The *ls()* and *objects()* functions return a vector of character strings with the names of the objects, including variables, in the specified environment.

Example

```
print(ls())

print(objects())
```

The above code produces the following output. This is a sample output and will depend on the variables declared in the environment.

Program Output

```
[1] "age2"    "agevar"  "ageVar1" "var.1"   "var.2"   "var.3"   "var_x"

[1] "age2"    "agevar"  "ageVar1" "var.1"   "var.2"   "var.3"   "var_x"
```

The *pattern* argument can be used to limit the results to only include names that match the specified pattern. In the following example, names of object that contains "var" will be displayed.

Example

```
print(ls(pattern="var"))
```

The above code produces the following output. This is a sample output and will depend on the variables declared in the environment.

Program Output

```
[1] "agevar" "var.1"  "var.2"  "var.3"  "var_x"
```

The variables with names starting with dot(.) are hidden. To list all the names including those starting with dot(.), the `"all.names = TRUE"` argument has to be passed to the *ls()* function.

Example

```
print(ls(all.names=TRUE))
```

The above code produces the following output. This is a sample output and will depend on the variables declared in the environment.

Program Output

```
[1] ".Random.seed" ".var1"        "age2"         "agevar"       "ageVar1"
[6] "var.1"        "var.2"        "var.3"        "var_x"
```

2.4.4. Removing Variables and Objects

The *rm()* function can be used to remove objects from the environment. All the objects to be removed can be specified successively as character strings, or in a vector list, or through a combination of both.

Example

```
print(ls())

rm (ageVar1, age2)

print(ls())

print(ageVar1)
```

The above code produces the following output. This is a sample output and will depend on the variables declared in the environment.

Program Output

```
[1] "age2"    "agevar"  "ageVar1" "var.1"   "var.2"   "var.3"   "var_x"

[1] "agevar" "var.1"  "var.2"  "var.3"  "var_x"

Error in print(ageVar1) : object 'ageVar1' not found
```

To delete all the variables in the environment, the *rm()* function has to be used in conjunction with the *ls()* function.

Example

```
print(ls())

rm(list = ls())

print(ls())

print(agevar)
```

The above code produces the following output. This is a sample output and will depend on the variables declared in the environment. The output indicates that all the objects have been removed

Program Output

```
[1] "agevar" "var.1"  "var.2"  "var.3"  "var_x"

character(0)

Error in print(agevar) : object 'agevar' not found
```

- *Create a new R Script.*

Activity	
Activity	• *Print the names of the objects, including variables, in your environment. Hint: See example 2.4.3.* • *Now delete all the variables in the environment.* • *In the same script, run the example 2.4.1 and then 2.4.2* • *Now, again print the names of the objects, including variables, in your environment.* • *You should have the following as output:* `[1] "var.1" "var.2" "var.3" "var_x"` • *Save the above script as **Activity 2_4_4***

2.4.5. Classes of Objects

An R object is anything that can be assigned to a variable including constants, data structures, functions, and even graphs. R has six basic or "atomic" classes of objects:

- Character
- Numeric (real numbers)
- Integer
- Complex
- Logical (True/False)
- Raw

Therefore, a vector, the simplest data structure, can store data of the above types as shown in Figure 2.4.

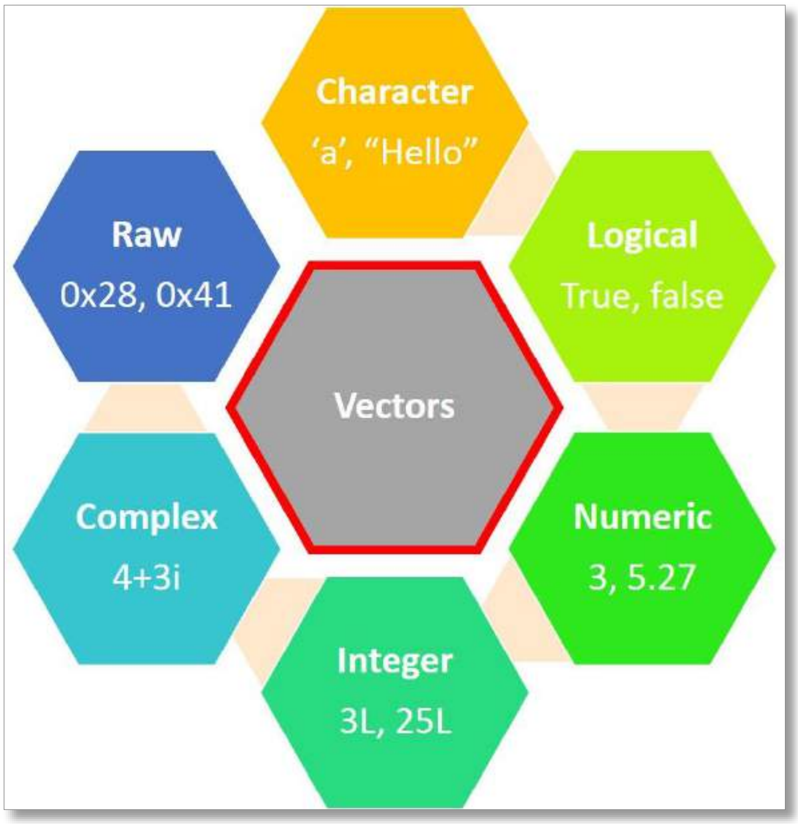


Figure 2.4 – Classes of Objects

2.5. Data Structures

R has a number of data structures for holding data. These include:

- Vectors including Scalars
- Matrices
- Arrays
- Data frames, and
- Lists.

These data structures differ in a number of ways such as the type of data they can hold, the way they are created, their structural complexity, and the way to identify and access the individual elements. Figure 2.5 gives an illustration of these data structures.

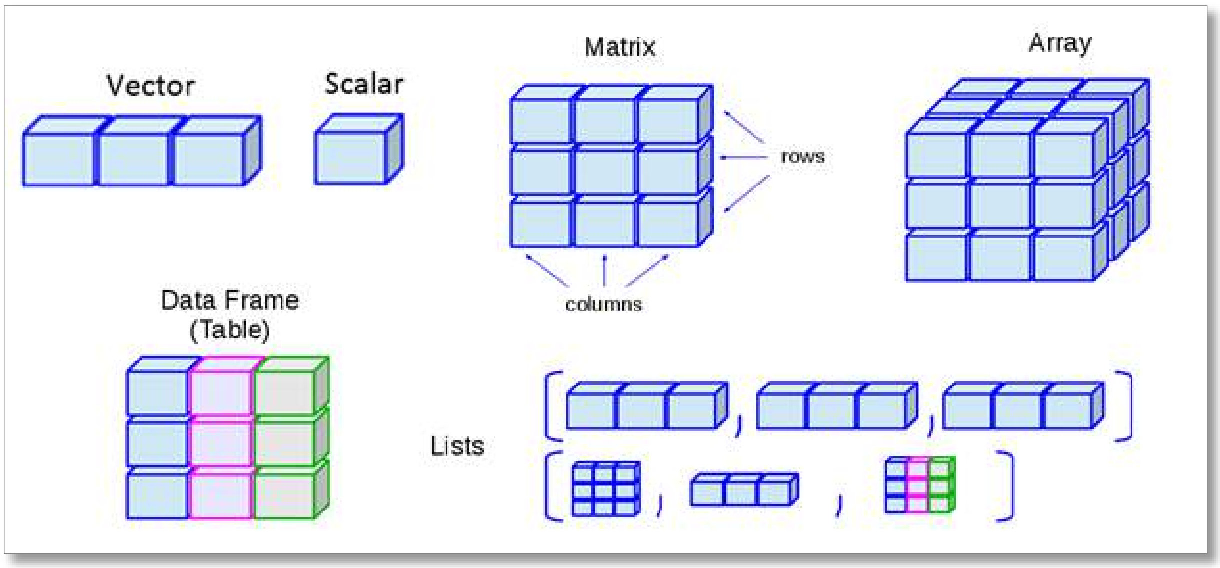


Figure 2.5 – R Data Structures

2.5.1. Vectors and Scalars

A vectors is a one-dimensional array that can hold numeric data, character data, or logical data. Scalars are one-element vectors. Figures 2.6 gives an illustration of a vector and a scalar.

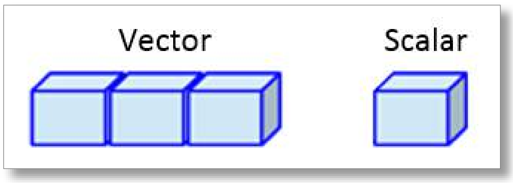


Figure 2.6 – Vector and Scalar

2.5.1.1. Vectors and Scalars

The following example shows the creation of a scalar which is a single value vector.

Example

```
a <- 10
```

```
b <- "R Programming is fun"
c <- TRUE

print(a)
print(b)
print(c)
```

The above code produces the following output.

Program Output

```
[1] 10

[1] "R Programming is fun"

[1] TRUE
```

2.5.1.2. Vector Creation

A generic function *c()* which combines its arguments is used to form multiple element vectors. The following example shows a vector *a* of type integer, a vector *b* of type double, a vector *c* of type character, a logical vector *d*, a complex vector *e* and a raw vector *f*.

Example

```
# vector of type integer
a <- c(1L, 2L, 3L, -4L, -2L, 7L)

# vector of type double
b <- c(12.5,8.7,-5.1,2.8)

# vector of type character
c <- c("item1", "item2", "item3")

# vector of type logical
d <- c(TRUE, FALSE, TRUE, FALSE, FALSE, TRUE)

# vector of type complex
e <- 3+2i

# vector of type raw
f <- charToRaw("Hello World")

print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
```

The above code produces the following output.

