

STATISTICAL COMPUTING USING R

M.Sc., STATISTICS First Year

Semester – I, Paper-II

Lesson Writers

Dr. R. Vishnu Vardhan

Associate Professor
Department of Statistics
Pondicherry University

Dr. D. Ramesh

Assistant Professor,
Department of Statistics and
Computer Applications, ANGRAU,
Agricultural College, Bapatla

Dr. Syed Jilani

Department of Statistics
University College of Sciences
Acharya Nagarjuna University

Dr. S. Bhanu Prakash

Assistant Professor
Freshman Engineering Department
Dept. of Mathematics & Statistics
Godavari Global University
Rajamahendravaram

Editor:

Dr. R. Vishnu Vardhan

Associate Professor
Department of Statistics
Pondicherry University

Director, I/c

Prof. V.VENKATESWARLU

MA.,M.P.S., M.S.W.,M.Phil., Ph.D.

CENTREFORDISTANCEEDUCATION

ACHARAYANAGARJUNAUNIVERSITY

NAGARJUNANAGAR – 522510

Ph:0863-2346222,2346208,

0863-2346259(Study Material)

Website: www.anucde.info

e-mail:anucdedirector@gmail.com

M.Sc., STATISTICS - STATISTICAL COMPUTING USING R

First Edition 2025

No. of Copies :

© Acharya Nagarjuna University

This book is exclusively prepared for the use of students of M.SC.(Statistics) Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited Circulation only.

Published by:

Prof. V.VENKATESWARLU,

Director, I/C

Centre for Distance Education,

Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the doorstep of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K.Gangadhara Rao

*M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University*

M.Sc.–Statistics Syllabus

SEMESTER-I

102ST24: STATISTICAL COMPUTING USING R

Unit-I:

Introduction to R language: Objects (Atomics) -Basic types, modes and attributes, comments, constants. R–Data Types: character, numeric, integer, logical, complex and raw data types. R– Operators: arithmetic, relational, logical, assignment and miscellaneous operators. R– Variables: variable assignment, data type, finding variables using ls()function, deleting Variables using rm() function, R-I/O console functions-scan(), print(), cat(), format(), setwd() and getwd() functions. R-vectors: creating vectors, vector assignment, manipulating vectors, arithmetic, generating regular sequences, logical vectors, and character vectors, index vectors, selecting and modifying subsets of a vector. Manipulating character vectors using strsplit(), paste(), grep(), gsub() functions; R-factors: creating factor variables, handling factordata, generating factor levels using gl() function.

Unit-II:

R-Matrices: Creating matrices, arithmetic operators on matrices, matrix facilities, forming partitioned matrices, cbind() and rbind() functions, R-Lists: creating a list, naming, accessing and manipulating list elements, converting a list to a vector. R-Data frames: creation, adding rows and variables to data frame, attach() and detach(), working with data frames, data reshaping. Reading and getting data into R using files: reading data and writing data from / to files of type CSV, EXCEL, text and other data type files using the save(), load(), read.csv()and read.table(),write.csv() and write.table() functions. Retrieving files using file.choose(),function.

Unit-III:

R – Control Structures: Decision making-if, if-else, ladder if-else, nested if-else, and switch statements. Loops-repeat, while and for statements. Loop control statements -- break and next. R – Functions: function definition, function components, built-in functions, user-defined function, syntax of a function, function arguments, arguments matching, scope and evaluation, calling a function, one-line functions, using default values in functions. Built in R-functions and writing own R-functions or R-codes for small standard statistical problems like finding summary statistics, correlation, one-sample t-test, two-sample t-test and paired samples t-test, etc. Group manipulation using apply family of functions- apply, sapply, lapply and tapply.

Unit-IV:

R-Probability Distributions: Computing values of pdf, cdf, quantile and generating samples for binomial, poisson, normal, exponential, Weibull and other prominent distributions using Built inR – functions. Plotting density and cumulative density curves for the distributions. Built inR-syntaxes for the Shapiro-Wilk test of normality, Kolmogorov-Smirnov test for one-sample and two-sample cases, Wilcoxon Mann-Whitney one-sample and two-sample U- tests, chi- square tests for association and goodness of fit. Writing own R-functions or R- codes: Fitting of binomial, Poisson, normal, exponential, Weibull and logistic distributions based on a given frequency data and test for goodness of fit. Solving a non-linear equation using Newton- Raphson method.

Unit-V:

R-Graphics: Use of high-level plotting functions for creating histograms, scatter plots, box-whisker plots, bar plot, dot plot, line charts using numeric data and categorical data, pie charts, bar Charts, Q-Q plot and curves. Controlling plot options using low-level plotting functions, adding lines, segments, points, polygon, grid to the plotting region; Add text using legend, text, mtext; and modify/adjust axes, putting multiple plots on a single page. Built-in R-syntaxes for one-way ANOVA, two-way ANOVA.

BOOKS FOR STUDY:

- 1) Dr. Mark Gardener (2012): *Beginning R – The Statistical Programming Language*, Wiley India Pvt Ltd.
- 2) W.N.Venables and D.M.Smith(2016): *An Introduction to R*
- 3) J.P.Lander (2014): *R for Everyone, Pearson Publications*
- 4) Garrett G. Grolemund: *Hands-On Programming with R*

BOOKS FOR REFERENCES:

- 1) De Vries, A., and Meys, J. (2016). *R For Dummies*, Second Edition, John Wiley & Sons Private Ltd, NY
- 2) Crawley, M.J. (2007). *The R Book*, John Wiley and Sons Private Ltd., NY.

