CHAPTER 1

INTRODUCTION OF THE LANGUAGE

Syllabus

Numeric, arithmetic, assignment, and vectors, Matrices and Arrays, Non-numeric Values, Lists and Data Frames, Special Values, Classes, and Coercion, Basic Plotting.

1.1 Introduction of the language

Brief history of R language:

R is based on the S language, first developed in the 1960s and 1970s by researchers at Bell Laboratories in New Jersey. With a view to embracing open-source software, R's developers—Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand—released it in the early 1990s under the GNU public license. Since then, the popularity of R has grown because of its unrivaled flexibility for data analysis and powerful graphical tools, all available for the princely sum of nothing. Most appealing feature of R is that any researcher can contribute code in the form of packages or libraries, so the rest of the world can have fast access to developments in statistics and data science. Today, the main source code archives are maintained by a dedicated group known as the R Core Team, and R is a collaborative effort.

R Programming Language:

R is an object oriented, high-level, and interpreted programming language, designed for statistical computing and graphics support. It is strictly case and character sensitive, which means that you enter instructions that follow the specific syntactic rules of the language into a console or command-line interface. The software then interprets and executes your code and returns any results. R is an object-oriented programming language, that means entities are stored as objects and have methods that act upon them. In such a language, class identification is referred to as inheritance. R programming is one of the most used languages used in data mining.

Basic features of R programming:

- R is an open-source tool
- R is also a cross-platform compatible language
- R is a great visualization tool
- R is a most common tool for data mining
- R is used for data science and machine learning tasks.

R Software:

R is an open source programming language, so we can download R software from https://cran.r-project.org/bin/windows/base/ for windows operating system, currently R 4.3.1 for windows, is the latest version available which is used in this textbook for all examples. You can also get R for macOS or Linux OS from the same website CRAN https://cran.r-project.org. After installing R we can also install R IDE that is RStudio to make our task easier. We can download RStudio from www.rstudio.com.

After installing R, the compiler window will be like this in Figure 1.1, where we can execute basic R scripts.

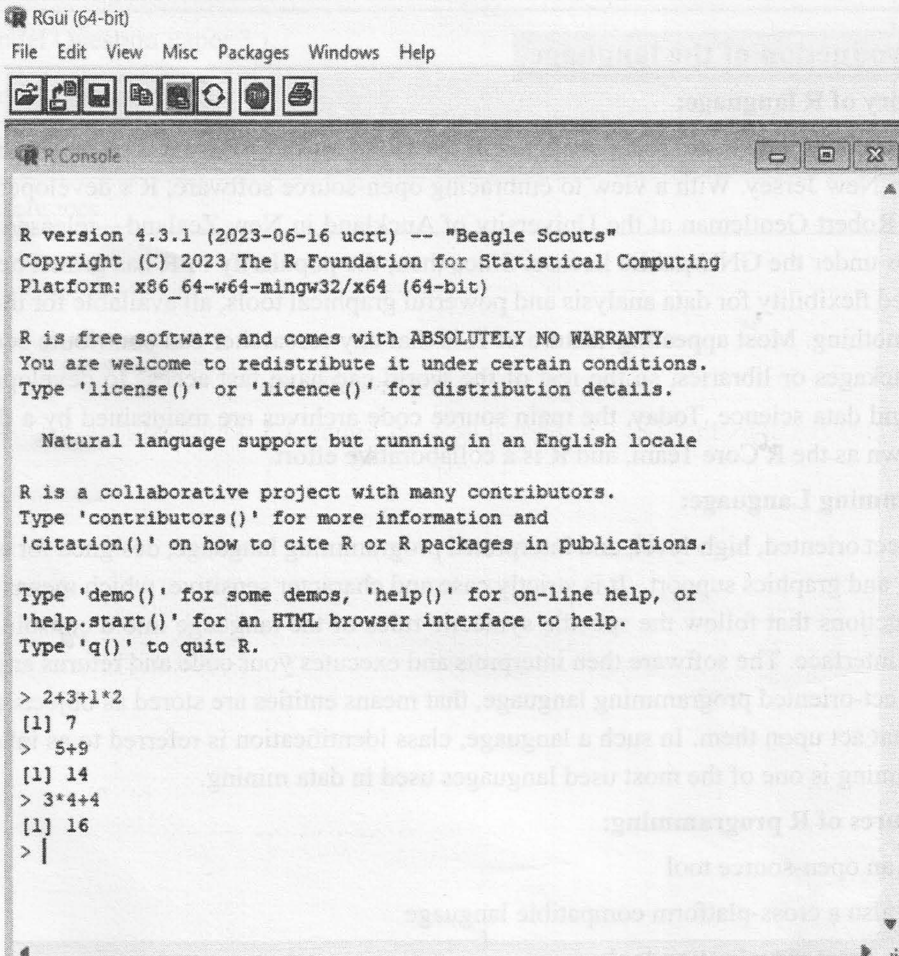


Figure 1.1: Working with RGui platform(64 bits)

RStudio download window will look like this, from where we can select Rstudio for windows. While installing RStudio, choose the existing R 64 bit to make use of the R in the RStudio.

1: Install R

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

DOWNLOAD AND INSTALL R

2: Install RStudio

DOWNLOAD RSTUDIO DESKTOP FOR WINDOWS

Size: 214.34 MB | SHA-256: FE628784 | Version: 2023.09.1+494 |
Released: 2023-10-17

All Installers and Tarballs

RStudio requires a 64-bit operating system.

Linux users may need to import Posit's public code-signing key prior to installation, depending on the operating system's security policy.

OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2023.09.1-494.EXE	214.34 MB	FE628784

Figure 1.2: RStudio Installation

After installing RStudio, the IDE will be visible as in Figure 1.3 with 3 windows(Console, Environment and Plot).

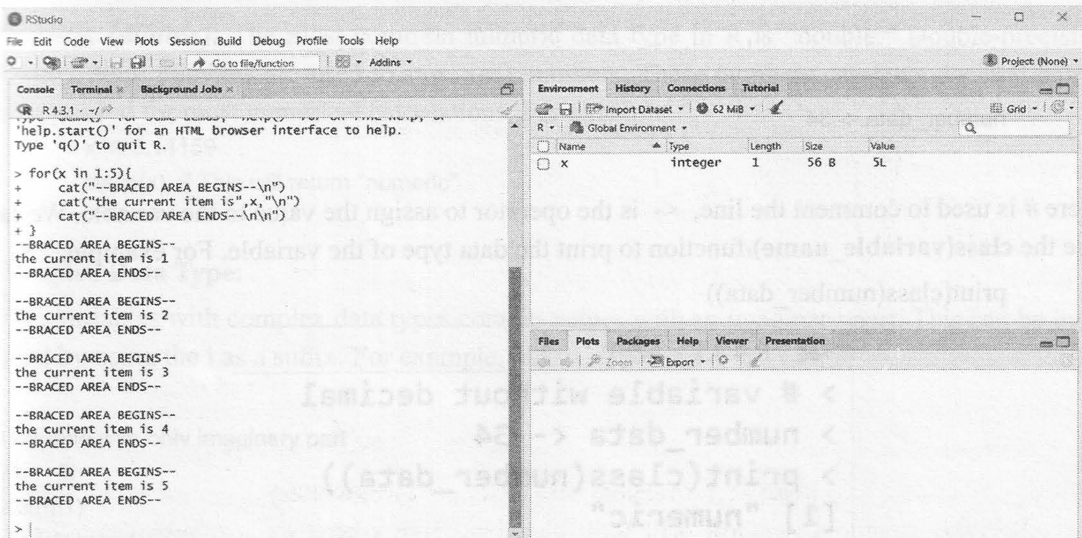


Figure 1.3: RStudio, an IDE for R

Once the RGui and RStudio are installed your platform is ready, taking R commands or programs, as we can see in Figure 1.3 a for loop code with cat statement to repeat the statements 5 times. Basic R programming can be executed in just RGui without RStudio as you can see in the Figure 1.1.

1.2 Numeric:

Numbers in R can be divided into 3 different categories:

- Numeric(Double): It represents both whole and floating-point numbers.

  For example, 124, 32.42, 33.23 etc.
- Integer: It represents only whole numbers and is denoted by L.