Program Output

```
[1]  1  2  3 -4 -2  7

[1] 12.5  8.7 -5.1  2.8

[1] "item1" "item2" "item3"
```

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

[1] 3+2i

[1] 48 65 6c 6c 6f 20 57 6f 72 6c 64
```

Note that vectors can contain data of the same type (numeric, character, or logical). If the arguments are of differing data types, they are coerced to a common type. E.g. The non-character values are coerced to the character type if one of the elements is a character.

Example

```
# Logical and numerical values are coerced to characters
a <- c("Red", 5L, 2.3, TRUE, 6)
print(a)
```

The above code produces the following output.

Program Output

```
[1] "Red"  "5"    "2.3"  "TRUE" "6"
```

2.5.1.3. Using the Colon and Sequence Operators

The colon operator (:) or sequence operator, *seq()*, can also be used to create multiple element vectors. The following example shows the use of the colon and sequence operators.

Example

```
# Creating a sequence from 1 to 10
a <- 1:10
print(a)

# Creating a sequence from -5.5 to 3.5
b <- -5.5:3.5
print(b)

# Discarding final element if not in sequence
c <- 2.2:8.8
print(c)

# Create vector with elements from 2.2 to 10 incrementing by 0.8
print(seq(2, 10, by = 0.8))
print(d)
```

The above code produces the following output.

Program Output

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

[1] -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5  1.5  2.5  3.5

[1] 2.2 3.2 4.2 5.2 6.2 7.2 8.2

[1]  2.0  2.8  3.6  4.4  5.2  6.0  6.8  7.6  8.4  9.2 10.0
```

2.5.1.4. Accessing Vector Elements

The elements of a vector can be accessed by using a numeric vector of positions within brackets. The indexing starts with 1. A negative index will the corresponding element from the result. Logical indexing, i.e. using TRUE, FALSE, can also be used for indexing.

The following code listing shows how the vector elements can be accessed by using indexes and logical indexing.

Example

```
# Accessing vector elements using indexes
days <- c("Mon","Tue","Wed","Thurs","Fri","Sat","Sun")
working <- days[c(1:5)]
print(working)

# Accessing vector elements using logical indexing
weekend <- days[c(FALSE,FALSE,FALSE,FALSE,FALSE,TRUE,TRUE)]
print(weekend)

# Dropping Elements with negative indices
tuition <- days[c(-1,-3,-7:-5)]
print(tuition)
```

The above code produces the following output.

Program Output

```
[1] "Mon"    "Tue"    "Wed"    "Thurs" "Fri"

[1] "Sat" "Sun"

[1] "Tue"    "Thurs"
```

2.5.2. Vector Manipulation

This section covers the main operations that can be performed on vectors, such as arithmetic, recycling and sorting.

2.5.2.1. Vector arithmetic

Vectors of the same length can be added, subtracted, multiplied or divided, producing another vector as output.

The following code listing shows examples of vector creation, addition, subtraction, multiplication and division.

Example

```
# Vector Creation
vec1 <- c(2,5,9,-3,6)
vec2 <- c(4,-2,6,12,5)

# Vector addition
add <- vec1+vec2
print(add)

# Vector subtraction
```

```
sub <- vec1-vec2
print(sub)

# Vector multiplication
multi <- vec1*vec2
print(multi)

# Vector division
div <- vec1/vec2
print(div)
```

The above code produces the following output.

Program Output

```
[1]  6  3 15  9 11

[1]  -2   7   3 -15   1

[1]   8 -10  54 -36  30

[1]  0.50 -2.50  1.50 -0.25  1.20
```

Activity

- *The measurements of four cylinders are as follows:*
- *Their height are: 8, 6, 5.5, 10 and,*
- *Their radius are: 1.5, 3, 4, 0.5*
- *Read these data into two vectors by giving the vectors appropriate names.*
- *Calculate the volume of each cylinder as follows:*

  *Volume = pi * radius * radius * height*
- *The Volumes should be saved in another vector and displayed accordingly.*
- *Save the above script as **Activity 2_5_2_1***

2.5.2.2. Vector Recycling

If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. In the following example, vectors *vec1* and *vec2* have unequal lengths. Therefore *vec1* will be recycled.

Example

```
# Vector Creation
vec1 = c(10, 20, 30)
vec2 = c(1, 2, 3, 4, 5, 6, 7, 8, 9)

# Vector addition
add <- vec1+vec2

# vec1 is recycled to c(10,20,30,10,20,30,10,20,30)
print(add)

# Vector subtraction
sub <- vec1-vec2
print(sub)
```

The above code produces the following output.

Program Output

```
[1]  6  3 15  9 11

[1]  -2   7   3 -15   1

[1]   8 -10  54 -36  30

[1]  0.50 -2.50  1.50 -0.25  1.20
```

2.5.2.3. Vector Sorting

The elements of a vector can be sorted using the *sort()* function as shown in the example below. By default, the elements are sorted in ascending order but they can also be sorted in descending order by setting the optional parameter decreasing to TRUE. Characters are also sorted based on their character code with lowercase letters appearing first if the vector is sorted in ascending order.

Example

```
# Vector Creation
vec1 <- c(2,5,7,6,-2,3)
vec2 <- c("Red","blue","Yellow","green")

# Sort vector elements
sorted <- sort(vec1)
print(sorted)

# Sort in the reverse order
revsort <- sort(vec1, decreasing = TRUE)
print(revsort)

# Sorting character
sorted <- sort(vec2)
print(sorted)

# Sorting in reverse order
revsort <- sort(vec2, decreasing = TRUE)
print(revsort)
```

The above code produces the following output.

Program Output

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

[1]  7  6  5  3  2 -2

[1] "blue"   "green"  "Red"    "Yellow"

[1] "Yellow" "Red"    "green"  "blue"
```

Activity

- *The following script contain some common errors. Copy and paste the faulty code into a new R script. Analyse the code and remove the errors so that the script can execute.*

```
vector1 <- c('one', 'two, 'three', 'four')
vec.var <- var(c(1, 3, 3, 4, 5,))
vec.mean <- mean(c(1, 3, 3, 4, 5)
vec.Min <- Min(c(5, 4, 3, 2, 1))
vec.max <- maxx(c(5, 4, 3, 2, 1))
vector2 <- c('a', 'b', 'f', 'g")

vec.var
vec.mean
vec.min
vec.max
vector2
```

- *Save the above script as **Activity 2_5_2_3***

2.5.3. Matrices

A matrix is R object in which the elements are arranged in a two-dimensional rectangular layout where each element has the same atomic type (numeric, character, or logical). Though it is possible to create matrices with only characters or logical values, in most cases, matrices containing numeric elements are created and used in mathematical calculations. Figure 2.7 gives an illustration of a matrix consisting of three rows and 3 columns.

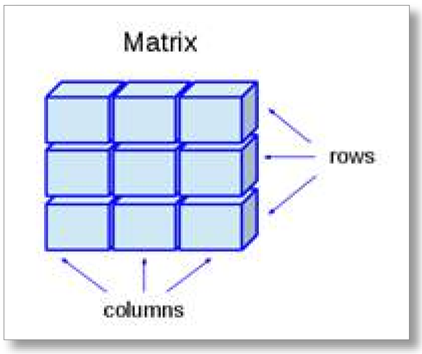


Figure 2.7 – Matrix

Matrices are created with the *matrix()* function. The general syntax is as follows:

Syntax

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

- **data** is the vector contains the elements of the matrix
- **nrow** is the row dimension (number of rows of the matrix)
- **ncol** is the column dimension (number of columns).
- **byrow** is a logical value. If set to TRUE, then the vector elements are arranged by row.
- **dimnames** contains optional row and column labels.

The following example shows different ways of creating a matrix.

Example

```
# Elements arranged sequentially by row
mat1 <- matrix(c(1:6), nrow = 3, byrow = TRUE)
print(mat1)

# Elements arranged sequentially by column
mat2 <- matrix(c(1:6), nrow = 3, byrow = FALSE)
print(mat2)

# Elements arranged sequentially using ncol and default col arrangment
mat3 <- matrix(c(7:12), ncol = 3)
print(mat3)

# Labelling the columns and rows.
rownames = c("row1", "row2", "row3")
colnames = c("col1", "col2")

mat4 <- matrix(c(13:18), nrow = 3, byrow = TRUE, dimnames = list(rownames,
colnames))
print(mat4)
```

The above code produces the following output.

Program Output

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

     col1 col2
row1   13   14
row2   15   16
row3   17   18
```

2.5.3.1. Combining Matrices

The columns of two matrices having the same number of rows can be combined into a larger matrix using the *cbind(),* which stands for column bind, function. In the following example, *matrixB* has 3 rows and 2 columns, and *matrixC* has 3 rows and 1 column. *MatrixB* and *matrixC* have been combined using the *cbind()* function to form *matrixA.*

Example

```
# Creating matrix B with 3 rows and 2 cols
matrixB = matrix(c(2, 4, 3, 1, 5, 7), nrow=3)
cat("MatrixB \n")
print(matrixB)
```

```
# Creating matrix B with 3 rows and 1 col
matrixC = matrix(c(7, 4, 2), nrow=3)
cat("MatrixC \n")
print(matrixC)

# Combine 2 cols from B and 1 col from C to create A
matrixA=cbind(matrixB,matrixC)
cat("MatrixA \n")
print(matrixA)
```

The above code produces the following output.

Program Output

```
MatrixB
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7

MatrixC
[1,]    7
[2,]    4
[3,]    2

MatrixA
     [,1] [,2] [,3]
[1,]    2    1    7
[2,]    4    5    4
[3,]    3    7    2
```

Similarly, the rows of two matrices having the same number of columns can be combined into a larger matrix using the *rbind()*, which stands for row bind, function. In the following example, *matrixB* has 2 rows and 3 columns, and *matrixC* has 1 row and 3 columns. *MatrixB* and *matrixC* have been combined using the *rbind()* function to form *matrixA*.

Example

```
# Creating matrix B with 3 cols and 2 rows
matrixB = matrix(c(2, 4, 3, 1, 5, 7), ncol=3)
cat("MatrixB \n")
print(matrixB)

# Creating matrix B with 3 cols and 1 row
matrixC = matrix(c(7, 4, 2), ncol=3)
cat("MatrixC \n")
print(matrixC)

# Combine 2 rows from B and 1 row from C to create A
matrixA=rbind(matrixB,matrixC)
cat("MatrixA \n")
print(matrixA)
```

The above code produces the following output.

Program Output

```
MatrixB
     [,1] [,2] [,3]
[1,]    2    3    5
[2,]    4    1    7

MatrixC
     [,1] [,2] [,3]
[1,]    7    4    2

MatrixA
     [,1] [,2] [,3]
[1,]    2    3    5
[2,]    4    1    7
[3,]    7    4    2
```

2.5.3.2. Accessing Elements of a Matrix

An element of a matrix can be accessed by using its column and row index as shown in the example below.