M.Sc DEGREE EXAMINATION

First Semester

Statistics::Paper- II-Statistical Computing Using R

MODEL QUESTION PAPER

Time: Three hours

Maximum:70 Marks

Answer ONE question from each unit

(5x14=70)

UNIT-I

1. (a) Explain different types of data types and give an illustration for each type.
(b) Explain various I/O console functions by means of illustrations.

(or)

2. (a) Explain how to create and manipulate vectors in R ?
(b) Explain the following functions with suitable illustrations.

UNIT -II

3. (a) Explain how to create and manipulate matrices in R with suitable illustrations. Also, explain various operators applicable on matrices.
(b) Explain, in detail, the creation and manipulation of data frames.

(or)

4. (a) Explain how to read data from various types of files by means of illustrations. Further, explain write.csv(), write.table(), file.choose(), setwd() and getwd() functions with suitable illustrations.
(b) Explain the creation and manipulation of lists.

UNIT - III

5. (a) Explain various control statements in R by writing their syntax. Give an illustration in each case.
(b) Write R program to find mean and median of the given sample without using built-in R functions.

(or)

6. (a) What are user-defined functions? Explain them in detail with suitable illustrations.
- (b) Write your own R function for two sample t-test.

UNIT – IV

7. (a) Write R-codes for generating samples of size $n=1000$ from each of the following probability distributions and for plotting the density functions for the respective distributions. i) Poisson(10) ii) $N(10,100)$ iii) $\exp(5)$
- (b) Write R- function for finding binomial probability and hence write R- code for fitting of binomial distribution based on a given frequency data and test for goodness of fit.

(or)

8. (a) Write down the built-in R-syntax for the following tests and explain them.
- i) Kolmogorov –Smirnov test for goodness of fit. ii) Wilcoxon Mann-Whitney two-sample U-test iii) Chi-square test for goodness of fit.
- (b) Write R- code for solving the equation $e^{2x} - x - 6 = 0$ using Newton-Raphson method.

UNIT –V

9. (a) Explain the following high-level plotting commands in details i) plot() ii) barplot() iii) pie() iv) hist()
- (b) Explain various low-level plotting commands available in R.

(or)

10. (a) Write down the built-in R-syntax for drawing bar chart and Q-Q plot.
- (b) Write down the built-in R-syntax for the following tests and explain them. i) CRD analysis ii) RBD analysis.
-

CONTENTS

S.NO.	LESSON	PAGES
1.	BASIC DATA TYPES AND OPERATORS	1.1 – 1.10
2.	VARIABLES AND INPUT AND OUTPUT FUNCTIONS	2.1 – 2.7
3.	VECTORS AND GENERATING REGULAR SEQUENCES OF FUNCTIONS	3.1 – 3.15
4.	R MATRICES	4.1 – 4.15
5.	R-LISTS	5.1 – 5.11
6.	READING AND GETTING DATA INTO R USING FILES	6.1 – 6.7
7.	CONTROL STATEMENTS	7.1 – 7.9
8.	LOOPING STATEMENTS	8.1 – 8.12
9.	R- FUNCTIONS	9.1 – 9.23
10.	PROBABILITY DISTRIBUTIONS IN R	10.1 – 10.20
11.	STATISTICAL TESTS IN R	11.1– 11.5
12.	R-CODES FOR FITTING DISTRIBUTIONS	12.1 – 12.17
13.	R-GRAPHICS	13.1 – 13.14
14.	R-GRAPHICS	14.1 – 14.16
15.	R-GRAPHICS	15.1 – 1.4

LESSON -1

BASIC DATA TYPES AND OPERATORS

OBJECTIVES:

After studying this unit, you should be able to:

- Recognize the significance of data in modern decision-making and computational processes.
- Students should have a solid understanding of the modes, attributes, and constants, and manage the R workspace efficiently.
- The student will learn apply arithmetic, relational, logical, assignment, and miscellaneous operators to perform various computational tasks.
- Develop simple R scripts for data analysis and statistical tasks, utilizing the fundamental concepts of R programming.

STRUCTURE:

1.1. What and why is R?

1.1.1 Features of R

1.1.2 Applications of R

1.2. Objects

1.3. Modes

1.3.1 Attributes

1.3.2 Comments

1.3.3 Constants

1.4 Basic data types

1.5 Operators

1.5.1 Arithmetic Operators

1.5.2 Relational Operators

1.5.3 Logical Operators

1.5.4. Assignment Operators

1.5.5 Miscellaneous Operators

1.6 Conclusion

1.7 Self Assessment Questions

1.8 Further Readings

1.1. WHAT AND WHY IS R:

R is a programming language and software environment that assists in the analysis of statistical data, the representation of images, and the generation of reports. R is a programming language that was initially developed by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand. The R Development Core Team is currently expanding the capabilities of R.

The core of R is an interpreted computer language that enables modular programming through the use of functions, as well as branching and looping. R makes it possible to integrate with procedures written in languages such as C, C++, .Net, Python, or FORTRAN, which leads to increased efficiency.

The GNU General Public Licence makes R freely accessible to the public, and pre-compiled binary copies of the programme are made available for a variety of operating systems, including Linux, Windows, and Mac Operating Systems.

R is a piece of free software that is shared under a copy left licence similar to that of GNU. It is also an official component of the GNU project known as GNU S. Starting from mid-1997, a central organisation known as the "R Core Team" has had the authority to make changes to the R source code archive.

1.1.1 Features of R:

R is a powerful programming language and environment primarily used for statistical computing and data analysis. Here are some key features:

1. Statistical Computing & Analysis

- Provides a wide range of **statistical tests** (e.g., regression, t-tests, ANOVA).
- Advanced **machine learning** and modeling capabilities.