  For example, 25L, 38L, 40L etc.
- Complex: It represents complex numbers with imaginary parts. The imaginary parts are denoted by i.

  For example, 2 + 3i, 9i, 6+8i etc.

Numeric Data Type:

Numeric is the most frequently used data type in R. It is the default data type whenever we declare a variable with numbers. We can store any type of number (with or without decimal) in a variable with numeric data type. There are several numeric data types to represent different kinds of numerical values. For example,

```
# decimal variable
decimal_data <- 123.45


# variable without decimal
number_data <- 34
```

Here # is used to comment the line, <- is the operator to assign the value to the variable. We can use the **class(variable_name)** function to print the data type of the variable. For example,

```
print(class(number_data))
```

```
R 4.3.1 · ~/
> # variable without decimal
> number_data <- 34
> print(class(number_data))
[1] "numeric"
>
```

Integer Data Type:

Integers are a type of numeric data that can take values without decimal. It's mostly used when you are sure that the variable can not have any decimal values in the future.

In order to create an integer variable, you must use the suffix L at the end of the value. For example,

```
integer_data <- 123L

# print the value of my_integer
print(integer_data)

# print the data type of integer_data
print(class(integer_data))
```

```
R 4.3.1 · ~/
> integer_data <- 123L
> # print the value of my_integer
> print(integer_data)
[1] 123
> # print the data type of integer_data
> print(class(integer_data))
[1] "integer"
>
```

Numeric (double): The most common numeric data type in R is "double." Double-precision floating-point numbers are used to store real numbers. These numbers can have decimal points and are used for most numerical calculations. For example:

```
x <- 3.14159
class(x)  # This will return "numeric"
```

Complex Data Type:

In R, variables with complex data types contain values with an imaginary part. This can be indicated by using the i as a suffix. For example,

```
# variable with only imaginary part
m1 <- 5i
print(m1)
print(class(m1))
```

```
# variable with both real and imaginary parts
m2 <- 3 + 3i
print(m2)
print(class(m2))
```

```
R 4.3.1 · ~/
> # variable with only imaginary part
> m1 <- 5i
> print(m1)
[1] 0+5i
> print(class(m1))
[1] "complex"
> # variable with both real and imaginary parts
> m2 <- 3 + 3i
> print(m2)
[1] 3+3i
> print(class(m2))
[1] "complex"
>
```

Logical: Logical data types, represented by "TRUE" and "FALSE" (or "T" and "F" for short), are used for boolean values. While they are not strictly numeric, they can be used in numeric calculations where "TRUE" is treated as 1 and "FALSE" is treated as 0.

Raw: The "raw" data type is used to store raw bytes and is not commonly used for typical numerical calculations. It is mainly used for low-level programming.

It's important to note that R is a dynamically typed language, so you don't need to specify the data type explicitly when creating variables. R will automatically infer the data type based on the value assigned to the variable.

Here's an example of dynamic typing in R:

```
a <- 42.5      # "a" is automatically assigned the "numeric" data type
b <- 7L        # "b" is automatically assigned the "integer" data type
c <- TRUE      # "c" is automatically assigned the "logical" data type
```

Operators in R programming:

Operators are used to perform operations on variables and values. In the example below, we use the + operator to add together two values: 20 + 30. R divides the operators in the following groups:

- Arithmetic operators : + - * / ^ % % % /%
- Assignment operators : <- <<- -> ->>
- Comparison operators : == != > < >= <=
- Logical operators : & && | ||!
- Miscellaneous operators : : %in% %*%

1.3 Arithmetic operators in R: Arithmetic operators are used with numeric variables or values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
^	Exponent	x ^ y
%%	Modulus (Remainder from division)	x %% y
%/%	Integer Division	x %/% y

1.4 Assignment operators in R:

Assignment operators are used to assign values to variables:

```
X <- 3     # assigning 3 to variable X
X <<- 3    # global variable X getting assigned as 3
3 -> X     # assigning 3 to variable X
3 ->> X    # global variable X getting assigned as 3
X          # print the value of X variable
```

```
> X <- 3 # assigning 3 to variable X
> X <<- 3 # global variable X getting assigned as 3
> 3 -> X # assigning 3 to variable X
> 3 ->> X # global variable X getting assigned as 3
> X # print the value of X variable
[1] 3
>
```

Note:

<<- is a global assigner. Variables that are created outside of a function are known as global variables. Global variables can be used by everyone, both inside of functions and outside. It is also possible to turn the direction of the assignment operator. For example - x <- 3 is the same as 3 -> x .

Comparison Operators in R : Comparison operators are used to compare two variables or values, it is same as in C or Java Programming:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators in R: Logical operators are used to combine conditional statements, it is also same as in C or Java programming:

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>&</td><td>Element-wise Logical AND operator. It returns TRUE if both elements are TRUE</td></tr>
<tr><td>&&</td><td>Logical AND operator - Returns TRUE if both statements are TRUE</td></tr>
<tr><td>|</td><td>Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE</td></tr>
<tr><td>||</td><td>Logical OR operator. It returns TRUE if one of the statements is TRUE.</td></tr>
<tr><td>!</td><td>Logical OR operator. It returns TRUE if one of the statements is TRUE.</td></tr>
</table>

1.5 Vectors:

Vectors in R are the same as the arrays in C/Java language which are used to hold multiple data values of the same type. One major difference is that in R the indexing of the vector will start from '1' and not from '0'. We can create numeric vectors and character vectors as well. To create a vector we can use the **c( )** function to combine the list of items to a vector, and separate the items by a comma and then we can assign it to a variable. For example- **V1 <- c(10,20,30,40,50,60, 70,80,90,100)**

Index →	1	2	3	4	5	6	7	8	9	10
Values →	10	20	30	40	50	60	70	80	90	100

Types of R vector: A vector in R can be numeric, character or logical.

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

For example-

```
R 4.3.1 · ~/
> # Creation of Vectors using C() function.
> v1<-c(4,5,6,7)
> #display types of vector
> typeof(v1)
[1] "double"
> #By using 'L' we can specify that we want integer values.
> v2<-c(1L, 4L, 2L, 5L)
> #display types of vector
> typeof(v2)
[1] "integer"
> v1
[1] 4 5 6 7
> v2
[1] 1 4 2 5
>
```

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

For Example -

```
R 4.3.1 · ~/
> # R program to create character vectors
> #by default numeric values are converted into characters
> v1 <- c('ab','2','cd',4, 'ef',6)
> # displaying data type of the vector
> typeof(v1)
[1] "character"
>
> v1   # displaying vector
[1] "ab"  "2"   "cd"  "4"   "ef"  "6"
>
```

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

For Example

```
R 4.3.1 · ~/
> #R program to create logical vectors
> # creating logical vector using C() function
>
> v1 <- c(TRUE,FALSE,TRUE,NA)
> #displaying the data type of vector v1
> typeof(v1)
[1] "logical"
> v1  # displaying the value of v1
[1]  TRUE FALSE  TRUE      NA
> |
```

Different ways of creating vectors in R and getting its length.

```
R 4.3.1 · ~/
> X <- c(61,4,21,67,89,22)
> cat('using c function', X, '\n')
using c function 61 4 21 67 89 22
>
> print(X)
[1] 61  4 21 67 89 22
> # using ':' to create a vector of continuous values.
> Z <- 2:17
> cat('using colon',Z)
using colon 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
>
> #using seq() function and length.out to defines the length of vector
> Y <- seq(1,10, length.out=5)
> cat('using seq() function', Y, '\n')
using seq() function 1 3.25 5.5 7.75 10
> length(Y)
[1] 5
> |
```

Accessing a vector using its index starting from 1.

```
R 4.3.1 · ~/
> # accessing elements with an index number.
> X <- c(2,5,18,1,12)
> cat('using subscript operator', X[2], '\n')
using subscript operator 5
>
> #by passing a range of values inside the vector index
> Y <-c(4,8,2,1,17)
> cat('using combine() function', Y[c(4,1)], '\n')
using combine() function 1 4
> |
```

1.6 Matrices:

A matrix is a two dimensional data set with columns and rows. A column is a vertical representation of data, while a row is a horizontal representation of data. A matrix can be created with the **matrix()** function. Specify the **nrow** and **ncol** parameters to get the amount of rows and columns, like vector in matrix also **c ( )** function is used to concatenate the items together.