Example

```
# Matrix Creation
rownames = c("row1", "row2", "row3")
colnames = c("col1", "col2")

mat <- matrix(c(13:18), nrow = 3, byrow = TRUE, dimnames = list(rownames,
colnames))
print(mat)

# Access the element in 1st row 2nd col
print(mat[1,2])

# Access the element at 3rd row 1st col
print(mat[3,1])

# Access only the 2nd row
print(mat[2,])

# Access only the 1st col
print(mat[,1])
```

The above code produces the following output.

Program Output

```
     col1 col2
row1   13   14
row2   15   16
row3   17   18

[1] 14

[1] 17

col1 col2
  15   16
```

```
row1 row2 row3
  13   15   17
```

Activity	• *Create three vectors x, y and z with each vector having 3 elements.*
	• *Vector x has integer elements 1, 4 and 5.*
	• *Vector y has integer elements 4, 9 and 6.*
	• *Vector z has integer elements 2, 1 and 7.*
	• *Combine the above 3 vectors to form the following matrix A:*
	`     x y z`
	`[1,] 1 4 2`
	`[2,] 4 9 1`
	`[3,] 5 6 7`
	• *Change the row names to a, b and c.*
	• *Save the above script as **Activity 2_5_3_2***

Activity	• *Create a vector with integers 1 to 12. Convert the vector to a 4 x 3 matrix B. Note that the column names should be x, y, z and the row names a, b, c, d.*
	• *Matrix B should therefore be as follows:*
	`  x y  z`
	`a 1 5  9`
	`b 2 6 10`
	`c 3 7 11`
	`d 4 8 12`

2.5.3.3. Matrix Arithmetic

Various arithmetic operations can be performed on matrices resulting in another matrix. However, the dimensions of the matrices involved in the operations should be the same, i.e. the matrices' number of rows and columns should match. The example below shows the four arithmetic operations being performed on two 2x3 matrices.

Example

```
# Create two 2x3 matrices
mat1 <- matrix(c(5, 7, -3, 4, 10, -1), nrow = 2)
print(mat1)

mat2 <- matrix(c(5, 4, 6, -5, 4, 2), nrow = 2)
print(mat2)

# Add the matrices
add <- mat1 + mat2
cat("Result of addition","\n")
print(add)

# Subtract the matrices
sub <- mat1 - mat2
cat("Result of subtraction","\n")
print(sub)
```

```
# Multiply the matrices
result <- mat1 * mat2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- mat1 / mat2
cat("Result of division","\n")
print(result)
```

The above code produces the following output.

Program Output

```
     [,1] [,2] [,3]
[1,]    5   -3   10
[2,]    7    4   -1

     [,1] [,2] [,3]
[1,]    5    6    4
[2,]    4   -5    2

Result of addition
     [,1] [,2] [,3]
[1,]   10    3   14
[2,]   11   -1    1

Result of subtraction
     [,1] [,2] [,3]
[1,]    0   -9    6
[2,]    3    9   -3

Result of multiplication
     [,1] [,2] [,3]
[1,]   25  -18   40
[2,]   28  -20   -2

Result of division
     [,1] [,2] [,3]
[1,] 1.00 -0.5  2.5
[2,] 1.75 -0.8 -0.5
```

Activity	

Activity	• *This activity continues from the previous activity where you had created matrices A and B.*
	• *Try the following: C = B + A*
	• *You should get the following error:*
	`Error in B + A : non-conformable arrays`
	• *This is due to the fact that B is a 4 x 3 matrix while A is a 3 x 3 matrix.*
	• *Create a vector z1 with integer values: 5, 9 and 0*
	• *Using rbind(), add this vector to A and rename the rows.*
	• *Now, type C = B + A and display C.*
	• *You should have the following result:*
	`  x  y  z`
	`a 2  9 11`
	`b 6 15 11`
	`c 8 13 18`
	`d 9 17 12`
	• *Save the above script as **Activity 2_5_3_3***

2.5.4. Arrays

Arrays are similar to matrices but can store data in more than two dimensions. E.g. an array of dimension (3, 4, 5) creates 5 rectangular matrices each with 3 rows and 4 columns. Similar to vectors and matrices, arrays can store data of only one data type. Figure 2.8 gives an illustration of an array.

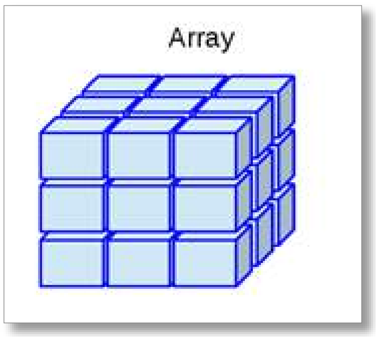


Figure 2.8 – Array

Arrays are created with an *array()* function and its general syntax is as follows:

Syntax

```
myarray <- array(vector, dimensions, dimnames)

   •  vector contains the data for the array
   •  dimensions is the maximum index for each dimension
   •  dimnames contains optional dimension labels.
```

The following listing gives an example of creating a three-dimensional (2x3x4) array of numbers.

Example

```
# Create a (2x3x4) array
array1 <- array(c(1:24),dim = c(2,3,4))
print(array1)
```

The above code produces the following output.

Program Output

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

Note it!

- *The* sample() *function is used to generate a random integer number. E.g.* `sample(1:10, 1)` *has as first argument a vector of valid numbers (1 to 10), and as second argument,* `1` *which indicates that one number should be returned.*

Example

```
# Create a (2x3x4) array
array1a <- array(sample(1:60,24),dim = c(2,3,4))
print(array1a)
```

The above code produces the following output.

Program Output

```
, , 1

     [,1] [,2] [,3]
[1,]    9   32   17
[2,]   27   57   12
```

```
, , 2

     [,1] [,2] [,3]
[1,]    7   25   52
[2,]   46   54   10

, , 3

     [,1] [,2] [,3]
[1,]   59   29   30
[2,]   34   19   36

, , 4

     [,1] [,2] [,3]
[1,]    2   18   56
[2,]   39   53   11
```

2.5.4.1. Naming Columns and Rows

Arrays can be created with vectors of different lengths. Moreover, names can be given to the rows, columns and matrices in the array by using the *dimnames* parameter as shown in the following example.

Example

```
# Create two vectors of different lengths
vector1 <- c(seq(5,15,by=2))
vector2 <- c(1,2,3)

# Naming the rows, columns and matrices
column.names <- c("Col1","Col2","Col3")
row.names <- c("Row1","Row2","Row3")
matrix.names <- c("Matrix1","Matrix2")

# Inputting the vectors, dimensions and names to the array
array2    <-    array(c(vector1,vector2),dim    =    c(3,3,2),dimnames    =
list(row.names,column.names,matrix.names))
print(array2)
```

The above code produces the following output.

Program Output

```
, , Matrix1

     Col1 Col2 Col3
Row1    5   11    1
Row2    7   13    2
Row3    9   15    3

, , Matrix2

     Col1 Col2 Col3
Row1    5   11    1
Row2    7   13    2
Row3    9   15    3
```

2.5.4.2. Accessing Array Elements

As it can be seen, arrays are a natural extension of matrices. Consequently, the identification of array elements follows from matrices. The following example prints some elements of the array defined in the previous example.

Example

```
# Print the element in the 2nd row and 3rd column of the 1st matrix
print(array2[2,3,1])

# Print the 1st row of the second matrix of the array
print(array2[1,,2])

# Print the 2nd Matrix.
print(array2[,,2])
```

The above code produces the following output.

Program Output

```
[1] 2

Col1 Col2 Col3
   5   11    1

     Col1 Col2 Col3
Row1    5   11    1
Row2    7   13    2
Row3    9   15    3
```

Activity

- *Create a 4x3x2 array of 24 elements using the random values between 1 and 50*
- *Name the columns, rows and matrices using names of your choice*
- *Print the array*
- *Print the second matrix*
- *Print the last row of the second matrix*
- *print the second column of the first matrix*
- *Save the above script as* ***Activity 2_5_4_2***

2.5.4.3. Manipulating Elements of an Array

As mentioned previously, an array is made up of matrices in multiple dimensions. Therefore, the elements of the matrices can be accessed and used to manipulate the elements of the array. In the following example, two (3x3x2) arrays, *array1* and *array2*, are created from vectors of different lengths. Two matrices are then extracted as follows: *matrix1* from the first matrix of *array1* and *matrix2* from the second matrix of *array2*. These two matrices are then arithmetically manipulated (using subtraction and addition) to yield *matrix3* and *matrix4*. *Array3* is finally created as a combination of *matrix3* and *matrix4*.

Example

```
# Naming the rows, columns and matrices
column.names <- c("Col1","Col2","Col3")
row.names <- c("Row1","Row2","Row3")
matrix.names <- c("Matrix1","Matrix2")

# Create two vectors of different lengths for array1
vector1 <- c(seq(5,21,by=2))
vector2 <- c(1,2,3)

# Inputting the vectors, dimensions and names to the array, array1
array1    <-    array(c(vector1,vector2),dim    =    c(3,3,2),dimnames    =
list(row.names,column.names,matrix.names))
cat ("Array1\n")
print(array1)

# Create two vectors of different lengths for array2
vector3 <- c(11,12,13)
vector4 <- c(1,-10,5,1,3,-2,6,2,9)

# Creating array2
array2    <-    array(c(vector3,vector4),dim    =    c(3,3,2),dimnames    =
list(row.names,column.names,matrix.names))
cat ("Array2\n")
print(array2)

# create matrices from the first matrix of these arrays
matrix1 <- array1[,,1]
matrix2 <- array2[,,2]

cat ("Matrix1 - 1st Matrix of Array1\n")
print(matrix1)

cat ("Matrix2 - 2nd Matrix of Array2\n")
print(matrix2)

# Subtracting the matrices to get array3
matrix3 <- matrix1 - matrix2
matrix4 <- matrix1 + matrix2

cat ("Matrix3\n")
print(matrix3)

cat ("Matrix4\n")
print(matrix4)

# Creating array3 from matrix3 and matrix4
array3    <-    array(c(matrix3,matrix4),dim    =    c(3,3,2),dimnames    =
list(row.names,column.names,matrix.names))
cat ("Array3 made up from Matrix 3 and Matrix 4\n")
print(array3)
```

The above code produces the following output.