2. Data Manipulation & Visualization

- Efficient tools for **data wrangling** (e.g., dplyr, tidyverse).
- **Data visualization** with libraries like ggplot2 and lattice.

3. Open-Source & Extensible

- Free to use and supported by a large community.
- Thousands of packages available via **CRAN** (Comprehensive R Archive Network).

4. Handling Big Data

- Supports **large datasets** with packages like data.table.
- Can integrate with **Hadoop**, Spark, and databases.

5. Reproducibility & Reporting

- **R Markdown** for creating dynamic reports.
- **Shiny** for interactive web applications.

6. Integration with Other Languages

- Can call **Python**, **C**, **C++**, **Java**, and SQL code.
- Works with tools like **Jupyter Notebooks**.

7. Rich Development Environment

- Supported by IDEs like **RStudio** and Jupyter.
- Version control integration with **Git**.

1.1.2 Applications of R:

1. Statistical analysis and making sense of data: At its core, R is the same thing as statistical research. It comes with all the tools you need to do a wide range of statistical tests, from simple descriptive statistics to complex regression models. R is great for more than just numbers. It's also great for showing data visually. `ggplot2` and other similar packages make it easy to make interesting graphs and charts that help you understand large datasets visually.

2. Exploring and cleaning the data: Exploring and cleaning the data are the first steps in any data analysis process. Because of what it can do, R is a great choice for dealing with missing values and outliers and checking the quality of the data as a whole before doing more in-depth analysis. In real life, R's powerful data preparation tools make sure that datasets are carefully prepared and improved so that insights are correct and reliable.

3. Predictive Modelling and Machine Learning: R has a lot of features for both predictive modelling and machine learning. It has many methods for regression, classification, and clustering, which makes it a great language for making models that can predict the future. R's machine learning features are very useful for real-time tasks like predicting stock prices, customer behaviour, or disease results, as they help make decisions based on data.

4. Biostatistics and Healthcare: R is a key tool in biostatistics; it is used to look at data from clinical trials, do epidemiological studies, and help healthcare workers make decisions based on data. Some of the ways it can be used in healthcare are in genomics, where it is very helpful for looking at genetic data, finding trends linked to diseases, and making personalized medicine possible.

5. Finance and Risk Management: Risk modelling, portfolio optimisation, and analyzing market trends are all things that the financial industry does with R. In financial analytics, where real-time insights can drive strategic decisions, R's ability to work with big datasets is very important.

People who want to become data scientists can learn how to use R programming for financial research and risk management by taking a well-rounded Data Science course.

6. Social Sciences and Market Research: R is used a lot in the social sciences to look at survey results, social media sentiment, and general opinion. Because it is so flexible, researchers can use it to learn from very large and different social datasets.

7. Environmental Science and Climate Research: R makes a big difference in environmental science by looking at climate data, guessing what will happen to the environment, and figuring out how our actions affect environments. Its uses in climate studies are very important for understanding and solving problems related to the environment. As worries about the planet's future grow, data scientists are turning to R for environmental study and climate modelling. This shows how R can be used in the real world to solve problems.

1.2 OBJECTS:

The entities that R creates and manipulates are known as objects. These may be variables, vectors, matrices, arrays of numbers, characteristics, functions are known general structures or more general structures built from such components.

During an R-session objects are created and stored by name. The R-command objects() or ls() can be used to display the names of the objects which are currently stored within R. the collection of objects currently stored is called the work space. We can remove any particular objects or object using the following function rm()

Example:

```
rm (x),
rm (mean)
```

To remove or erase all existing objects rm(list=ls())

1.3 MODES:

In R, data modes and classes define the fundamental attributes and behavior of a data object. For example, different modes and classes are handled differently by core functions like print(), summary(), and plot().

Data Object Modes:

All data in R is an object and all objects have a “mode.” The mode determines what type of information can be found within the object and how that information is stored. Atomic “modes” are the basic building blocks for data objects in R. There are 6 basic atomic modes:

Data Mode	Storage	Example
logical	Logical	TRUE or FALSE
numeric	integer, single or double	Floating point real numbers; 3, 0.753
complex	Complex	$3 + 2i$
character	character	strings in quotes (“”) or apostrophes (‘)
function	special or built-in	do.it <- function(x) {...}
name	Symbol	any name assigned to an object (e.g. "my.data")

1.3.1 ATTRIBUTES:

Objects can have **attributes**. Attributes are part of the object. These include:

- names
- dimnames
- dim
- class
- attributes (contain metadata)

You can also glean other attribute-like information such as length (works on vectors and lists) or number of characters (for character strings).

1.3.2 COMMENTS:

We can add comments to our code using the `#` character. It is useful to document our code in this way so that others (and us the next time we read it) have an easier time following what the code is doing.

We can also change a variable's value by assigning it a new value:

```
weight_kg <- 57.5
```

```
weight_kg
```

```
[1] 57.5
```

1.3.3. CONSTANTS:

Constants, as the name suggests, are entities whose value cannot be altered. Basic types of constant are numeric constants and character constants.

Numeric Constants

All numbers fall under this category. They can be of type integer, double or complex.

It can be checked with the `typeof()` function.

Numeric constants followed by `L` are regarded as integer and those followed by `i` are regarded as complex.

```
typeof(5)
```

```
[1] "double"
```

```
typeof(5L)
```

```
[1] "integer"
```

```
typeof(5i)
```

```
[1] "complex"
```

1.4 BASIC DATA TYPES:

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

- **Numeric:**
- **Integer:**
- **Character (String):**
- **Logical (Boolean):**
- **Complex**
- **raw**

Numeric:

Decimal values are called **numerics** in R. It is the default computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
x = 10.5    # assign a decimal value
x          # print the value of x
[1] 10.5
class(x)    # print the class name of x
[1] "numeric"
```

Integer:

In order to create an **integer** variable in R, we invoke the as.integer function. We can be assured that y is indeed an integer by applying the is.integer function.