Matrices are particularly useful for mathematical and statistical operations, and they are an essential part of data analysis and manipulation. You can create and work with matrices in R using various functions and operations. Here's how to work with matrices in R:

I. Creating a Matrix:

You can create a matrix in R using the matrix() function. Here's the basic syntax:

matrix(data, nrow = number_of_rows, ncol = number_of_columns, byrow = FALSE)

- **data:** A vector of data elements that will be filled into the matrix.
- **nrow:** The number of rows in the matrix.
- **ncol:** The number of columns in the matrix.
- **byrow:** If byrow is set to TRUE, the data is filled row-wise; if set to FALSE, it's filled column-wise (the default).

Example:

```
# Create a 2x3 matrix filled column-wise
mat <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
```

II. Accessing Matrix Elements:

You can access elements in a matrix using square brackets and specifying the row and column indices. Remember that indexing in R starts from 1.

Example:

```
# Access the element in the first row and second column
element <- mat[1, 2]  # element will be 2
```

Other example-

```
# Create a matrix with 3 rows and 2 columns
> mat <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)
# Print the matrix
> mat
```

```
R 4.3.1 · ~/
> # Create a matrix with 3 rows and 2 columns
> mat <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)
> # Print the matrix
> mat
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
```

Creating a matrix with strings:

```
strmat <- matrix(c("bangalore", "chennai", "delhi", "mumbai"), nrow = 2, ncol = 2)
```

```
R 4.3.1 · ~/
> strmat <- matrix(c("bangalore", "chennai", "delhi", "mumbai"), nrow = 2, ncol = 2)
> strmat
     [,1]        [,2]
[1,] "bangalore" "delhi"
[2,] "chennai"   "mumbai"
>
```

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

For example -

```
R 4.3.1 · ~/
> strmat <- matrix(c("bangalore", "chennai", "delhi", "mumbai"), nrow = 2, ncol = 2)
> strmat
     [,1]        [,2]
[1,] "bangalore" "delhi"
[2,] "chennai"   "mumbai"
> strmat[1,2]
[1] "delhi"
>
```

III. Matrix Operations:

You can perform various operations on matrices, such as addition, subtraction, multiplication, and transposition. Here's how you can do some basic matrix operations:

- Matrix Addition and Subtraction:

```
mat1 <- matrix(c(1, 2, 3, 4), nrow = 2)
mat2 <- matrix(c(5, 6, 7, 8), nrow = 2)
result_add <- mat1 + mat2
result_sub <- mat1 - mat2
```

- Matrix Multiplication:

```
mat1 <- matrix(c(1, 2, 3, 4), nrow = 2)
mat2 <- matrix(c(5, 6, 7, 8), nrow = 2)
result_mul <- mat1 %*% mat2
```

- Matrix Transposition:

```
mat <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE)
transposed_mat <- t(mat)
```

IV. Matrix Functions:

R provides several functions for working with matrices, such as dim(), rownames(), and colnames() to get matrix dimensions and row/column names, diag() to create a diagonal matrix, and solve() to find the inverse of a matrix.

Example:

```
mat <- matrix(c(1, 2, 3, 4), nrow = 2)
dimensions <- dim(mat) # dimensions will be 2x2
```

Matrices are a fundamental data structure in R for working with tabular data and performing various mathematical and statistical operations. They are commonly used in linear algebra, statistics, and data analysis.

1.7 Arrays:

Compared to matrices, arrays can have more than two dimensions. We can use the array() function to create an array, and the dim parameter to specify the dimensions of an array. It is an essential data storage structure defined by a fixed number of dimensions. Arrays are used for the allocation of space at contiguous memory locations. One-dimensional arrays are called vectors with the length being their only dimension. Two-dimensional arrays are called matrices, consisting of fixed numbers of rows and columns. Arrays consist of all elements of the same data type.

I. Creating an Array:

You can create an array in R using the array() function. The basic syntax is as follows:

```
array(data, dim = (nrow, ncol, nmat), dimnames=names) OR
```

```
array(data, dim = c(dim1, dim2, ...))
```

- **data:** A vector of data elements that will be used to populate the array.
- **dim:** A vector specifying the dimensions of the array, including the number of rows and columns for each dimension.

Example:

```
# Create a 3-dimensional array
arr <- array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim = c(2, 3, 2))
```

II. Accessing Array Elements:

You can access elements in an array using square brackets and specifying the indices for each dimension. Array indices in R start from 1 for each dimension.

Example:

```
# Access an element in the array
element <- arr[1, 2, 1] # element will be 2
```

III. Array Operations:

You can perform various operations on arrays, such as addition, subtraction, multiplication, and more. The operations are performed element-wise, just like with matrices.

Example:

```
arr1 <- array(1:12, dim = c(2, 3, 2))
arr2 <- array(13:24, dim = c(2, 3, 2))
result_add <- arr1 + arr2
result_sub <- arr1 - arr2
```

IV. Array Functions:

R provides functions for working with arrays, such as dim(), dimnames(), and array(). You can use these functions to get the dimensions and dimension names of an array.

Example:

```
dimensions <- dim(arr1) # dimensions will be c(2, 3, 2)
```

V. Higher-Dimensional Arrays: Arrays in R can have more than three dimensions, making them suitable for complex data structures. You can access and manipulate elements in arrays with as many dimensions as needed.

VI. Specialized Arrays: In addition to regular arrays, R also provides specialized types of arrays like data frames (2D arrays with column names) and tensors (n-dimensional arrays used in deep learning libraries like TensorFlow and Keras).

Arrays are versatile data structures in R that are particularly useful for storing and manipulating multidimensional data. They are commonly used in scientific computing, image processing, and working with data in a structured way, especially when data has more than two dimensions.

One-Dimensional Array: A vector is a one-dimensional array, which is specified by a single dimension. A Vector can be created using the 'c()' function. A list of values is passed to the c() function to create a vector.

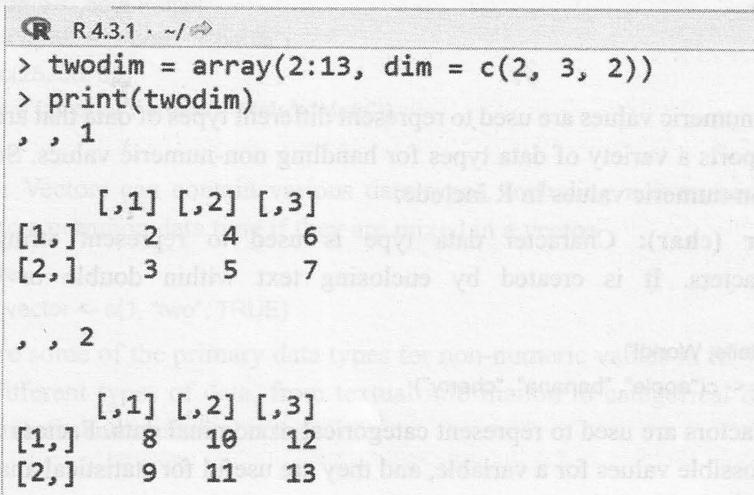
For example:

```
> list1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
> print (list1)
[1] 1 2 3 4 5 6 7 8 9
```

Multi-Dimensional Array: A two-dimensional matrix is an array specified by a fixed number of rows and columns, each containing the same data type. A matrix is created by using array() function to which the values and the dimensions are passed.

For example-

```
twodim = array(2:13, dim = c(2, 3, 2))
print(twodim)
```



```
R 4.3.1 · ~/
> twodim = array(2:13, dim = c(2, 3, 2))
> print(twodim)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    8   10   12
[2,]    9   11   13
```

We can access the arrays by using its name and row number, column number with matrix number.

For example-

print ("2nd row 3rd column of the matrix1 ")

print (twodim[2, 3, 1])

```
> print ("2nd row 3rd column of the matrix1 ")
[1] "2nd row 3rd column of the matrix1 "
> print (twodim[2, 3, 1])
[1] 7
>
```

1.8 Non-numeric Values:

Non-numeric values can be identified from the vector in R using **is.na( ) & which( )** function. In the following example ***which(is.na(as.numeric(vector_name)))*** function returns the index numbers of the non-numeric values as 4th and 6th values "AB" & "5,6" are non-numeric.