Program Output

```
Array1

, , Matrix1

     Col1 Col2 Col3
Row1    5   11   17
Row2    7   13   19
Row3    9   15   21

, , Matrix2

     Col1 Col2 Col3
Row1    1    5   11
Row2    2    7   13
Row3    3    9   15

Array2

, , Matrix1

     Col1 Col2 Col3
Row1   11    1    1
Row2   12  -10    3
Row3   13    5   -2

, , Matrix2

     Col1 Col2 Col3
Row1    6   11    1
Row2    2   12  -10
Row3    9   13    5

Matrix1 - 1st Matrix of Array1
     Col1 Col2 Col3
Row1    5   11   17
Row2    7   13   19
Row3    9   15   21

Matrix2 - 2nd Matrix of Array2
     Col1 Col2 Col3
Row1    6   11    1
Row2    2   12  -10
Row3    9   13    5

Matrix3
     Col1 Col2 Col3
Row1   -1    0   16
Row2    5    1   29
Row3    0    2   16

Matrix4
     Col1 Col2 Col3
Row1   11   22   18
Row2    9   25    9
Row3   18   28   26

Array3 made up from Matrix 3 and Matrix 4
, , Matrix1
```

```
     Col1 Col2 Col3
Row1   -1    0   16
Row2    5    1   29
Row3    0    2   16

, , Matrix2

     Col1 Col2 Col3
Row1   11   22   18
Row2    9   25    9
Row3   18   28   26
```

Activity

- *Copy and run the above example in a new R script.*
- *Create an array, array4, with the second matrix of array1 and the first matrix of array2.*
- *From array4, subtract array3 and save it as array5*
- *Print array5 which should display the following:*

```
, , Matrix1

     Col1 Col2 Col3
Row1    2    5   -5
Row2   -3    6  -16
Row3    3    7   -1

, , Matrix2

     Col1 Col2 Col3
Row1    0  -21  -17
Row2    3  -35   -6
Row3   -5  -23  -28
```

- *Save the above script as* ***Activity 2_5_4_3***

2.5.4.4. Calculations across the Elements of an Array

Calculations can performed across the array elements using the *apply()* function and its general syntax is as follows:

Syntax

```
apply(x, margin, fun)
```

- **x** is an array, including a matrix.
- **Margin** is a vector giving the subscripts which the function will be applied over. 1 indicates rows, 2 indicates columns, c(1,2) indicates rows and columns.
- **fun** is the function (any R or user-defined function) to be applied across the elements of the array.

The following example shows the use of the:

1. Sum function applied to the elements of each row

2. Product function applied to the elements of each column
3. Mean function applied to the elements of each column, and,
4. Sum function applied to the respective elements across all matrices

Example

```
# Naming the rows, columns and matrices
column.names <- c("Col1","Col2","Col3")
row.names <- c("Row1","Row2","Row3")
matrix.names <- c("Matrix1","Matrix2")

# Create two vectors of different lengths for array1
vector1 <- c(seq(5,21,by=2))
vector2 <- c(1,2,3)

# Inputting the vectors, dimensions and names to the array, array1
array1   <-   array(c(vector1,vector2),dim   =   c(3,3,2),dimnames   =
list(row.names,column.names,matrix.names))
cat ("Array1\n")
print(array1)

# Use apply to calculate the sum of the rows across all matrices
rows.sum <- apply(array1, c(1), sum)
cat("rows.sum \n")
print(rows.sum)

# Use apply to calculate the product of the columns across all matrices
cols.prod <- apply(array1, c(2), prod)
cat("cols.prod \n")
print(cols.prod)

# Use apply to calculate the mean of the columns across all matrices
cols.mean <- apply(array1, c(2), mean)
cat("cols.mean \n")
print(cols.mean)

#  Use  apply  to  calculate  the  sum  of  the  respective  elements  across  all
matrices
matrix.sum <- apply(array1, c(1,2), sum)
cat("matrix.sum \n")
print(matrix.sum)
```

The above code produces the following output.

Program Output

```
Array1
, , Matrix1

     Col1 Col2 Col3
Row1    5   11   17
Row2    7   13   19
Row3    9   15   21

, , Matrix2

     Col1 Col2 Col3
Row1    1    5   11
Row2    2    7   13
Row3    3    9   15
```

```
rows.sum
Row1 Row2 Row3
  50   61   72

cols.prod
    Col1     Col2     Col3
    1890   675675 14549535

cols.mean
Col1 Col2 Col3
 4.5 10.0 16.0

matrix.sum
     Col1 Col2 Col3
Row1    6   16   28
Row2    9   20   32
Row3   12   24   36
```

Activity	• *This activity continues from the previous activity.* • *For array5, calculate and display the:* • *sum of the rows across all matrices* • *sum of the columns across all matrices* • *product of the rows across all matrices* • *product of the columns across all matrices* • *mean of the columns across all matrices* • *the sum of the respective elements across all matrices* • *Save the above script as* ***Activity 2_5_4_4***

2.5.5. Data Frames

A data frame is a two-dimensional array-like structure or a table in which each column can contain different types of data (numeric, character, etc...) and each row contains one set of values from each column. A data frame is one of the most common data structures that are used in R and is similar to the datasets manipulated in statistical analysis tools such as *SAS*, *SPSS*, and *Stata*. Figure 2.9 gives an illustration of a data frame.

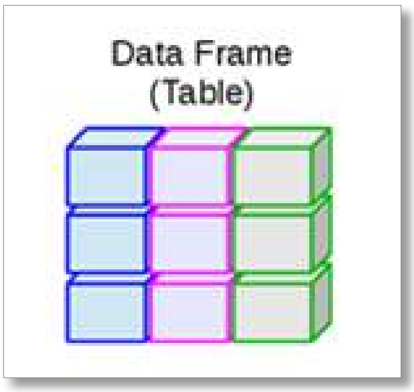


Figure 2.9 – Data Frame

A data frame is created with the *data.frame()* function and its general syntax is as follows:

Syntax

```
mydata <- data.frame(col1, col2, col3,…)

   •  col1, col2, col3, … are column vectors of any type (such as character,
      numeric, or logical)
   •  Names for each column can be provided with the names function.
```

The following code listing creates a data frame with information about students. Five vectors are initially created: *stud.id*, *stud.name*, *stud.dob*, *stud.course*, *stud.cpa* and used in the *data.frame()* function to create the data frame stud.

Example

```
# Setting the vectors
stud.id = c(1011:1015)
stud.name = c("John", "Mary", "Peter", "Janet", "Williams")
stud.dob  = as.Date(c("1998-02-05", "1998-11-11", "1997-12-26", "1996-09-
24", "1997-10-05"))
stud.course = c("IC320", "IC311", "E565", "E318", "E319")
stud.cpa = c(70.9, 44.7, 83.4, 68.4, 51.9)

# Create the data frame
stud = data.frame(stud.id, stud.name, stud.dob, stud.course, stud.cpa)
print(stud)
```

The above code produces the following output.

Program Output

```
  stud.id stud.name   stud.dob stud.course stud.cpa
1    1011      John 1998-02-05       IC320     70.9
2    1012      Mary 1998-11-11       IC311     44.7
3    1013     Peter 1997-12-26        E565     83.4
4    1014     Janet 1996-09-24        E318     68.4
5    1015  Williams 1997-10-05        E319     51.9
```

2.5.5.1. The Str and Summary functions

The *str()* function compactly display the internal **str**ucture of an R object, in this case, a data frame. *Summary()* is a generic function that can be used to produce result summaries of a data frame. The following example shows how to display the internal structure of the *stud* data frame using the *str()* function and also displays its summary using the *summary()* function.

Example

```
# Display the internal structure
str(stud)

# Display the summary
print(summary(stud))
```

The above code produces the following output.

Program Output

```
'data.frame':   5 obs. of  5 variables:
 $ stud.id    : int  1011 1012 1013 1014 1015
 $ stud.name  : Factor w/ 5 levels "Janet","John",..: 2 3 4 1 5
 $ stud.dob   : Date, format: "1998-02-05" "1998-11-11" ...
 $ stud.course: Factor w/ 5 levels "E318","E319",..: 5 4 3 1 2
 $ stud.cpa   : num  70.9 44.7 83.4 68.4 51.9

    stud.id          stud.name     stud.dob             stud.course     stud.cpa
 Min.   :1011   Janet    :1   Min.   :1996-09-24   E318 :1      Min.   :44.70
 1st Qu.:1012   John     :1   1st Qu.:1997-10-05   E319 :1      1st Qu.:51.90
 Median :1013   Mary     :1   Median :1997-12-26   E565 :1      Median :68.40
 Mean   :1013   Peter    :1   Mean   :1997-11-20   IC311:1      Mean   :63.86
 3rd Qu.:1014   Williams:1    3rd Qu.:1998-02-05   IC320:1      3rd Qu.:70.90
 Max.   :1015                 Max.   :1998-11-11                Max.   :83.40
```

Activity

- *Create the following data frame.*

```
     Name Age Weight Sex
1    John  15     49   M
2    Mary  20     71   F
3   Peter  17     58   M
4 William  19     62   M
5    Kate  25     67   F
```

- *Display the internal structure of the data frame.*
- *Save the above script as* ***Activity 2_5_5***

2.5.5.2. Extracting data from the Data Frame

The elements of a data frame can be extracted by using the subscript notation, used previously with matrices, or by specifying the column names. The $ notation can also be used to refer to a specific variable from a given data frame. Using the *stud* data frame created earlier, the following code listing demonstrates these approaches.

Example

```
# Extract first 2 columns using indexes
stud[1:2]

# Extract Specific columns
stud[c("stud.course", "stud.cpa")]

# Extract Specific columns using the $ notation
result <- data.frame(stud$stud.course,stud$stud.cpa)
print(result)

# Extract 2nd and 3rd rows
stud[2:3,]

# Extract 1st and 4th row with 2nd and 5th column
stud[c(1,4),c(2,5)]
```

The above code produces the following output.