```
y = as.integer(3)
y          # print the value of y
[1] 3
class(y)    # print the class name of y
[1] "integer"
is.integer(y) # is y an integer?
[1] TRUE
```

Incidentally, we can coerce a numeric value into an integer with the same as.integer function.

```
as.integer(3.14) # coerce a numeric value
[1] 3
```

And we can parse a string for decimal values in much the same way.

```
as.integer("5.27") # coerce a decimal string
[1] 5
```

Complex

A **complex** value in R is defined via the pure imaginary value *i*.

```
z = 1 + 2i # create a complex number
```

```
z # print the value of z
```

```
[1] 1+2i
```

```
class(z) # print the class name of z
```

```
[1] "complex"
```

The following gives an error as -1 is not a complex value.

```
sqrt(-1) # square root of -1
```

```
[1] NaN
```

Logical:

A **logical** value is often created via comparison between variables.

```
x = 1; y = 2 # sample values
```

```
z = x > y # is x larger than y?
```

```
z # print the logical value
```

```
[1] FALSE
```

```
class(z) # print the class name of z
```

```
[1] "logical"
```

Standard logical operations are "&" (and), "|" (or), and "!" (negation).

```
u = TRUE; v = FALSE
```

```
u & v # u AND v
```

```
[1] FALSE
```

```
u | v # u OR v
```

```
[1] TRUE
```

```
!u # negation of u
```

```
[1] FALSE
```

Character:

A **character** object is used to represent string values in R. We convert objects into character values with the `as.character()` function:

```
x = as.character(3.14)
```

```
x # print the character string
```

```
[1] "3.14"
```

```
class(x) # print the class name of x
```

```
[1] "character"
```

Two character values can be concatenated with the `paste` function.

```
fname = "Joe"; lname = "Smith"
```

```
paste(fname, lname)
```

```
[1] "Joe Smith"
```

raw Data

In R, raw data types refer to the **raw** class, which is used to store raw bytes. It is primarily used for handling binary data, such as reading and writing files in binary format or dealing with cryptographic operations.

You can create raw data using the `as.raw()` function:

```
# Creating a raw vector
```

```
raw_vec <- as.raw(c(65, 66, 67)) #ASCII codes for 'A', 'B', 'C'
```

```
raw_vec
```

Output:

```
[1] 41 42 43
```

1.5 OPERATORS:

Introduction

R is a powerful programming language widely used for statistical computing and data analysis. One of the fundamental aspects of R is its use of operators to perform various computations and logical evaluations. Operators in R can be classified into different categories, including arithmetic, relational, logical, assignment, and miscellaneous operators. Understanding these operators is crucial for effectively writing and executing R scripts.

1.5.1 Arithmetic Operators

Arithmetic operators in R are used to perform basic mathematical computations. These include addition, subtraction, multiplication, division, exponentiation, and modulo operations. The following table outlines the arithmetic operators in R:

<i>Operator</i>	<i>Description</i>	<i>Example</i>
+	Addition	5 + 3 results in 8
-	Subtraction	10 - 4 results in 6
*	Multiplication	6 * 2 results in 12
/	Division	9 / 3 results in 3
^ or **	Exponentiation	2^3 or 2**3 results in 8
%%	Modulus (Remainder)	10 %% 3 results in 1
%/%	Integer Division	10 %/% 3 results in 3

1.5.2 Relational Operators

Relational operators in R are used to compare values and return Boolean results (TRUE or FALSE). These operators are essential for decision-making in programming.

Operator	Description	Example
==	Equal to	5 == 5 results in TRUE
!=	Not equal to	5 != 3 results in TRUE
>	Greater than	7 > 4 results in TRUE
<	Less than	3 < 5 results in TRUE
>=	Greater than or equal to	6 >= 6 results in TRUE
<=	Less than or equal to	4 <= 5 results in TRUE

1.5.3 Logical Operators

Logical operators in R are used for evaluating logical expressions and combining multiple conditions.

<i>Operator</i>	<i>Description</i>	<i>Usage</i>
&	<i>Element wise logical AND operation</i>	<i>a&b</i>
	<i>Element wise logical OR operation</i>	<i>a b</i>
!	<i>Element wise logical NOT operation</i>	<i>!a</i>
&&	<i>Operand wise logical AND operation</i>	<i>a&&b</i>
	<i>Operand wise logical OR operation</i>	<i>a b</i>

1.5.4 Assignment Operators

Assignment operators in R are used to assign values to variables. R provides multiple assignment operators.

Operator	Description	Example
<=	Left assignment	x <- 10
=>	Right assignment	10 -> x
=	Alternative assignment	x = 5

Operator	Description	Example
<<-	Global assignment (used in functions)	x <<- 20

1.5.5 Miscellaneous Operators

Miscellaneous operators in R include operators used for special operations, such as sequence generation, membership checking, and matrix multiplication.

<i>Operator</i>	<i>Description</i>	<i>Example</i>
:	Sequence generator	1:5 results in 1 2 3 4 5
%in%	Membership checking	3 %in% c(1, 2, 3, 4) results in TRUE
%*%	Matrix multiplication	A %*% B (for matrices A and B)

1.6 SUMMARY:

R is a powerful language for data analysis and visualization, offering rich features and an extensive ecosystem for statistical computing. Understanding its core concepts, such as objects, data types, and operators, is essential for efficient programming in R.

1.7 SELF ASSESSMENT QUESTIONS:

1. Explain the difference between a vector and a list in R.
2. Write an R script to demonstrate the use of arithmetic operators.
3. What are attributes in R? Give an example.
4. How do you comment code in R?
5. Discuss the applications of R in different industries.
6. Explain the different modes available in R with examples.
7. Describe logical operators in R with examples.

1.8 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Grolemond : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON -2

VARIABLES AND INPUT AND OUTPUT FUNCTIONS

OBJECTIVES:

After studying this unit, you should be able to:

- To understanding the basic data types and operators
- To know the concept of Structure and data types and operators
- To acquire knowledge about significance of various data types in R (e.g., numeric, character, logical), and understand their role in data manipulation..
- To understand the purpose and objectives of pivotal provisions of the data types and operators.

STRUCTURE:

2.1 Introduction

2.2 R Variables

2.2.1 Variable Assignment

2.2.2 Data Types in R

2.3 Finding variable:

2.3.1 Deleting Variables

2.4 R I/O Functions

2.4.1 Scan ()

2.4.2 Print()

2.4.3 cat()

2.4.4 format ()

2.5 getwd()

2.5.1 setwd()

2.6 Conclusion

2.7 Self Assessment Questions

2.8 Further Readings

2.1 INTRODUCTION:

A variable in R is a name assigned to a value or an object that can store different types of data such as numbers, text, or complex structures. Variables are used to store and manipulate data efficiently in R programming.

2.2 R VARIABLES:

2.2.1 Variable Assignment:

There are three different assignment operators. Those are leftwards(<-), rightwards(->) and equal to(=). Two of them leftward and rightward can be used normally used in functions, where as the operator equal to is only allowed at the top level(in the complete expression typed at the command prompt or as one of the sub expressions in a braced list of expressions). We use the print() or cat()function. This function is used to combine multiple items.