For example-

```
Console  Terminal ×  Background Jobs ×
R 4.3.1 · ~/
> x <- c(1, 2, 3, "ABC", 4, "5,6")  # creating a vector
> x                                 # printing vector
[1] "1"    "2"    "3"    "ABC" "4"    "5,6"
>
> x_nonum <- which(is.na(as.numeric(x)))  # get indices of non-numeric
Warning message:
In which(is.na(as.numeric(x))) : NAs introduced by coercion
> x_nonum
[1] 4 6
>
```

In R, non-numeric values are used to represent different types of data that are not numerical in nature. R supports a variety of data types for handling non-numeric values. Some of the key data types for non-numeric values in R include:

Character (char): Character data type is used to represent strings of text or individual characters. It is created by enclosing text within double or single quotes. Example:

```
text <- "Hello, World!"
char_vec <- c("apple", "banana", "cherry")
```

Factor: Factors are used to represent categorical or nominal data. Factors can have levels that define the possible values for a variable, and they are useful for statistical analysis.

Example:

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

Logical (logical): The logical data type represents binary values, which can be either TRUE or FALSE. It is often used for logical and boolean operations.

Example:

```
is_student <- TRUE
is_adult <- FALSE
```

Date and Time: R provides various data types for date and time representation, such as Date, POSIXct, and POSIXlt.

Example:

```
current_date <- Sys.Date()
current_time <- Sys.time()
```

Character Vector (character): Character vectors are collections of character values, often used to store text data.

Example:

```
names <- c("Alice", "Bob", "Charlie")
```

Lists: Lists are versatile data structures that can store mixed types of data, including numeric and non-numeric values. They are useful for grouping different types of data into a single object.

Example:

```
my_list <- list(name = "John", age = 30, is_student = TRUE)
```

Data Frames: Data frames are similar to data tables in a relational database. They can store tabular data with columns of different data types, including character, numeric, and factors. Data frames are widely used in data analysis and statistics.

Example:

```
data_frame <- data.frame(
Name = c("Alice", "Bob", "Charlie"),
Age = c(25, 30, 35),
Gender = factor(c("Female", "Male", "Male"))
)
```

Vectors: Vectors can contain various data types, including non-numeric values. R will coerce values to a common data type if they are mixed in a vector.

Example:

```
mixed_vector <- c(1, "two", TRUE)
```

These are some of the primary data types for non-numeric values in R. They are essential for handling different types of data, from textual information to categorical data and more, in various data analysis and statistical tasks.

1.9 Lists:

A list in R can contain many different data types inside it. A list is a collection of data which is ordered and changeable. To create a list, use the list() function. R allows accessing elements of an R list using the index value. In R, the indexing of a list starts with 1 instead of 0 like in other programming languages.

A list in R programming is a versatile data structure that can store a collection of objects, which may be of different data types. Lists are often used to group together and organize various types of data or objects. Lists are defined by enclosing the elements within square brackets [ ] and separated by commas. Here's how to create and work with lists in R:

I. Creating a List:

You can create a list using the list() function or by using the square brackets []. Here are examples of both methods:

Using the list() function:

```
my_list <- list(item1, item2, item3, ...)
```

For example:

```
# List of strings
countrylist <- list("India", "US", "UK")
# Print the list
countrylist
```

```
R 4.3.1 · ~/
> # List of strings
> countrylist <- list("India", "USA", "UK")
> # Print the list
> countrylist
[[1]]
[1] "India"

[[2]]
[1] "USA"

[[3]]
[1] "UK"
```

Each item in a list can be of any data type, including numbers, text, vectors, matrices, data frames, and even other lists.

Example:

```
# Creating a list using the list() function
my_list <- list(
  name = "Alice",
  age = 30,
  is_student = TRUE,
  grades = c(95, 88, 75),
  contact = data.frame(email = "alice@example.com", phone = "123-456-7890")
  )
```

II. Accessing List Elements:

You can access individual elements of a list using double square brackets [[ ]] or by using the dollar sign $. The double brackets extract the actual element, while the dollar sign extracts an element by name.

```
# Accessing list elements
name <- my_list$name
grades <- my_list[["grades"]]
```

III. Adding Elements to a List:

You can add new elements to a list using the assignment operator <-.

```
my_list$new_element <- "New data"
my_list[["new_element2"]] <- "More data"
```

IV. Removing Elements from a List:

To remove elements from a list, you can use the NULL value or the rm() function.

```
my_list$new_element <- NULL
rm(my_list$new_element2)
```

V. List Functions:

R provides various functions to work with lists, such as length(), names(), and str() to get the number of elements in a list, retrieve the names of list elements, and inspect the structure of the list, respectively.

```
list_length <- length(my_list)
list_names <- names(my_list)
str(my_list)
```

VI. Joining two lists: There are several ways to join, or concatenate, two or more lists in R. The most common way is to use the **c()** function, which combines two elements together:

```
R 4.3.1 · ~/
> list1 <- list("A", "B", "C")
> list2 <- list(1, 2, 3)
> list3 <- c(list1, list2)  # adding two list
> list3
[[1]]
[1] "A"

[[2]]
[1] "B"

[[3]]
[1] "C"

[[4]]
[1] 1

[[5]]
[1] 2

[[6]]
[1] 3
```

VII. Creating a list by naming all its components:

```
empId = c(1, 2, 3, 4)
```

```
empName = c(„Amit“, „Bali“, „Uma“, „SK“)
numberOfEmp = 4
empList = list(
                    "ID" = empId,
                    "Names" = empName,
                    "Total Staff" = numberOfEmp
)
print(empList)

# Accessing a top level components by indices
cat("Accessing name components using indices\n")
print(empList[[2]])

# Accessing a inner level components by indices
cat("Accessing Bali from name using indices\n")
print(empList[[2]][2])

# Accessing another inner level components by indices
cat("Accessing 4 from ID using indices\n")
print(empList[[1]][4])
```

```
R 4.3.1 · ~/
> #Creating a list by naming all its components:
>     empId = c(1, 2, 3, 4)
>     empName = c("AK", "SK", "Uma", "Pooja")
>     numberOfEmp = 4
>     empList = list(
+         "ID" = empId,
+         "Names" = empName,
+         "Total Staff" = numberOfEmp
+     )
>     print(empList)
$ID
[1] 1 2 3 4

$Names
[1] "AK"    "SK"    "Uma"    "Pooja"

$`Total Staff`
[1] 4
```

```
> # Accessing a top level components by indices
> cat("Accessing name components using indices\n")
Accessing name components using indices
> print(empList[[2]])
[1] "AK"      "SK"      "Uma"     "Pooja"
> # Accessing a inner level components by indices
> cat("Accessing Bali from name using indices\n")
Accessing Bali from name using indices
> print(empList[[2]][2])
[1] "SK"
> # Accessing another inner level components by indices
> cat("Accessing 4 from ID using indices\n")
Accessing 4 from ID using indices
> print(empList[[1]][4])
[1] 4
> |
```

Lists are commonly used in R for organizing and storing diverse types of data. They are particularly useful when you need to keep different types of objects together, such as a mix of numeric data, text, and more, in a single data structure.

1.10 Data Frames:

Data Frames in R Language are generic data objects of R, that are used to store tabular data. Data frames can also be matrices where each column of a matrix can be of different data types. Data-Frame in R is made up of three principal components, the **data, rows,** and **columns.**

Data frame in R is a two-dimensional data structure that is used to store tabular data in a structured format. Data frames are similar to tables in a relational database or spreadsheets, and they are a fundamental data structure for data analysis and manipulation in R. Each column of a data frame can be of a different data type, but all columns must have the same length. Here's how to work with data frames in R:

I. **Creating a data frame in R:** To create a data frame we can use data.frame() command, and then pass each of the vectors we have created as arguments to the function.