Program Output

```
  stud.id stud.name
1    1011      John
2    1012      Mary
3    1013     Peter
4    1014     Janet
5    1015  Williams

  stud.course stud.cpa
1       IC320     70.9
2       IC311     44.7
3        E565     83.4
4        E318     68.4
5        E319     51.9

  stud.stud.course stud.stud.cpa
1            IC320          70.9
2            IC311          44.7
3             E565          83.4
4             E318          68.4
5             E319          51.9

  stud.id stud.name   stud.dob stud.course stud.cpa
2    1012      Mary 1998-11-11       IC311     44.7
3    1013     Peter 1997-12-26        E565     83.4

  stud.name stud.cpa
1      John     70.9
4     Janet     68.4
```

Activity

- *This activity follows from the previous one.*
- *From the data frame, extract the age and the weight.*
- *Extract the name and the sex and save the result in another data frame*
- *Display this new data frame.*
- *Save the updated script as* ***Activity 2_5_5_2***

2.5.5.3. Adding more data to an existing Data Frame

Columns and rows can be added to expand an existing data frame. A column can be added to a data frame by adding the column vector using a new column name. The following code listing demonstrates the addition of a new column to an existing data frame.

Example

```
# Setting the vectors
stud.id = c(1011:1015)
stud.name = c("John", "Mary", "Peter", "Janet", "Williams")
stud.dob = as.Date(c("1998-02-05", "1998-11-11", "1997-12-26", "1996-09-
24", "1997-10-05"))
stud.course = c("IC320", "IC311", "E565", "E318", "E319")
stud.cpa = c(70.9, 44.7, 83.4, 68.4, 51.9)

# Create the data frame.
```

```
stud = data.frame(stud.id, stud.name, stud.dob, stud.course, stud.cpa)
cat("stud \n")
print(stud)

# Adding the column Class to the existing data frame
stud$stud.class=c("First Class", "Third Class", "First Class", "Merit",
"Second Class")
print(stud)
```

The above code produces the following output.

Program Output

```
  stud.id stud.name   stud.dob stud.course stud.cpa
1    1011      John 1998-02-05       IC320     70.9
2    1012      Mary 1998-11-11       IC311     44.7
3    1013     Peter 1997-12-26        E565     83.4
4    1014     Janet 1996-09-24        E318     68.4
5    1015  Williams 1997-10-05        E319     51.9

  stud.id stud.name   stud.dob stud.course stud.cpa   stud.class
1    1011      John 1998-02-05       IC320     70.9  First Class
2    1012      Mary 1998-11-11       IC311     44.7  Third Class
3    1013     Peter 1997-12-26        E565     83.4  First Class
4    1014     Janet 1996-09-24        E318     68.4        Merit
5    1015  Williams 1997-10-05        E319     51.9 Second Class
```

Activity

- *This activity follows from the previous one.*
- *The Height column was missed in the previous data frame and is as follows:*

```
Height
175
155
182
167
165
```

- *Add this information column-wise to the previous one.*
- *Display this new data frame.*
- *How many rows and columns does the new data frame have?*
- *Investigate on how can the above information be obtained by using dim().*
- *Save the above script as* ***Activity 2_5_5_3***

With respect to rows, it is imperative that the new rows follow the same structure as in the existing data frame. The rbind() function is then used to add the new rows permanently to the data frame. The following code listing first creates a data frame *stud.newdata* with new rows and then binds these rows to the above data frame *stud* using the *rbind()* function to create the final data frame *stud.final*.

Example

```
# print the existing data frame
cat("The existing data frame \n")
print(stud)

# Create the second data frame
stud.newdata <-   data.frame(
  stud.id = c (1016:1018),
  stud.name = c("Anderson","Christine","Jordan"),
  stud.dob = as.Date(c("1998-05-25","1996-08-13","1997-12-10")),
  stud.course = c("IC320M","IC311","IC320"),
  stud.cpa = c(55.7,65.4,33.8),
  stud.class=c("Second Class","Merit","Fail")
)

# print the new data frame
cat("The new data frame \n")
print(stud.newdata)

# Bind the two data frames
stud.final <- rbind(stud,stud.newdata)
cat("The final data frame \n")
print(stud.final)
```

The above code produces the following output.

Program Output

```
The existing data frame
  stud.id stud.name   stud.dob stud.course stud.cpa   stud.class
1    1011      John 1998-02-05       IC320     70.9  First Class
2    1012      Mary 1998-11-11       IC311     44.7  Third Class
3    1013     Peter 1997-12-26        E565     83.4  First Class
4    1014     Janet 1996-09-24        E318     68.4        Merit
5    1015  Williams 1997-10-05        E319     51.9 Second Class

The new data frame
  stud.id stud.name   stud.dob stud.course stud.cpa   stud.class
1    1016  Anderson 1998-05-25      IC320M     55.7 Second Class
2    1017 Christine 1996-08-13       IC311     65.4        Merit
3    1018    Jordan 1997-12-10       IC320     33.8         Fail

The final data frame
  stud.id stud.name   stud.dob stud.course stud.cpa   stud.class
1    1011      John 1998-02-05       IC320     70.9  First Class
2    1012      Mary 1998-11-11       IC311     44.7  Third Class
3    1013     Peter 1997-12-26        E565     83.4  First Class
4    1014     Janet 1996-09-24        E318     68.4        Merit
5    1015  Williams 1997-10-05        E319     51.9 Second Class
6    1016  Anderson 1998-05-25      IC320M     55.7 Second Class
7    1017 Christine 1996-08-13       IC311     65.4        Merit
8    1018    Jordan 1997-12-10       IC320     33.8         Fail
```

Activity

- *This activity follows from the previous one.*
- *Create another data frame with the following information:*

```
     Name Age Weight Sex Height
1 Micheal  22     82   M    185
2   Janet  17     44   F    169
3    Samy  33     52   F    157
```

- *Add this data frame to the previous data frame and display the updated data frame.*
- *Create a data frame of only the rows 2, 5, 7 and 8 and with only the columns Name, Sex, Height.*
- *The Height of Kate has been wrongly inserted. It should have been 170. Change it in the new data frame*
- *Display the new data frame to see the changed record.*
- *Save the updated script as* ***Activity 2_5_5_3***

2.5.6. Lists

In R, lists are considered as the most complex of the data types. Basically, a list is an ordered collection of objects which can therefore contain different data types such as numbers, strings, vectors, matrices, arrays, another list inside it etc... Figure 2.10 gives an illustration of a list.

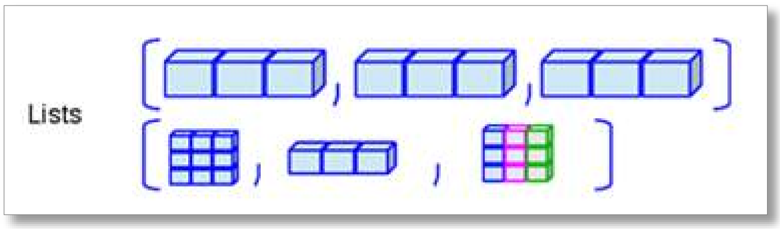


Figure 2.10 – Lists

A list can be created using the *list()* function and its general syntax is as follows:

Syntax

```
mylist <- list(object1, object2, …)

  •  col1, col2, col3, … are column vectors of any type (such as character,
     numeric, or logical)
  •  Names for each column can be provided with the names function.
     E.g. mylist <- list(name1=object1, name2=object2, …)
```

The following listing shows a basic example of list creation.

Example

```
# Create a vector of strings
list.str = c("Red", "Green", "Blue")

# Create a vector of numbers
list.num = c(12, 15, 19, 21, 11)
```

```
# Assign a logical value to a variable
list.log = TRUE

# Create a matrix of sequential numbers
list.mat = matrix(1:8, nrow=4)

# Create a list containing strings, numbers, a logical values and a matrix
list_data <- list(list.str, list.num, list.log, list.mat)
print(list_data)
```

The above code produces the following output.

Program Output

```
[[1]]
[1] "Red"   "Green" "Blue"

[[2]]
[1] 12 15 19 21 11

[[3]]
[1] TRUE

[[4]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

2.5.6.1. List Creation with Named Objects

As mentioned previously, names can be assigned to objects by using the *names()* function. The following example shows firstly, the naming of objects to the already created *list1_data*, and secondly, the creation of *list2_data* with named objects.

Example

```
# Give names to the elements in the list.
names(list1_data) <- c("Colours vector", "Numbers vector", "Logical value",
"Numbers matrix")
cat("List1 \n")
print(list1_data)

# List title
list2.title = "My named list"

# Create a vector of strings
list2.str = c("Yellow", "Purple", "Cyan")

# Create a vector of numbers
list2.num = c(22, -2, 0, 77, 29)

# Assign a logical value to a variable
list2.log = FALSE

# Create a matrix of sequential numbers
list2.mat = matrix(1:10, nrow=5)

# Create a list with names assigned to the objects
```

```
list2_data      <-      list(title=list2.title,      colours=list2.str,
numbers=list2.num, logical=list2.log, matrix=list2.mat)
cat("List2 \n")
print(list2_data)
```

The above code produces the following output.

Program Output

```
List1
$`Colours vector`
[1] "Red"   "Green" "Blue"

$`Numbers vector`
[1] 12 15 19 21 11

$`Logical value`
[1] TRUE

$`Numbers matrix`
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

List2
$title
[1] "My named list"

$colours
[1] "Yellow" "Purple" "Cyan"

$numbers
[1] 22 -2  0 77 29

$logical
[1] FALSE

$matrix
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

2.5.6.2. Accessing List Elements

Similar to vectors and matrices, the elements of a list can be accessed by their indexes. If a list has named objects, these names can also be used to access the elements.

The following example first creates list3_data. Then, the elements of *list3_data* are accessed by using both indexes and names.

Example

```
# Create a list containing a vector, a matrix and a list.
list3_data  <-  list(c("Mon","Tues","Wed",  "Thurs"),  matrix(c(4,7,-2,-
5,0,8), nrow = 2), list("Fri",22.5))

# Give names to the elements in the list.
names(list3_data) <- c("Days", "Values", "Nested List")

cat ("List 3 \n")
print(list3_data)

# Access the first element of the list
cat ("Assessing list element 1 \n")
print(list3_data[1])

# Access the third element which is also a list
cat ("Assessing list element 3 \n")
print(list3_data[3])

# Access the list second element by using its name (Values)
cat ("Assessing element named 'Values' \n")
print(list3_data$Values)
```

The above code produces the following output.