```
#Assignment using leftward operator
```

```
x<-("operator")
```

```
#Assignment using rightward operator
```

```
("variable")-> y
```

```
#Assignment using equal operator
```

```
z=c(1,2,3)
```

```
print(x)
```

```
print(y)
```

```
print(z)
```

```
cat("x is ",x,"\n")
```

```
cat("y is ",y,"\n")
```

```
cat("z is ",z,"\n")
```

Output:

```
[1] "operator"
```

```
[1] "variable"
```

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

```
x is operator
```

```
y is variable
```

```
z is 1 2 3
```

2.2.2 Data Types in R:

R supports multiple data types, which are crucial for storing and processing different kinds of information. The main data types include:

1. **Numeric (double and integer):** Used for numbers.


```
num <- 10.5 # Double
int <- as.integer(5) # Integer
```

2. **Character:** Used for text strings.

```
char <- "Hello, R!"
```

3. **Logical:** Represents TRUE or FALSE values.

```
bool <- TRUE
```

4. **Complex:** Used for complex numbers.

```
comp <- 4 + 3i
```

5. **Factor:** Used to represent categorical data.

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

output:

```
Numeric (Double) 10.5
Integer 5
Character Hello, R!
Logical TRUE
Complex 4+3i
Factor Male
```

2.3 FINDING VARIABLE:

When you are running commands in an R command prompt, the instance might get stacked up with lot of variables.

To find all R variables that are live at a point in R command prompt or R script file, `ls()` is the command that returns a character vector.

```
>ls()

[1] "a" "b" "c" "d" "x" "y" "z"

p=35.4

w=34

t=45

>ls()

[1] "a" "b" "c" "d" "p" "t" "w" "x" "y" "z"
```

2.3.1 Deleting Variables

The `rm()` function in R is used to remove objects from the environment. It is a powerful tool for managing memory and ensuring that unnecessary variables do not clutter the workspace.

- **Removing a Single Variable**

To delete a specific variable, use:

```
rm(x)
```

This will remove `x` from the environment, making it inaccessible.

- **Removing Multiple Variables**

You can delete multiple variables at once by passing multiple variable names to `rm()`:

```
rm(y, z)
```

- **Removing All Variables**

To clear all variables from the workspace, use:

```
rm(list = ls())
```

This removes all objects from the global environment, effectively resetting it.

- **Checking if a Variable Exists**

After removing a variable, you can check if it still exists using:

```
exists("x")
```

If `x` has been deleted, this function will return `FALSE`.

- **Protecting Variables from Accidental Deletion**

If you want to protect certain variables from being deleted, you can selectively remove others:

```
rm(list = setdiff(ls(), c("important_var")))
```

This keeps `important_var` while deleting all other variables.

2.4 R I/O FUNCTIONS:

2.4.1 Scan():-

`scan()` can accept a variety of connection functions. It reads data from a file, a URL or into a vector. It can be embedded in a call to `matrix()` or `array()`.

```
x<-scan("", "int")
1: 32
2: 43
3: 54
4: 67
5:
Read 4 items
> x
[1] "32" "43" "54" "67"
```

2.4.2 Print():

This function can be used to display the entire object, and is invoked when an expression is not assigned to a value. For lists and arrays include subscripting information.

```
print(7)
[1] 7

print(matrix(c(1,2,3,4),ncol=2))

[,1] [,2]

[1,]  1   3

[2,]  2   4
```

2.4.3 cat()

cat() converts numerical complex elements in the same way as print(). It uses the minimum field width necessary for each element, rather than the same field width for all elements. cat() will drop attributes of its inputs. This can write to a file by passing string to the file argument.