Syntax:

```
data_frame_name <- data.frame(column1, column2, column3, ...)
```

Example 1:

```
# Creating a data frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Gender = factor(c("Female", "Male", "Male"))
)
```

Example 2:

```
friends <- data.frame(
    friends_id = c(1:5),
    friends_name = c("Virat", "Jaddu","Rohit", "Shami","Bumra"),
stringsAsFactors = FALSE
)
# printing the data frame
print(friends)
```

Output:

```
R 4.3.1 · ~/
> friends <- data.frame(
+     friends_id = c(1:5),
+     friends_name = c("Virat", "Jaddu","Rohit", "Shami","Bumra"),
+     stringsAsFactors = FALSE
+ )
> # printing the data frame
> print(friends)
  friends_id friends_name
1          1        Virat
2          2        Jaddu
3          3        Rohit
4          4        Shami
5          5        Bumra
>
```

Getting data from an R data frame: We can access its rows or columns. One can extract a specific column from an R data frame using its column name.

For example-

```
result <- data.frame(friends $friends_name)
print(result)
```

```
R 4.3.1 · ~/
> result <- data.frame(friends $friends_name)
> print(result)
  friends.friends_name
1                Virat
2                Jaddu
3                Rohit
4                Shami
5                Bumra
>
```

II. Accessing Data Frame Elements:

You can access individual columns of a data frame using the $ operator or by using square brackets []. The $ operator extracts a column by name, while square brackets allow you to select one or more columns.

```
# Acc essing columns of a data frame
names_column <- df$Name
age_column <- df[["Age"]]
```

III. Viewing Data Frames:

You can view the contents of a data frame by simply typing its name, and R will display the table-like structure in the console. To view the first few rows, you can use the head() function.

```
# View the entire data frame
    df
# View the first few rows
    head(df)
```

IV. Adding and Removing Columns:

You can add new columns to a data frame using the assignment operator <-. To remove columns, you can use the NULL value.

```
# Adding a new column
    df$City <- c("New York", "Los Angeles", "Chicago")
# Removing a column
    df$City <- NULL
```

Removing Rows and Columns from R data frame: We can remove rows and columns from a data frame in R from the already existing data frame.

Example:

```
# Create a data frame
 data <- data.frame(
  friend_id = c(1, 2, 3, 4, 5),
 friend_name = c("Virat","Jaddu","Rohit","Shami","Bumra"),
 location = c("Bangalore", "Chennai", "Mumbai", "Gujrat", "Mumbai")
 )
 # Remove a row with friend_id = 3
 data <- subset(data, friend_id != 3)
```

```
R 4.3.1 · ~/
> # Create a data frame
> data <- data.frame(
+     friend_id = c(1, 2, 3, 4, 5),
+     friend_name = c("Virat", "Jaddu","Rohit", "Shami","Bumra"),
+     location = c("Bangalore", "Chennai", "Mumbai", "Gujrat", "Mumbai")
+ )
> # Remove a row with friend_id = 3
>
> data <- subset(data, friend_id != 3)
> data  #printing data
  friend_id friend_name  location
1         1       Virat Bangalore
2         2       Jaddu   Chennai
4         4       Shami    Gujrat
5         5       Bumra    Mumbai
> |
```

V. Subset and Filter Data Frames:

You can create subsets of a data frame using logical conditions. For example, to filter rows where the age is greater than 30:

```
subset_df <- df[df$Age > 30, ]
```

VI. Data Frame Functions:

R provides various functions for working with data frames, such as dim() to get the dimensions of a data frame, names() to retrieve column names, and str() to inspect the structure of the data frame.

```
df_dimensions <- dim(df)
column_names <- names(df)
str(df)
```

Data frames are essential for data manipulation, analysis, and visualization in R. They are widely used in tasks such as data cleaning, exploration, and modeling. Data frames provide a structured way to work with real-world data, where variables may have different data types and missing values.

1.11 Special Values:

There are four special values in R which are used to check the data in any existing data set. These 4 special values are : NA, NULL, NaN and Inf.

NA is used to represent a missing value in the R data set. NULL is to represent an empty data, similar to NA, NaN is to represent a not a number and Inf is representing infinite value. These special values play various roles in data analysis, data manipulation, and programming in R.

To identify these data from the data set R has respective functions for these special values as given below:

Summary	NA	NULL	NaN	Inf
class()	"logical"	"NULL"	"numeric"	"numeric"
length()	1	0	1	1
check	is.na()	is.null()	is.nan()	is.finite()

Example:

```
> a <- NaN
> is.nan(a)    #the correct way to check if the value is NaN
```

```
> a <- NaN
> is.nan(a) #the correct way to check if the value is NaN
[1] TRUE
> length(a)
[1] 1
>
```

NA and NaN:

- NA represents missing or undefined values. It is used to indicate the absence of data for a particular observation or variable.
- NaN (Not-a-Number) is used to represent undefined or unrepresentable numeric values, typically resulting from mathematical operations that don't yield a meaningful result.

NULL: NULL is used to indicate the absence of an object. It is often used in function arguments or as a placeholder when you want to create an empty **data structure.**

Inf and -Inf: Inf represents positive infinity, while -Inf represents negative infinity. These are often used in numeric calculations, particularly for values that exceed the range of finite numbers.

NA_integer_, NA_real_, NA_character_: These special values are used to represent missing values of specific data types (integer, numeric, and character). They are more explicit versions of NA for these data types.

NA_complex_: This special value represents missing values for complex numbers.

NA_: This is a more general version of NA that can be used for missing values in various data types, allowing R to infer the appropriate data type based on the context.

NaN (Complex NaN): This represents a complex NaN value, used for undefined or unrepresentable complex numbers.

TRUE, FALSE, and T, F: These are the logical values used to represent true and false, often used in logical and boolean operations.

Inf, -Inf, and NaN (Complex): Similar to their real counterparts, these special values represent complex infinity and complex NaN values. The ellipsis (...) is used in function definitions to indicate a variable number of arguments or parameters. It allows a function to accept an arbitrary number of additional arguments.

NA for factors: In the context of factors, the value NA is used to indicate missing levels.

These special values are important for handling missing data, exceptional cases, and various data types in R. They are often used in data cleaning, data analysis, and statistical operations. It's important to be aware of these special values and how they are handled in R to effectively work with data and write robust code.

1.12 Classes:

A class is just a blueprint or a data type of an object. It represents the set of properties or methods that are common to all objects of one type. R has a three-class system. These are ***S3, S4,*** and ***Reference Classes.***

In R programming, a class is a bluepr int or template for creating objects, which are instances of the class. Classes define the structure and behavior of objects, allowing you to group data and functions together into a single unit. Object-oriented programming (OOP) is an important

programming paradigm in R, and R provides a flexible and extensible system for defining and using classes. Here are the key concepts related to classes in R:

I. S3 Class

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

Example 1:

```
# Create an S3 class object
x <- 1:5
class(x) <- "myclass"
```

Example 2:

```
# create a list with required components
movieList <- list(name = "Master", leadActor = "Vijay")
class(movieList) <- "movie"        # giving name "movie" to the class
movieList          #printing the properties and values of the object movieList.
```

Example with Output:

```
R 4.3.1 · ~/
> # create a list with required components
> movieList <- list(name = "Master", leadActor = "Vijay")
> class(movieList) <- "movie" # giving name "movie" to the class
> movieList
$name
[1] "Master"

$leadActor
[1] "Vijay"

attr(,"class")
[1] "movie"
>
```

Example 2:

```
># create a list for student class
>stud <- list(roll = 123, name = "Syed", prog="BCA", batch="2022")
># giving name "STUDENT" to the class
>class(stud) <- "STUDENT"
>stud      # printing data from the stud object. It can also be written as print(stud)
```

Here stud is an object of STUDENT class.

II. S4 Class

S4 class is an improved class over the S3. It has a formally defined structure which helps in making objects of the same class look more or less similar. Class components are properly defined using the **setClass()** function and objects are created using the **new()** function.

Example:

```
#creating a class "student" with 3 slots in the list(name,age and GPA)
setClass("student", slots = list(name = "character", age = "numeric", GPA = "numeric"))
# creating an object "s" of the "student" class
s <- new("student", name = "Mohan", age = 22, GPA = 8.5)
# print the object
print(s)
```

Example with output

```
R 4.3.1 · ~/
> setClass("student", slots = list(name = "character", age = "numeric", GPA = "numeric"))
> # creating an object "s" of the "student" class
> s <- new("student", name = "Mohan", age = 22, GPA = 8.5)
> # print the object
> print(s)
An object of class "student"
Slot "name":
[1] "Mohan"

Slot "age":
[1] 22

Slot "GPA":
[1] 8.5
```

III. Reference Class

Reference classes are very similar to other object oriented programming languages. It is basically a S4 classed with an environment added to it.