Program Output

```
List 3
$Days
[1] "Mon"   "Tues"  "Wed"   "Thurs"

$Values
     [,1] [,2] [,3]
[1,]    4   -2    0
[2,]    7   -5    8

$`Nested List`
$`Nested List`[[1]]
[1] "Fri"

$`Nested List`[[2]]
[1] 22.5


Assessing list element 1
$Days
[1] "Mon"   "Tues"  "Wed"   "Thurs"

Assessing list element 3
$`Nested List`
$`Nested List`[[1]]
[1] "Fri"

$`Nested List`[[2]]
[1] 22.5


Assessing element named 'Values'
     [,1] [,2] [,3]
[1,]    4   -2    0
[2,]    7   -5    8
```

2.5.6.3. Manipulating the Elements of a List

The elements of a list can be added, updated or deleted. While we can only add or delete elements found at the end of a list, any element at any index can be updated. The following code listing shows the manipulations, including addition, deletion and update, being carried out on *list3_data*.

Example

```
# Printing List 3
cat ("List 3 \n")
print(list3_data)

cat ("List 3 Manipulations \n")

# Add a new element at the end of the list
list3_data[4] <- "New added element"
print(list3_data[4])

# Remove the last element
list3_data[4] <- NULL

# Print the 4th Element
print(list3_data[4])

# Update the 3rd Element
list3_data[3] <- "Updated element"
print(list3_data[3])
```

The above code produces the following output.

Program Output

```
List 3
$Days
[1] "Mon"    "Tues"  "Wed"    "Thurs"

$Values
     [,1] [,2] [,3]
[1,]    4   -2    0
[2,]    7   -5    8

$`Nested List`
$`Nested List`[[1]]
[1] "Fri"

$`Nested List`[[2]]
[1] 22.5


List 3 Manipulations
[[1]]
[1] "New added element"

$<NA>
NULL

$`Nested List`
[1] "Updated element"
```

Activity

- *Create a list Date such that it contains the following information. You may consider creating 3 appropriate vectors.*

```
$year
 [1] 81 82 83 84 85 86 87 88 89 90

$month
 [1]  1  2  3  4  5  6  7  8  9 10 11 12

$day
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[31] 31
```

- *Replace the values of year element in Date list with the years 00 till 10 (2000 – 2010)*
- *Delete the value 2 of the month component of the list Date.*
- *Save the updated script as* ***Activity 2_5_6_3***

2.5.7. Other R Objects

2.5.7.1. Factors and Tables

Factors are vector objects and are used to classify the data and store it as levels. They are very useful in columns with a limited number of values, e.g. *male*, *female* and *true*, *false*. Factors are very useful in statistical analysis and modelling. Factors are created using the *factor()* function with a vector as input. A number of insights can be obtained if the factors are tabulated or if their frequencies are obtained. In R, this can be achieved by using the *table()* function. The following example shows the use of the *factor(), is.factor()* and *table()* functions.

Example

```
# Create a vector as input
gender <- c("male", "female", "male", "male", "male", "female", "male")
print(gender)
print(is.factor(gender))

# Factoring the vector gender
factored.gender = factor(gender)
print(factored.gender)
print(is.factor(factored.gender))

# Get the frequencies of the factors using table
table(factored.gender)
```

The above code produces the following output.

Program Output

```
[1] "male"   "female" "male"   "male"   "male"   "female" "male"

[1] FALSE

[1] male   female male   male   male   female male
Levels: female male

[1] TRUE
```

```
factored.gender
female   male
     2      5
```

2.5.7.2. Factors in Data Frame

If a data frame is made up of columns of text data, R treats the text columns as categorical data and automatically creates factors on them. The following code listing shows how R automatically factors text columns.

Example

```
# Create the vectors for data frame
age <- c(18,22,21,19,18)
gender <- c("male","female","female","female","male")
course <- c("IC320","IC320","IC311","IC311","IC320")

# Create the data frame
stud <- data.frame(age,gender,course)
print(stud)

# Test if the age column is a factor
print(is.factor(stud$age))

# Test if the gender column is a factor
print(is.factor(stud$gender))

# Print the gender column so see the levels
print(stud$gender)

# Get the gender frequencies
table(stud$gender)

# Test if the course column is a factor
print(is.factor(stud$course))

# Print the course column so see the levels
print(stud$course)

# Get the course frequencies
table(stud$course)
```

The above code produces the following output.

Program Output

```
  age gender course
1  18   male  IC320
2  22 female  IC320
3  21 female  IC311
4  19 female  IC311
5  18   male  IC320

[1] FALSE

[1] TRUE

[1] male   female female female male
Levels: female male
```

```
female   male
     3      2

[1] TRUE

[1] IC320 IC320 IC311 IC311 IC320
Levels: IC311 IC320

IC311 IC320
    2     3
```

2.6. Control Structures

In programming languages, control structures allows a programmer to control the flow of execution of a program which is a series of instructions. Basically, these control structures allow the programmer to put some *logic* into the code. Table 2.4 highlights the 2 main types of control structures: decisions and loops, with each one of them having a number of constructs.

Control Structures	Statement	Description
Decisions	*if*	Tests a condition and acts upon it
	if...else	The optional else statement are executed when the Boolean expression is false
	if...else if...else	Tests various conditions and act upon them accordingly
Loops	*while*	Executes a set of instructions as long as a condition is true
	repeat	Executes a loop until the break statement terminates it
	for	Executes a set of instructions a fixed number of times

Table 2.4 – Control Structures

While most of these control structures are used writing functions it is imperative to understand them before actually seeing them in functions.

2.6.1. *if* statement

Like in most programming languages, the *if* control structure is probably the most commonly used in R. This control structure tests a specific condition and acts upon it depending on whether the condition is true or false. Figure 2.11 gives an illustration of the *if* control structure.

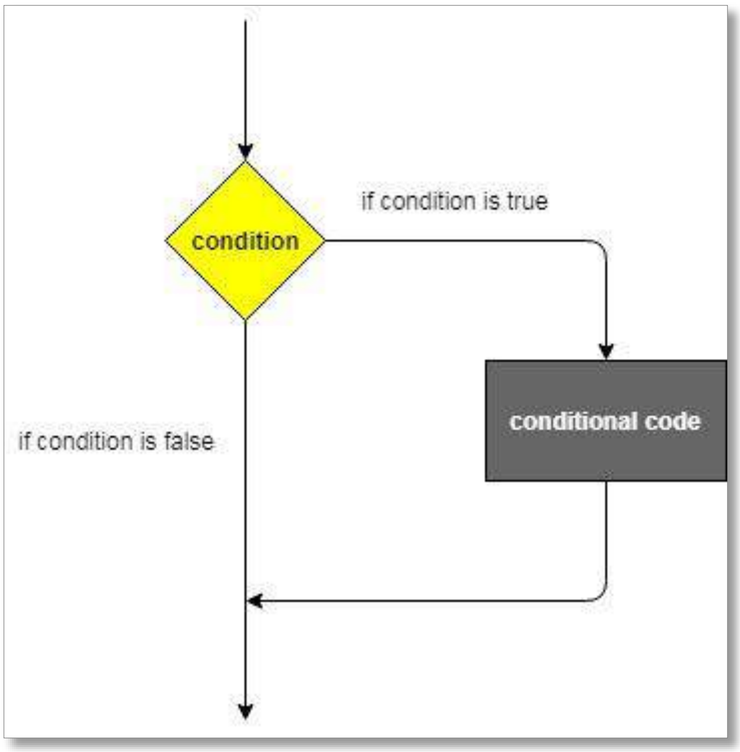


Figure 2.11 – *if* control structure

In R, the basic syntax for creating an *if* control structure is:

Syntax
```
if(boolean_expression) {
   // statement(s) to be executed if the expression is true.
}
```

The following example shows the use of the *if* control structure. The Boolean condition being tested in this example is whether x is greater than 10. Since the variable x has been assigned the value 15, therefore the Boolean condition evaluates to true, and consequently, `x is greater than 10` is displayed.

Example
```
x = 15
if (x > 10) {
  print("x is greater than 10")
}
```

The above code produces the following output.

Program Output
```
[1] "x is greater than 10"
```

2.6.2. The *if...else* and the *if...else if...else* Statement

Variations of the *if* control structure exists in most programming languages. An *if* statement can be followed by:

- An optional *else* statement which will be executed when the Boolean expression evaluates to false.
- An optional *else if...else* statement which is used in situations which require testing of various conditions.

Programmers should be aware of the following important notes when using *if, else if, else* statements:

- An *if* may have zero or one *else* but it must come after all the *else ifs*.
- Similarly, An *if* may have zero or many *else ifs* and they must come before the *else*.
- If an *else if* succeeds, the remaining *else ifs* or *else* will NOT be tested.

The basic syntax for creating an *if...else* and *the if...else if...else* control structures are as follows:

Syntax

- **if...else** control structure

```
if(boolean_expression) {
   // statement(s) to be executed if the expression is true
} else {
   // statement(s) to be executed if the expression is false
}
```

- **if...else if...else** control structure

```
if(boolean_expression 1) {
   // Executes if the boolean expression 1 is true
} else if(boolean_expression 2) {
   // Executes if the boolean expression 2 is true
} else if(boolean_expression 3) {
   // Executes if the boolean expression 3 is true
} else {
   // executes when none of the above condition is true
}
```

The following listing first shows the use of the *if...else* control structure. The *sample()* function is used to generate a random integer number. E.g. `sample(1:10, 1)` has as first argument a vector of valid numbers (1 to 10), and as second argument, `1` which indicates that one number should be returned. The generated value is then compared to 5 and a corresponding message is displayed accordingly. Similarly, the example also shows the use of the *if...else if...else* control structure.

Example

```
cat("An if-else example \n")

#  generate one random number between 1 and 10
x <- sample(1:10, 1)

# print the random number
print (x)

# if-else control structure
if (x >= 5) {
  print("x is greater than or equal to 5")
} else {
  print("x is less than 5")
}

cat("\n An if...else if...else example \n")

#  generate one random number between 1 and 12
y <- sample(1:12, 1)

# print the random number
print (y)

# if-else control structure
if (y >= 9) {
  print("y is greater than or equal to 9")
} else if (y >= 6) {
  print("y is greater than or equal to 6 but less than 9")
} else if (y >= 3) {
  print("y is greater than or equal to 3 but less than 6")
}else {
  print("y is less than 3")
}
```

The above code produces the following output. This is a sample output and will depend on the random value generated.