```
#Example for cat() function

a=23

b=42

cat("The sum of a&b is",a+b,"\n")

cat("The product of a&b is",a*b,"\n")
```

Output:

The sum of a&b is 65

The product of a&b is 966

2.4.4 Format():

The function format() allows you to format an R object for printing. Essentially, format() treats the elements of a vector as character strings using a

common format. This is especially useful when printing numbers and quantities under different formats.

```
format(13.7)
[1] "13.7"
>format(13.123456)
[1] "13.12346"
```

2.5 getwd():

R looks for your data file in the default directory. You can find the default directory by using the `getwd()` command like so:

```
getwd()
[1] "C:/Documents and Settings/Administrator/My Documents"
getwd()
[1] "/Users/markgardener"
getwd()
[1] "/home/mark"
```

2.5.1 setwd()

If your file is somewhere else you must type its name and location in full. The location is relative to the default directory; in the preceding example the file was on the desktop so the command ought to have been:

```
data6 = scan(file = 'Desktop/test data.txt')
```

The filename and directories are all case sensitive. You can also type in a URL and link to a file over the Internet directly; once again the full URL is required.

It may be easier to point permanently at a directory so that the files can be loaded simply by typing their names. You can alter the working directory using the `setwd()` command:

`setwd('pathname')`

When using this command, replace the `pathname` part with the location of your target directory.

The location is always relative to the current working directory, so to set to my Desktop I used the following:

```
setwd('Desktop')
getwd()
[1] "/Users/markgardener/Desktop"
```

To step up one level you can type the following:

```
setwd('.')
```

You can look at a directory and see which files/folders

2.6 SUMMARY:

Understanding variables, data types, and input/output functions in R is crucial for efficient programming and data analysis. Mastering these fundamental concepts allows users to write better scripts, manage data effectively, and interact seamlessly with the R

environment. The ability to assign, manipulate, and delete variables, along with performing input/output operations, forms the backbone of successful data handling in R.

2.7 SELF ASSESSMENT QUESTIONS:

1. Explain the process of assigning values to variables in R.
2. List and describe the basic data types in R.
3. Write the R command to display all variables in the current environment.
4. How can a variable be removed from the workspace in R?
5. Differentiate between `print()` and `cat()` functions in R.
6. Demonstrate the use of `getwd()` and `setwd()` functions with an example.
7. What is the purpose of the `scan()` function in R?
8. Explain the `format()` function with an example.

2.8 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Golemund : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON -3

VECTORS AND GENERATING REGULAR SEQUENCES OF FUNCTIONS

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the Concept and Importance of Vectors in R
- Students should have a solid understanding of the fundamentals of R Programming
- Understand apply logical vectors for indexing and filtering, and handle character vectors for text data manipulation.
- To understand the purpose and objectives of pivotal provisions of the vectors and generating regular sequences of functions.

STRUCTURE:

- 3.1 Introduction**
- 3.2 R vectors**
- 3.3 Generating Regular Sequences**
- 3.4 Logical Vectors**
- 3.5 Character Vectors**
- 3.6 Index Vectors**
- 3.7 Selecting and Modifying Subsets of a Vector**
- 3.8 Manipulating character vectors**
- 3.9 Factors**
- 3.10 Conclusion**
- 3.11 Self Assessment Questions**
- 3.12 Further Readings**

3.1 INTRODUCTION:

Vectors are one of the most fundamental data structures in R. A vector is a sequence of elements of the same data type. Vectors can hold numeric, integer, character, logical, or complex values. They are commonly used for performing operations on multiple data elements simultaneously.

3.2 R VECTORS

3.2.1 Creating Vectors

When you want to create vector with more than one element, you should use `c()` function which means to combine the elements into a vector.

```
# Create a vector.  
apple <- c('red','green',"yellow")  
print(apple)
```

```
# Get the class of the vector.  
print(class(apple))
```

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

```
[1] "red" "green" "yellow"  
[1] character"
```

3.2.2vector Assignment

A vector is a basic data structure in R. It is a sequence of elements of the same type. Vectors are the most common data type in R and are used extensively in data analysis and statistical computations.

```
# Assigning a character vector  
character_vector <- c("apple", "banana", "cherry")  
character_vector
```

Output:

```
[1] "apple" "banana" "cherry"
```

3.2.3 Manipulating Vectors

Vectors can be modified by adding or removing elements:

```
vec <- c(1, 2, 3)  
vec <- c(vec, 4, 5)  
vec
```

Output:

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

3.2.4 Arithmetic Operations on Vectors

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

```
# Create two vectors.
```

```
v1 <- c(3,8,4,5,0,11)
```

```
v2 <- c(4,11,0,8,1,2)
```

```
# Vector addition.
```

```
add.result <- v1+v2
```

```
print(add.result)
```

```
# Vector subtraction.
```

```
sub.result <- v1-v2
```

```
print(sub.result)
```

```
# Vector multiplication.
```

```
multi.result <- v1*v2
```

```
print(multi.result)
```

```
# Vector division.
```

```
divi.result <- v1/v2
```

```
print(divi.result)
```

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

```
[1] 7 19 4 13 1 13
```

```
[1] -1 -3 4 -3 -1 9
```

```
[1] 12 88 0 40 0 22
```

```
[1] 0.7500000 0.7272727 Inf 0.6250000 0.0000000 5.5000000
```

3.3 GENERATING REGULAR SEQUENCES:

Regular sequences are commonly used in R for generating sequences of numbers with a specific pattern. These sequences are often used in data manipulation, indexing, and simulation tasks. R provides several functions to generate regular sequences efficiently.

3.3.1 . seq() Function

The seq() function is the primary function in R to generate sequences. It allows creating sequences with customized steps, lengths, and patterns.

Syntax:

seq(from, to, by, length.out, along.with)

- from: Starting value of the sequence.
- to: Ending value of the sequence.
- by: Increment between sequence values.
- length.out: Desired length of the sequence.
- along.with: Takes the length of an existing object to define the sequence.

Examples:

Sequence from 1 to 10