Example:

```
# defining the reference class "student"
student<-setRefClass("student", fields=list( name="character", age="numeric", GPA="numeric" )
)
# creating an object of the "student" reference class
s <- student$new(name = "Mohan", age = 22, GPA = 8.5)
# print the object
print(s)
```

IV. Constructor and Methods:

Constructors are functions used to create objects of a specific class, and methods are functions associated with a class that operate on its objects. In R, methods are usually implemented as functions with the same name as the generic function and with a class-specific method. For example, you can create a method for the print function that is specific to your class.

Example:

```
setClass("Person", slots = list(name = "character", age = "numeric"))
setMethod("print","Person", function(x) {
cat("Name:", x@name, "\n")
```

```
    cat("Age:", x@age, "\n")
  })
person <- new("Person", name = "Charlie", age = 35)
print(person)
```

V. Inheritance:

Inheritance allows you to create new classes that are based on existing classes, inheriting their properties and methods. This is a fundamental concept in OOP. In R, you can define inheritance relationships using the setClass() or setRefClass() functions.

Example:

```
setClass("Employee", contains = "Person", slots = list(employee_id = "integer"))
```

Classes in R provide a way to organize and encapsulate data and functionality, making code more modular, readable, and maintainable. Depending on your needs and the complexity of your project, you can choose between S3, S4, or reference classes to define your custom classes in R.

1.13 Coercion or type casting in R:

Coercing an object from one type of class to another is known as explicit coercion. It is achieved through some functions which are similar to the base functions. But they differ from base functions as they are not generic and hence do not call S3 class methods for conversion.

Coercion, also known as type casting, is the process of converting data from one data type to another in R. R provides several functions and operators for type coercion to ensure that data can be correctly interpreted and used in various operations. Coercion may be automatic or explicit, depending on the context and the operation being performed.

Difference between conversion, coercion and cast: Normally, whatever is converted implicitly is referred to as coercion and if converted explicitly then it is known as casting. Conversion signifies both types- coercion and casting.

I. Automatic Coercion:

Automatic coercion occurs when R converts data types implicitly to ensure that operations can be carried out without errors. For example, in arithmetic operations involving different data types, R may automatically coerce one or more values to a common data type.

Example:

```
x <- 5
y <- 3.5
result <- x + y  # Automatic coercion of x to a numeric (result will be 8.5)
```

II. Explicit Coercion:

Explicit coercion is when you specifically instruct R to convert a data type to another using conversion functions. R provides functions for explicit type casting, including:

- as.character(): Converts to character data type.
- as.numeric(): Converts to numeric data type.
- as.integer(): Converts to integer data type.
- as.logical(): Converts to logical (boolean) data type.
- as.factor(): Converts to factor data type.
- as.data.frame(): Converts to data frame data type.

Example:

```
x <- 5
y <- as.character(x) # Explicitly coerce x to a character (y will be "5")
```

Explicit coercion to character

There are two functions to do so **as.character()** and **as.string()**. If v is a vector which is needed to be converted into character then it can be converted as:

- as.character(v, encoding = NULL)
- as.string(v, encoding = NULL)

Here the encoding parameter informs R compiler about encoding of the vector and helps internally in managing character and string vectors.

Example-

```
#Creating a list of even numbers
x<-c(2, 4, 6, 8)
x
# Converting it to character type using as.character encoding
as.character(x)
x
```

Execution:

```
R 4.3.1 · ~/
> #Creating a list of even numbers
> x<-c(2, 4, 6, 8)
> x
[1] 2 4 6 8
> # Converting it to character type using as.character encoding
> as.character(x)
[1] "2" "4" "6" "8"
```

Explicit coercion function

Functions	Description
as.logical	Converts the value to logical type. If 0 is present then it is converted to FALSE, Any other value is converted to TRUE
as.integer	Converts the object to integer type
as.double	Converts the object to double precision type
as.complex	Converts the object to complex type
as.list	It accepts only dictionary type or vector as input arguments in the parameter
as.character() , as.string()	It convert into character or string

Example of other explicit coercion functions-

```
# Creating a vector list
  x<-c(2, 4, 6, 7, 8, 9)
# Checking its class
  class(x)
# Converting it to integer type
  as.numeric(x)
# Converting it to double type
  as.double(x)
# Converting it to logical type
  as.logical(x)
# Converting it to a list
  as.list(x)
# Converting it to complex numbers
  as.complex(x)
```

III. Changing Factor Levels:

When working with factors, you may need to change the levels. You can use the levels() function to set the levels of a factor.

Example:

```
gender <- factor(c("Male", "Female", "Male"))
levels(gender) <- c("M", "F")
```

IV. Handling Missing Values:

In R, missing values are often represented by NA. You can explicitly specify the data type for NA values using the functions mentioned above.

Example:

```
x <- NA
y <- as.logical(x)  # Explicitly coerce NA to logical (y will be NA)
```

V. Data Frame Column Coercion:

When working with data frames, you may need to convert a column to a different data type. You can use functions like as.character(), as.numeric(), etc., to convert data frame columns explicitly.

Example:

```
df <- data.frame(A = c(1, 2, 3), B = c("4", "5", "6"))
df$B <- as.numeric(df$B)  # Explicitly coerce column B to numeric
```

Coercion is an essential part of working with data in R. By understanding and using the appropriate coercion methods, you can ensure that your data is in the correct format for various operations and analyses.

1.14 Basic Plotting:

Plotting is one of the main features of R programming, through which you can perform data visualization to get the insight of the data in graphical format. R has a number of built-in tools for basic plotting in graph types such as **histograms, scatter plots, bar charts, boxplots** and many more. There is a generic function, **plot()** for plotting x-y data for data visualization.

Types of data visualizations using R: Data set is the first requirement for performing data visualization using any types of plotting tools. Suppose the data set of air quality is given as :

Ozone	Solar R.	Wind	Temp	Month	Day
41	190	7.4	67	5	1
36	118	8.0	72	5	2
12	149	12.6	74	5	3
18	313	11.5	62	5	4
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6

1.14.1 Bar Plot:

There are two types of bar plots- **horizontal** and **vertical** which represent data points as horizontal or vertical bars of certain lengths proportional to the value of the data item. They are generally used for continuous and categorical variable plotting. By setting the **horiz** parameter to **true** and **false**, we can get horizontal and vertical bar plots respectively.

R code for bar plotting:

```
# Horizontal Bar Plot for Ozone concentration in air
```

barplot (airquality $Ozone, main = 'Ozone Concentration in air', xlab = 'ozone levels', horiz = TRUE)

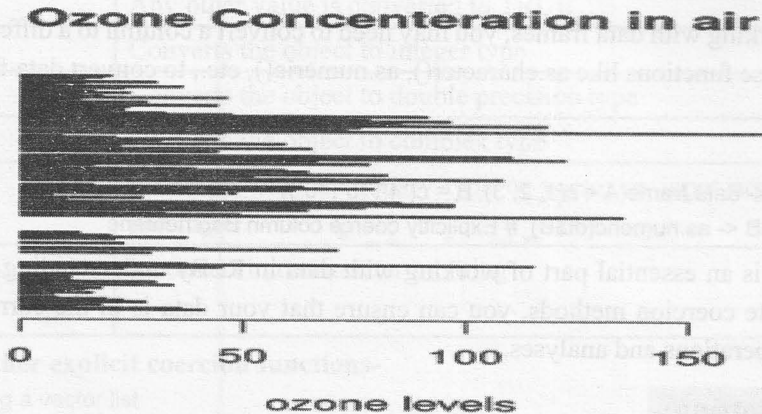


Fig 1.4: Bar plotting with True horiz parameter

```
# Vertical Bar Plot for Ozone concentration in air
```

barplot(airquality $Ozone, main = 'Ozone Concenteration in air', xlab = 'ozone levels', col ='blue', horiz = FALSE)

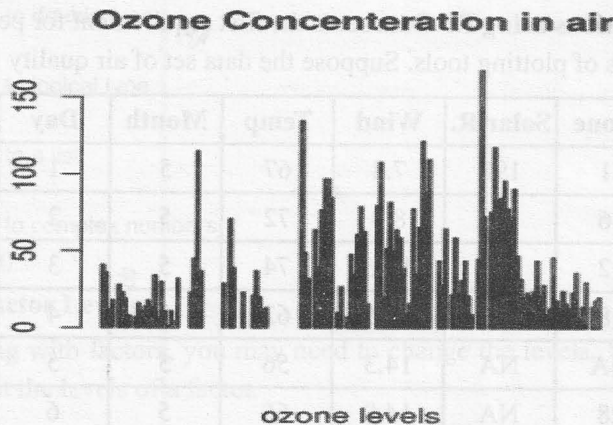


Fig 1.5: Bar plotting with False horiz parameter

1.14.2 Histogram:

A histogram is like a bar chart as it uses bars of varying height to represent data distribution. However, in a histogram values are grouped into consecutive intervals called bins. In a Histogram, continuous values are grouped and displayed in these bins whose size can be varied.