Program Output

```
An if-else example
[1] 5
[1] "x is greater than or equal to 5"

 An if...else if...else example
[1] 8
[1] "y is greater than or equal to 6 but less than 9"
```

Activity

- *What is the output y in the following:*

  z=5

  *if(z<0) y=z*3 else y=z*5*

- *What is the output n in the following:*

  z='i'

  if (z=='a') n=1 else

  if (z=='e') n=2 else

  if (z=='i') n=3 else

  if (z=='o') n=4 else n=5

- *Save the script as **Activity 2_6_2***

2.6.3. Loops

There may be situations where a group of statements have to be executed a number of times. Like other programming languages, R provides the following loop constructs:

1. The *While* Loop - Repeats a statement or a number of statements as long as a condition is true.
2. *Repeat* loop – Executes a statement or a group of statements a number times and uses the *break* statement to terminate the loop.
3. *For* loop – A counter-controlled loop which repeats a statement or a block of statements a number of times.

Note: With any type of loop, care has to be taken to avoid infinite loops which are loops that run forever.

2.6.3.1. The *While* Loop

The *While* loop is a pre-test loop and therefore tests the condition before executing the body of the loop. Figure 2.12 gives an illustration of the *While* loop.

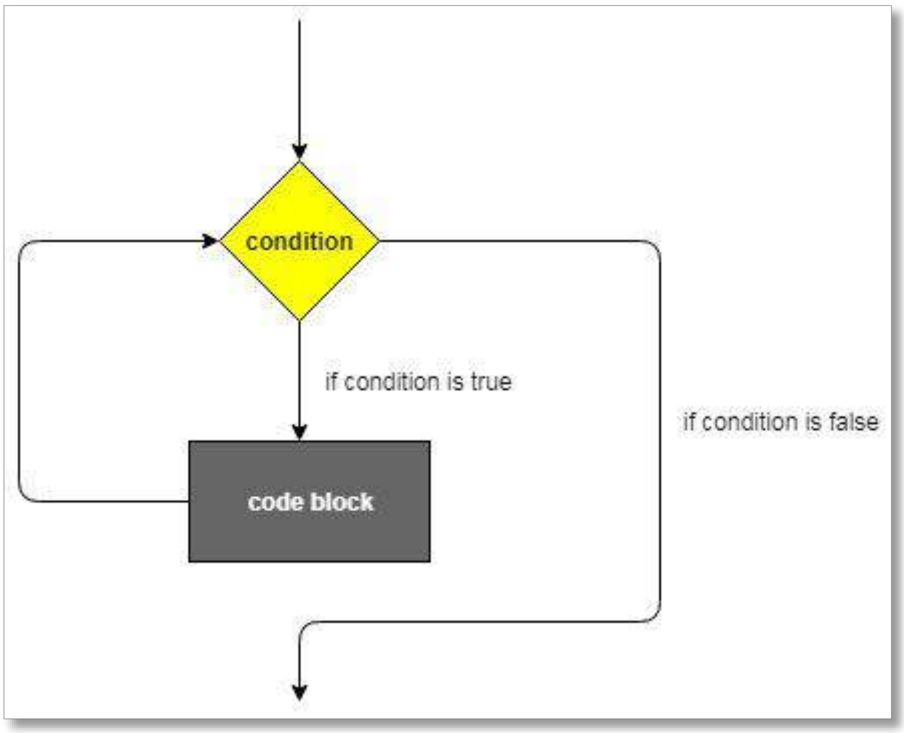


Figure 2.12 – The *While* Loop

The basic syntax for a *While* loop is given below.

Syntax

```
While (condition) {
   Statements …
}
```

In the following example, a loop control variable, *counter*, has been initialised to *one*. The *While* loop condition tests whether the value of *counter* is less than or equal to 5. As long as this condition is true, the statements found inside the loop are iterated. Hence after the fifth execution, the counter variable has value *6* and is hence not less than or equal to 5. Consequently the loop terminates and the program executes the statements found after the loop.

Example

```
# Initialise a loop control variable, counter, to 1
counter <- 1

# While count variable is less than or equal to 5, loop
while (counter <= 5) {
  cat("This line is being executed", counter, "time(s). \n")

  # Increment counter variable by 1
  counter = counter + 1
}

cat("This is the end of the loop. \n")
cat("Loop control variable counter has value:", counter)
```

The above code produces the following output.

Program Output

```
This line is being executed 1 time(s).
This line is being executed 2 time(s).
This line is being executed 3 time(s).
This line is being executed 4 time(s).
This line is being executed 5 time(s).
This is the end of the loop.
Loop control variable counter has value: 6
```

Note: In the above example, if *counter* is not incremented, i.e. remains with the value of 1, the test condition *(counter<=5)* will remain true and this will lead to an infinite loop.

Activity

- *Using a while loop starting with x = 0, display all the numbers up to 50 but skipping numbers 10, 25 and 35.*
- *Using a while loop, create a multiplication table of 4 with the first value being 4 and the last one being 100.*
- *Use a while loop to investigate the value of n such that product of*

  1 x 2 x 3 x 4 x … x n

  just crosses 1 million.
- *Save the updated script as **Activity 2_6_3_1***

2.6.3.2. The *Repeat* Loop

The Repeat Loop also executes a statement or a group of statements again and again until a stop condition is met which uses the *break* statement to terminate the loop. Figure 2.13 gives an illustration of the *Repeat* loop.

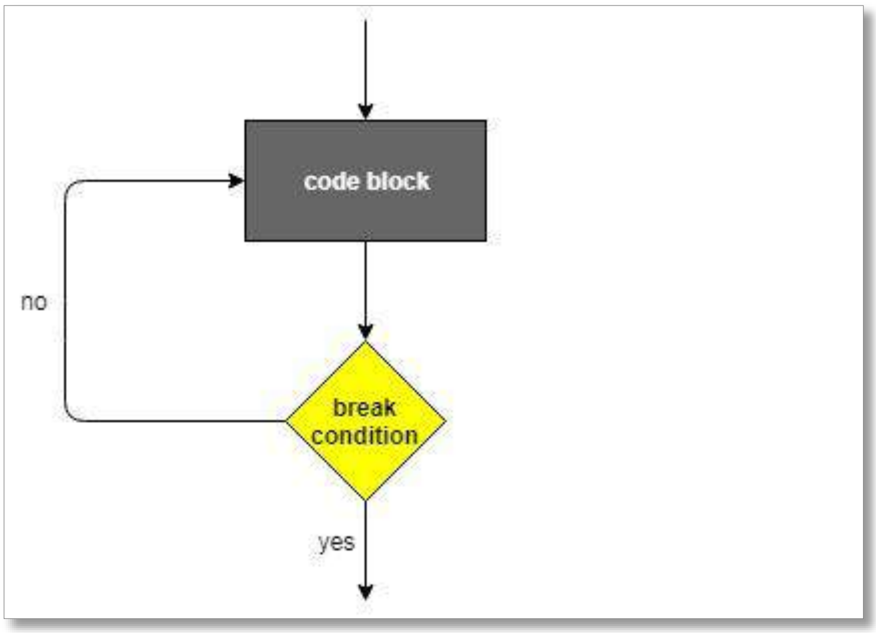


Figure 2.13 – The *Repeat* Loop

The general syntax for a Repeat loop is given below.

Syntax

```
repeat {
   statements…
   if(condition) {
      break
   }
}
```

In the following code listing, a *counter* variable is initialised to 10. A number of statements are iterated a number of time until *counter* reaches 0. The loop then terminates and the statements found after the loop are executed.

Example

```
# Initialise a loop control variable, counter, to 10
counter <- 10

# Repeat until counter is 0
repeat {
  cat(counter, "\n")

  # Decrement counter variable by 1
  counter = counter - 1

  # if counter is 0, terminate the loop
  if (counter==0){
    break
  }
}

cat("This is the end of the loop. \n")
```

The above code produces the following output.

Program Output

```
10
9
8
7
6
5
4
3
2
1
This is the end of the loop.
```

Activity

- *Using a repeat loop, print all the numbers ranging from 1 to 50.*
- *Using a repeat loop, print all the even numbers in the sequence 1 to 50.,*
- *Write a repeat loop that iterates over the numbers 1 to 10 and prints the cube of each number.*
- *Save the updated script as* ***Activity 2_6_3_2***

2.6.3.3. The *For* Loop

In R, *for* loops take an iterator variable and assign it to successive values from a sequence or vector. Unlike *While* and *Repeat* loops which repeat statement(s) based on conditions, *for* loops are most commonly used for iterating over the elements of an object, list, vector etc… Figure 2.14 gives an illustration of the *For* loop.

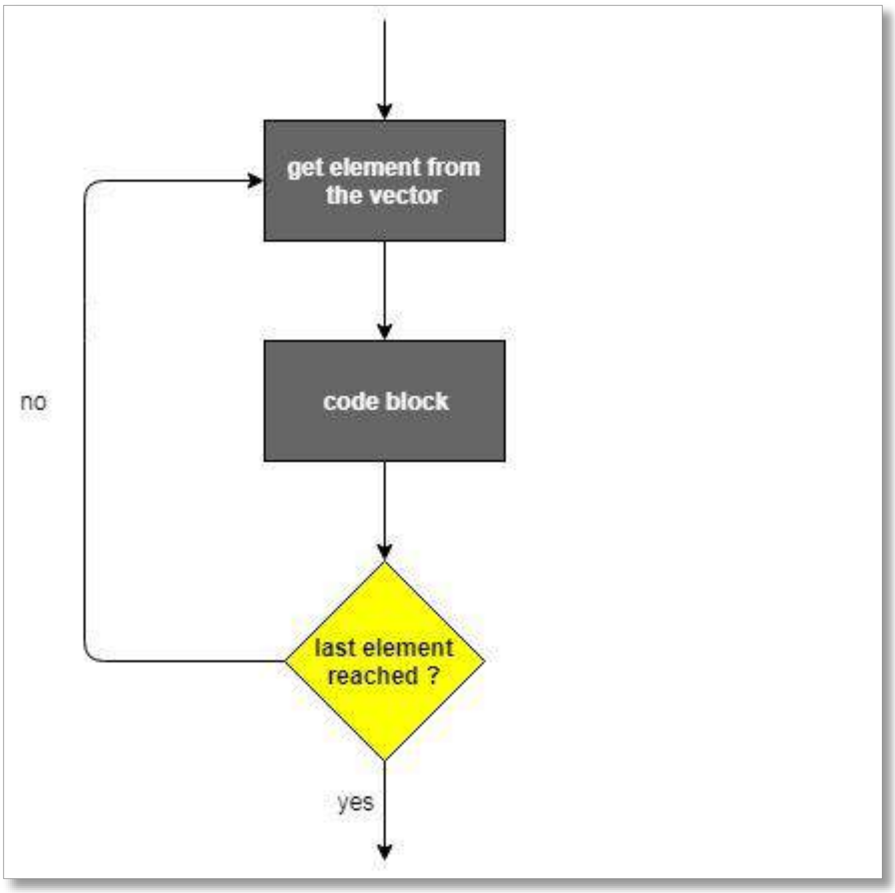


Figure 2.14 – The *For* Loop

The basic syntax for a *For* loop is given below.