```
seq(1, 10)
```

Sequence from 1 to 10 with a step of 2

```
seq(1, 10, by = 2)
```

Sequence from 1 to 10 with exactly 5 elements

```
seq(1, 10, length.out = 5)
```

Sequence along the length of another vector

```
seq(along.with = c(3, 5, 7, 9))
```

Output:

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

```
[1] 1 3 5 7 9
```

```
[1] 1.00 3.25 5.50 7.75 10.00
```

```
[1] 1 2 3 4
```

3.3.2 . Colon Operator

The colon operator: is a shorthand way to create a sequence of integers with a step of 1.

Syntax:

start:end

Examples:

```
# Sequence from 1 to 10
```

```
1:10
```

```
# Sequence from 5 to 1
```

```
5:1
```

Output:

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

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

3.3.3 . rep() Function

The rep() function is used to replicate the values in a vector.

Syntax:

```
rep(x, times, each, length.out)
```

x: The values to replicate.

times: Number of times to replicate the entire vector.

each: Number of times to replicate each element.

length.out: Desired length of the result.

Examples:

```
# Replicate the vector 1, 2, 3 two times
```

```
rep(c(1, 2, 3), times = 2)
```

```
# Replicate each element of the vector 1, 2, 3 two times
```

```
rep(c(1, 2, 3), each = 2)
```

```
# Replicate the vector 1, 2, 3 to a specified length of 5
```

```
rep(c(1, 2, 3), length.out = 5)
```

Output:

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

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

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

3.4 LOGICAL VECTORS:

Logical vectors are vectors that contain only TRUE, FALSE, or NA values. They are often used in conditions, indexing, and logical operations.

Syntax:

```
c(TRUE, FALSE, NA)
```

Examples:

```
# Creating a logical vector
```

```
logical_vector <- c(TRUE, FALSE, TRUE, NA)
```

```
logical_vector
```

```
# Using logical vectors for indexing
```

```
numeric_vector <- c(10, 20, 30, 40)
```

```
numeric_vector[logical_vector]
```

Output:

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

```
[1] 10 30
```

3.5 CHARACTER VECTORS:

Character vectors are vectors that contain character strings. They are often used to represent names, labels, and categorical data.

Syntax:

```
c("string1", "string2", "string3")
```

Examples:

```
# Creating a character vector
char_vector <- c("apple", "banana", "cherry")
char_vector
```

```
# Accessing elements of a character vector
char_vector[2]
```

Output:

```
[1] "apple" "banana" "cherry"
[1] "banana"
```

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

Output

```
[1] "Alice" "Bob" "Charlie"
```

3.6 INDEX VECTORS:

Index vectors are used to select elements from a vector based on their positions. They can be numeric, logical, or character vectors.

Syntax:

```
vector[index]
```

Examples:

```
vec <- c(10, 20, 30, 40, 50)
```

```
vec[2]
```

```
vec[c(1, 3, 5)]
```

Output

```
[1] 20
```

```
[1] 10 30 50
```

3.7 SELECTING AND MODIFYING SUBSETS OF A VECTOR:

Selecting and modifying subsets of a vector is a common task in R, which allows users to extract or change parts of a vector.

Syntax:

```
vector[index] <- value
```

Example:1

```
# Selecting a subset
```

```
vec <- c(1, 2, 3, 4, 5)
```

```
subset <- vec[2:4]
```

```
subset
```

```
# Modifying a subset
```

```
vec[2:4] <- c(10, 20, 30)
```

```
vec
```

Output:

```
[1] 2 3 4
```

```
[1] 1 10 20 30 5
```

Example:2

```
vec <- c(5, 10, 15, 20, 25)
```

```
vec[vec > 10] <- 100
```

```
vec
```

Output

```
[1] 5 10 100 100 100
```

3.8 MANIPULATING CHARACTER VECTORS:

Character vectors in R can be manipulated using various functions. Some key functions include `strsplit()`, `paste()`, `grep()`, and `gsub()`.

3.8.1 strsplit() Function

The `strsplit()` function splits character strings into substrings based on a specified delimiter.

Syntax:

```
strsplit(x, split)
```

Example:

```
text <- "apple,banana,grape"
split_text <- strsplit(text, ",")
print(split_text)
```

Output:

```
[[1]]
[1] "apple" "banana" "grape"
```

3.8.2 paste () Function

The `paste()` function concatenates character vectors.

Syntax:

```
paste(..., sep = " ", collapse = NULL)
```

Example:

```
words <- c("Hello", "World")
joined_text <- paste(words, collapse = " ")
print(joined_text)
```

Output:

```
[1] "Hello World"
```

3.8.3 grep() Function

The `grep()` function searches for patterns in a character vector and returns the matching indices.

Syntax:

```
grep(pattern, x, ignore.case = FALSE, value = FALSE)
```

Example:

```
text_vector <- c("apple", "banana", "cherry")
match_index <- grep("ban", text_vector)
print(match_index)
```

Output:

```
[1] 2
```

3.8.4 gsub() Function

The `gsub()` function replaces all occurrences of a pattern in a character string.

Syntax:

```
gsub(pattern, replacement, x)
```

Example:

```
text <- "Hello, World!"
new_text <- gsub("World", "R", text)
print(new_text)
```

Output:

```
[1] "Hello, R!"
```

3.9 FACTORS:

Factors are the data objects which are used to categorize the data and store it as levels.

They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like "Male", "Female" and True, False etc. They are useful in data analysis for statistical modeling.

Factors are created using the factor () function by taking a vector as input.

Example

```
# Create a vector as input.
```

```
data<c("East","West","East","North","North","East","West", "West","West","East","North")
```

```
print(data)
```

```
print(is.factor(data))
```

```
# Apply the factor function.
```

```
factor_data <- factor(data)
```

```
print(factor_data)
```

```
print(is.factor(factor_data))
```

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

```
[1] "East" "West" "East" "North" "North" "East" "West" "West" "West"
```

```
"East" "North"
```

```
[1] FALSE
```

```
[1] East West East North North East West West West East North
```

```
Levels: East North West
```

```
[1] TRUE
```


3.9.1 Factors in Data Frame

On creating any data frame with a column of text data, R treats the text column as categorical data and creates factors on it.