R code for generating histogram:

```
# Histogram for Maximum Daily Temperature
data(airquality)
        hist(airquality $Temp, main =" ", xlab ="Temperature(Fahrenheit)", xlim = c(50, 125),
                                    col ="yellow", freq = TRUE)
```

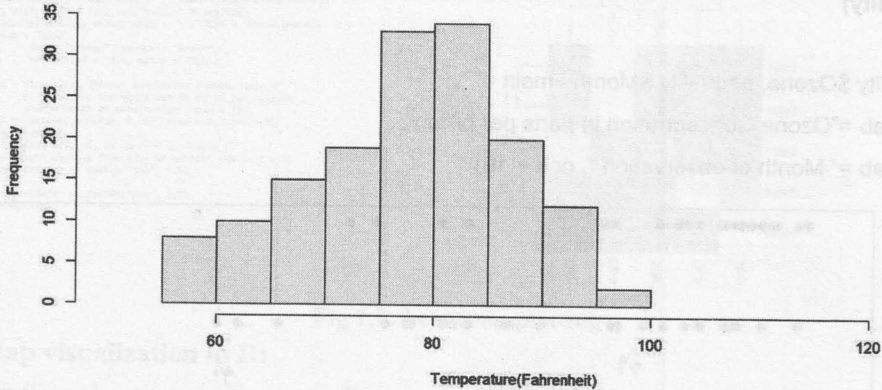


Fig 1.6 : Histogram plotting

1.14.3 Box Plot:

The statistical summary of the given data is presented graphically using a **boxplot**. A box plot depicts information like the minimum and maximum data point, the median value, first and third quartile, and interquartile range.

R code to generate the box plotting:

```
# Box plot for average wind speed
data(airquality)
boxplot(airquality $Wind, main = " ", xlab ="Miles per hour", ylab = "Wind",
                              col = "orange", border = "brown",
                              horizontal = TRUE, notch = TRUE )
```

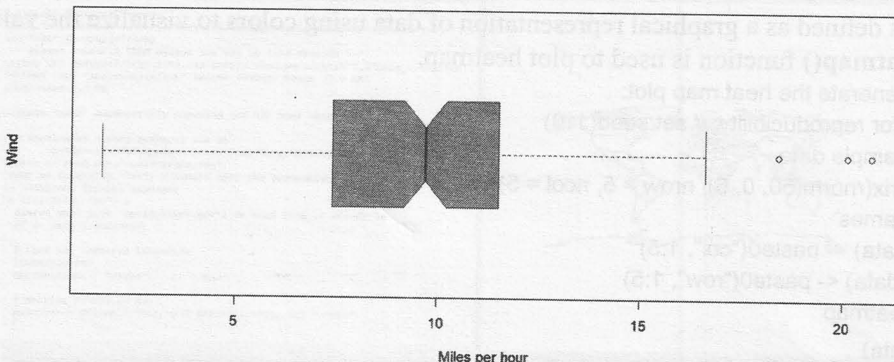


Fig 1.7: Box plotting

1.14.4 Scatter Plot:

A scatter plot is composed of many points on a Cartesian plane. Each point denotes the value taken by two parameters and helps us easily identify the relationship between them.

R code to generate the scatter plot:

```
# Scatter plot for Ozone Concentration per month
data(airquality)


plot ( airquality $Ozone, airquality $Month,  main =" ",
       xlab ="Ozone Concentration in parts per billion",
       ylab =" Month of observation ", pch = 19)
```

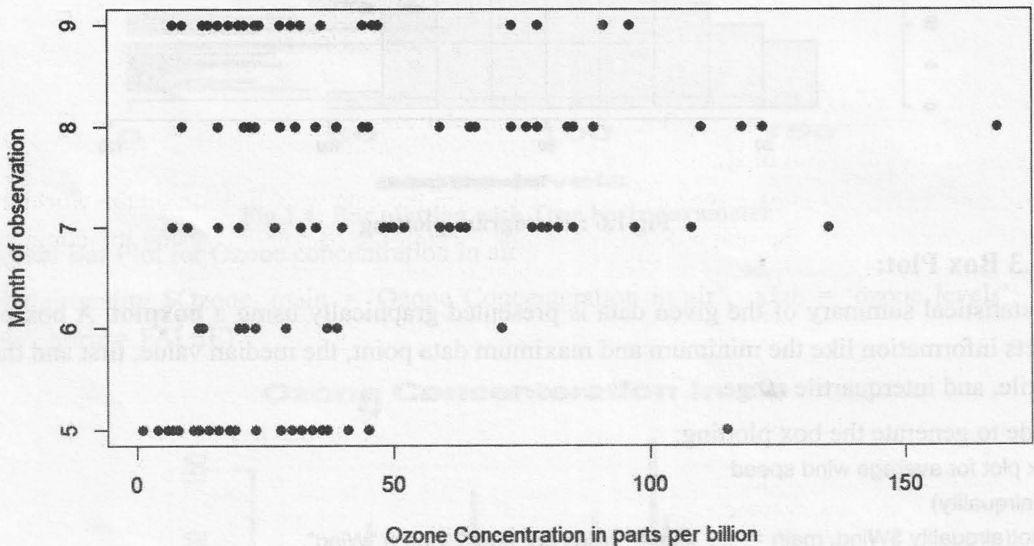


Fig 1.8: Scatter plotting

1.14.5 Heat Map:

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. **heatmap()** function is used to plot heatmap.

R code to generate the heat map plot:

```
# Set seed for reproducibility, # set.seed(110)
# Create example data
data <- matrix(rnorm(50, 0, 5), nrow = 5, ncol = 5)
# Column names
colnames(data) <- paste0("col", 1:5)
rownames(data) <- paste0("row", 1:5)
# Draw a heatmap
heatmap(data)
```

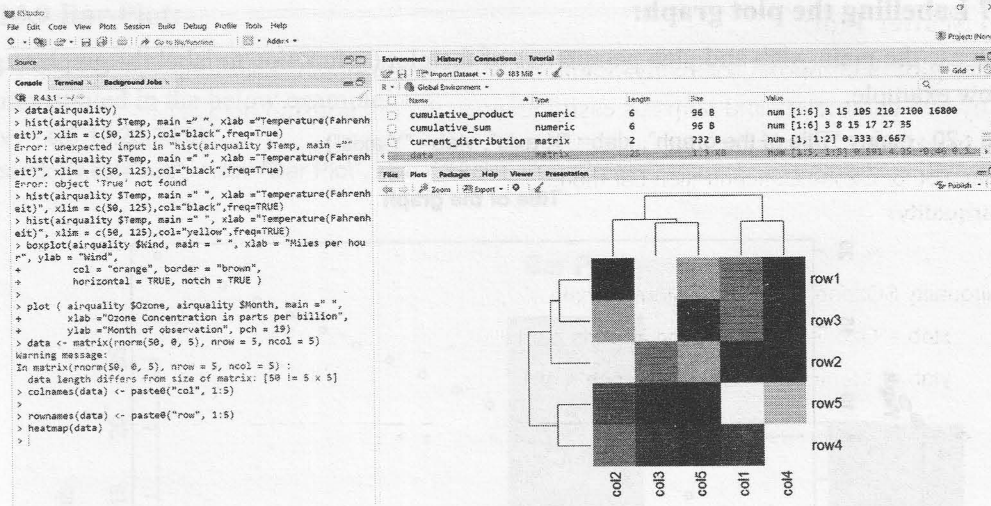


Fig 1.9: heat map plotting

1.14.6 Map visualization in R:

After installing the **maps** packages in R we can perform the data visualization on the world map as well. Before running the below R code you must install the package by using the command: **install.packages("maps")**