Syntax

```
for (value in sequence) {
   statements
}
```

The following code listing consists of 3 examples of *for* loops. In the first example, the variable *i* is successively assigned to the values in the sequence 1 to 10 and is displayed in the loop. In the second example, 4 elements (letters) have been assigned to a vector *myVector1*. The loop control variable *j* is then assigned the values in the sequence 1 to 4 and is then used an index to retrieve the elements from the vector. The third example uses the built in constant *LETTERS* which stores the 26 upper-case letters of the Roman alphabet.

Example

```
# For Loop Example 1
cat("For Loop Example 1 \n")

# variable i will be successively assigned to values in the sequence 1 to
10
for(i in 1:10) {
  print(i)
}

# For Loop Example 2
```

```
cat("For Loop Example 2 \n")

# assign 4 elements to myVector1
myVector1 <- c("a", "b", "c", "d")

for(j in 1:4) {
  ## Print out each element of myVector1
  print(myVector1[j])
}

# For Loop Example 3
cat("For Loop Example 3 \n")

myVector2 <- LETTERS[5:8]
for (k in myVector2) {
  print(k)
}
```

The above code produces the following output.

Program Output

```
For Loop Example 1
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10

For Loop Example 2
[1] "a"
[1] "b"
[1] "c"
[1] "d"

For Loop Example 3
[1] "E"
[1] "F"
[1] "G"
[1] "H"
```

Activity

- *Write a for() loop that prints all the letters in a vector containing the following letters "q", "w", "e", "r", "z" and "c".*
- *Write a for() loop that prints the first five numbers of this vector: 7, 4, 3, 8, 9, 25, 10, 22 and 37*
- *Use a for() loop to re-implement the example in section 2.6.3.2 and consequently displays the same output.*
- *Save the updated script as **Activity 2_6_3_3***

2.7. Functions

A function is a group of statements that performs a specific task. Line in any other programming language, R has a number of in-built functions and, at the same time, it also allows the users to create their own functions, referred as *user defined* functions.

When there is a call to a function, the R interpreter passes control to the function together with arguments, if any, for the function to perform its specific actions. Consequently, the function performs its task and then returns the control, with the result if any, to the interpreter. The result may then be stored in other objects for further manipulation.

2.7.1. In-Built Functions

An in-built function is one which is already pre-defined in the programming language and which can be directly called in a program. Simple examples of in-built functions are *seq(), min(), max(), sum(), mean(),range(), round(), sqrt()* etc...

The following code listing shows examples of the use of the above in-built functions.

Example

```
# Create a sequence of numbers from 1 to 10.
print(seq(1,10))

# Print the minimum of a set of numbers.
print(min(2,10,5,8,4,3,9,7,6))

# Print the maximum of a set of numbers.
print(max(2,10,5,8,4,3,9,7,6))

# Find sum of the numbers from 1 to 10.
print(sum(1:10))

# Find mean of the numbers from 1 to 10.
print(mean(1:10))

# Print the range of values of a vector.
x <- c(1,4,8,6,2)
print(range(x))

# Print the square root of 8 and round it to 2 decimal places.
print(round(sqrt(8),2))
```

The above code produces the following output.

Program Output

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

[1] 2

[1] 10

[1] 55

[1] 5.5
```

```
[1] 1 8

[1] 2.83
```

2.7.2. User-Defined Functions

Programmers can also create their own functions. Once these have been created, they can be used in the same way as built-in functions. The basic syntax for creating a *Function* is given below.

Syntax

```
function_name <- function(arg_1, arg_2, ...) {
   statement(s)
}
```

2.7.2.1. Function Components

The different components of a function are as follows:

- Function Name: the actual name of the function
- Argument(s): The optional input(s) to a function
- Function Body: A set of statements that defines what the function does.
- Return Value: The optional result returned to the caller as the effect of calling the function.

2.7.2.2. User Defined Functions with no Argument

The inputs to a function are optional, i.e. a function may or may not have arguments. The following code listing gives 2 examples of how user defined functions, with no arguments, are created and used.

Example

```
# Creating a function Welcome which prints a statement
welcome <- function() {
  cat("Programming with functions is challenging!\n")
}

# Calling the function welcome without supplying any argument
welcome()

# Creating a square function without an argument
square <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}

# Call the square function
square()
```

The above code produces the following output.

Program Output

```
Programming with functions is challenging!

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

2.7.2.3. User Defined Functions with Arguments

A function may accept arguments. As mentioned earlier, an argument is an input that is supplied to and used by the function. While an argument is the actual value that is passed to a function, a parameter is a variable in a method definition. Therefore, when a function is called, the arguments are the data being passed into the method's parameters.

The following code listing gives 2 examples of how user defined functions, with arguments, are created and used. In the first example, the function accepts an argument which is a temperature in Celsius and calculates and displays its equivalent in Fahrenheit. Here, *temp_C* is the parameter while *25* is the argument. The second example shows a function that calculates the square of the numbers *1* till the number supplied as argument.

Example

```
# Creating a function with arguments
CelsiusToFahrenheit <- function(temp_C) {
  temp_F = temp_C * 9/5 + 32
  cat(temp_C, "in Celsius is", temp_F, "in Fahrenheit.\n")
}

# Calling the function with argument 25
CelsiusToFahrenheit(25)

# Creating a square function with an argument
square <- function(x) {
  for(i in 1:x)
    print(i^2)
}

# Call the square function
square(4)
```

The above code produces the following output.

Program Output

```
25 in Celsius is 77 in Fahrenheit.

[1] 1
[1] 4
[1] 9
[1] 16
```

2.7.2.4. Calling a Function with Argument Values (by position and by name)

The arguments in a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

The following code listing shows a user defined functions with 4 arguments: *a*, *b*, *c* and *d*. In the first function call, the arguments are supplied in sequence while in the second function call, the arguments are supplied by using the appropriate names. The output shows that both lead to the same result.

Example

```
# Create a function with 4 arguments.
Calculate <- function(a,b,c,d) {
  result <- a * b + c * d
  print(result)
}

# Call the function by position of arguments.
Calculate(2,5,4,3)

# Call the function by names of the arguments.
Calculate(d=3,b=5,c=4,a=2)
```

The above code produces the following output.

Program Output

```
Programming with functions is challenging!

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

2.7.2.5. Functions with Return statement

Line most programming languages, functions in R can return only a single object. However, this is not a limitation since a list containing several objects can also be returned. The keyword *return*() is generally used to return objects or values in a function. However,

The following code listing shows a user defined functions with 3 arguments: *num1*, *num2* and *num3*. The function calculates the greatest of the 3 arguments and returns this result which is then saved in a variable *largerNum* and then displayed.

Example

```
# Create a function, greater, that calculates maximum of 3 numbers.
greater <- function(num1,num2,num3) {

  if (num1>num2 & num1>num3)
    largest = num1
  else if (num2>num1 & num2>num3)
    largest = num2
  else
    largest = num3
  return (largest)
}

# Call the greater function and save its result
largerNum = greater(5,9,7)

# Print the result
print(largerNum)
```

The above code produces the following output.

Program Output

```
[1] 9
```

Note: In the absence of an explicit *return()* statement, the last expression evaluated in a function becomes the return value, i.e. the result of invoking the function.

Activity

- *Create a function that returns the difference between two numbers. The function should subtract the smaller number from the bigger one.*
- *Create a function that given an alpha numeric vector, it keeps only the numbers. For example, if the input is a vector w="b", "d", "8", "5", "q" , the function will return w= "8", "5".*
- *Create a function returns the grade of a student given his mark. The grading scheme is given in the table below:*

Mark	Award
Mark >= 80	A
Mark >= 60 & < 80	B
Mark >= 40 & < 60	C
Mark < 40	D

- *Write appropriate calls to test the above functions.*
- *Save your script as **Activity 2_7***

Unit Summary

Summary

In this unit you learned the fundamentals of the R programming language. You have used the Console window and the Script Editor to write your codes. The unit provides an overview of the arithmetic, relational and logical operators. You became familiar with the major R data structures and also worked with the two main control structures: decisions and loops. Finally the unit ends with built-in and user-defined functions.

References

1. Kabacoff, R.I., 2010. *R in Action*. Manning.
2. Grolemund, G. and Wickham, H., 2017. *R for Data Sience*. O’Reilly, January 2017 First Edition
3. Peng, R.D., 2015. *R programming for data science*. Lulu. com.
4. Venables, W. N., Smith D. M. and the R Core Team, 2017. *An Introduction to R - Notes on R: A Programming Environment for Data Analysis and Graphics.* [ONLINE] Available at: https://cran.r-project.org/doc/manuals/R-intro.html [Accessed 30 April 2018]
5. Tutorialspoint. 2018. *R Tutorial*. [ONLINE] Available at: https://www.tutorialspoint.com/r/index.htm. [Accessed 31 March 2018].
6. R Exercises. 2018. *Homepage*. [ONLINE] Available at: https://www.r-exercises.com/. [Accessed 30 April 2018].

Further Reading

1. Zumel, N., Mount, J. and Porzak, J., 2014. *Practical data science with R* (pp. 101-104). Manning.
2. Matloff, N., 2011. *The art of R programming: A tour of statistical software design*. No Starch Press.
3. Teetor, P., 2011. *R Cookbook: Proven recipes for data analysis, statistics, and graphics*. "O'Reilly Media, Inc.".
4. R Exercises. 2018. *Homepage*. [ONLINE] Available at: https://www.r-exercises.com/. [Accessed 30 April 2018].
5. Edureka, 2017. *R Tutorial For Beginners*. [Online video] Available at: https://www.youtube.com/watch?v=eDrhZb2onWY [Accessed 30 April 2018].].