R code for data visualization on world map using the map function as given below:

```
# Reading the dataset and converting it into the Dataframe
        data <- read.csv("worldcities.csv")
        df <- data.frame(data)
# Loading the required libraries
        library(maps)
        map(database = "world")
# marking points on the world map
        points(x = df$lat[1:500], y = df$lng[1:500], col = "Red")
```

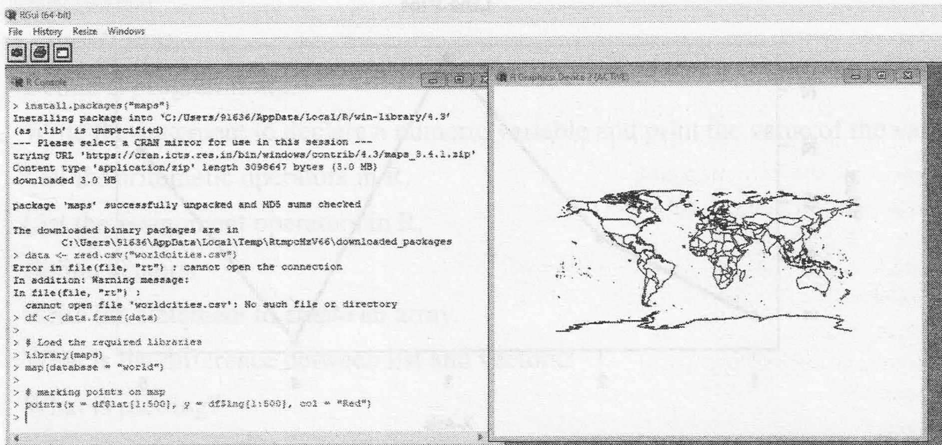


Fig 1.10: Map visualization

1.14.7 Labelling the plot graph:

We can use the main, xlab and ylab parameters of the plot ( ) function to label the graph as given in below example.

```
>plot( 1 : 20 , main= "Title of the graph", xlab= "X-axis", ylab= "Y-axis")
```

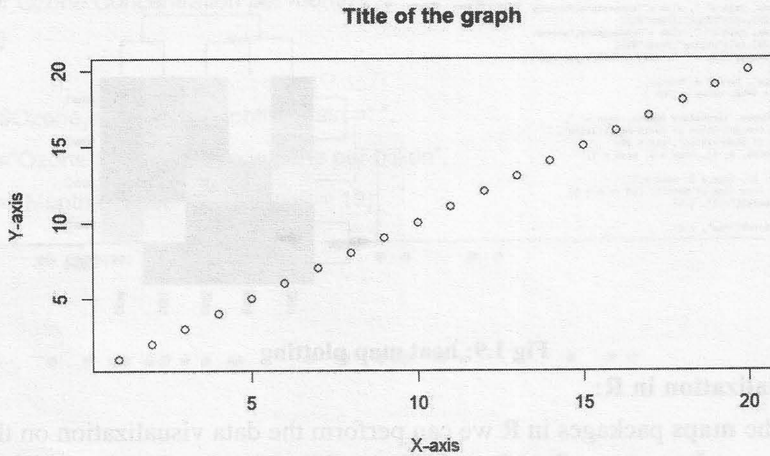


Fig 1.11: Labelling the plot

1.14.8 Line Plot:

We can also visualize our data using the plot using the connected line in the graph. Lets see the below example for getting a line plot.

```
#creating a vector v with a dummy data set and creating a line plot.
> v <- c(5,12,28,3,20)
> plot(v, type = "o", col="black", lwd=3, main="Line Plot", xlab="X-Axis", ylab="Y-Axis")
```

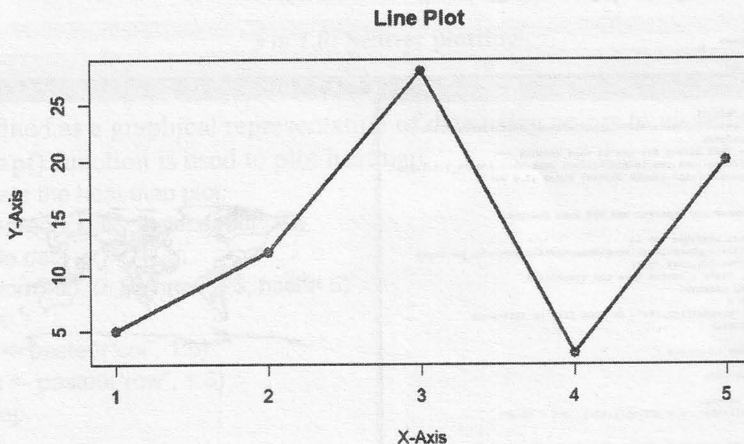


Fig 1.12 Creating a line plot with labelled graph

1.14.9 Bar Plot:

We can create a bar plot with a given data set in R programming using **barplot( )** function, as demonstrated in the below example.

```
> v <- c(5,12,28,3,20)
> barplot(v,col="red", main="Bar Plot", xlab="XAxis", ylab="YAxis")
```

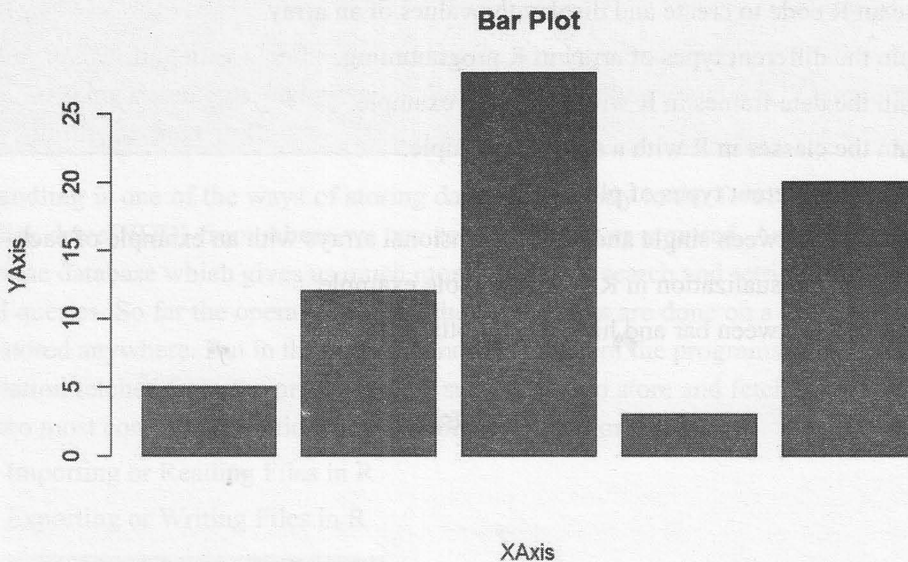


Fig 1.13 Creating a bar plot

1.15 Exercise

Short Questions:

1. Define R programming.
2. Write the statement to declare a numeric variable and print the value of the variable.
3. List the arithmetic operators in R.
4. List the assignment operators in R.
5. Define vectors.
6. Write the statement to create an array.
7. What is the difference between list and vectors?
8. What is plotting?

9. What is coercion in R?
10. What is a data frame?

Long Questions:

1. Differentiate between R and C programming.
2. Explain the different types of operators in R.
3. Create an R code to create and display the values of an array.
4. Explain the different types of array in R programming.
5. Explain the data frames in R with a suitable example.
6. Explain the classes in R with a suitable example.
7. Explain the different types of plotting.
8. Differentiate between single and multidimensional arrays with an example of each.
9. Create a Map visualization in R with a suitable example.
10. Differentiate between bar and histogram plotting.