```
# Create the vectors for data frame.  
  
height <- c(132,151,162,139,166,147,122)  
  
weight <- c(48,49,66,53,67,52,40)  
  
gender <- c("male","male","female","female","male","female","male")  
  
# Create the data frame.  
  
input_data <- data.frame(height,weight,gender)  
  
print(input_data)  
  
# Test if the gender column is a factor.  
  
print(is.factor(input_data$gender))  
  
# Print the gender column so see the levels.  
  
print(input_data$gender)
```

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

```
height weight gender  
1 132 48 male  
2 151 49 male  
3 162 66 female  
4 139 53 female  
5 166 67 male  
6 147 52 female  
7 122 40 male
```

```
[1] TRUE
```

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

```
Levels: female male
```

Changing the Order of Levels

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

```
data <-
```

```
c("East", "West", "East", "North", "North", "East", "West", "West", "West", "East", "North")
```

```
# Create the factors
```

```
factor_data <- factor(data)
```

```
print(factor_data)
```

```
# Apply the factor function with required order of the level.
```

```
new_order_data <- factor(factor_data, levels = c("East", "West", "North"))
```

```
print(new_order_data)
```

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

```
[1] East West East North North East West West West East North
```

```
Levels: East North West
```

```
[1] East West East North North East West West West East North
```

```
Levels: East West North
```

3.9.2 Generating Factor Levels

We can generate factor levels by using the `gl()` function. It takes two integers as input which indicates how many levels and how many times each level.

Syntax

`gl(n, k, labels)`

Following is the description of the parameters used:

- `n` is a integer giving the number of levels.
- `k` is a integer giving the number of replications.
- `labels` is a vector of labels for the resulting factor levels.

Example

```
v <- gl(3, 4, labels = c("Tampa", "Seattle", "Boston"))
```

```
print(v)
```

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

```
Tampa Tampa Tampa Tampa Seattle Seattle Seattle Seattle Boston
```

```
[10] Boston Boston Boston
```

```
Levels: Tampa Seattle Boston
```

3.10 SUMMARY:

Vectors are fundamental data structures in R that store elements of the same type. They can be created using `c()`, manipulated using indexing, and operated on using vectorized arithmetic. Logical, character, and factor vectors offer additional flexibility for data handling. Understanding these concepts is crucial for data analysis and manipulation in R.

3.11 SELF ASSESSMENT QUESTIONS:

1. Define a vector in R and explain how it is created.
2. What are the different ways to generate sequences in R?
3. Explain how to modify subsets of a vector with examples.
4. Differentiate between `grep()` and `gsub()` functions.
5. How do factors help in categorical data representation?
6. Write R code to create a numeric vector, perform arithmetic operations, and extract a subset.
7. Explain the use of `paste()` function with an example.
8. How do logical vectors work in R? Provide an example.

9. Describe the importance of factors in data analysis.
10. Write an R program to split a character string using `strsplit()` and modify it using `gsub()`.

3.12 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Grolemund : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON 4

R MATRICES

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the importance of matrices
- Students should have a solid understanding about the concept of matrices
- The student will learn apply arithmetic, relational, logical, assignment, and miscellaneous operators to perform various computational tasks.
- Further, the student will be familiar with graphical facilities for data analysis available in R.

STRUCTURE

4.1 Introduction

4.2 Creating Matrices

4.3 Arithmetic Operators On Matrices

4.4 Accessing Elements of a matrix

4.5 Matrix Computations

4.5.1 Creating a Matrix

4.5. 2. Accessing Elements

4.5. 3. Matrix Operations

4.5.4. Combining Matrices

4.5. 5. Diagonal Matrix

4.5.6. Extract Diagonal Elements

4.5.7. Identity Matrix

4.5.8. Apply Functions to Rows/Columns

4.6 Forming Partitioned Matrices

4.6.1. Partitioned Matrices

4.6.2. cbind()

4.6.3. rbind()

4.6.4. Combining rbind() and cbind() Together

4.7 Conclusion

4.8 Self Assessment Questions

4.9 Further Readings

4.1 INTRODUCTION:

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations. A Matrix is created using the **matrix()** function.

Syntax

The basic syntax for creating a matrix in R is:

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

Following is the description of the parameters used:

- **Data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.
- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
- **dimname** is the names assigned to the rows and columns.

Example:

Create a matrix taking a vector of numbers as input

Elements are arranged sequentially by row.

```
M <- matrix(c(3:14), nrow=4, byrow=TRUE)
```

```
print(M)
```

Elements are arranged sequentially by column.

```
N <- matrix(c(3:14), nrow=4, byrow=FALSE)
```

```
print(N)
```

Define the column and row names.

```
rownames = c("row1", "row2", "row3", "row4")
```

```
colnames = c("col1", "col2", "col3")
```

```
P <- matrix(c(3:14), nrow=4, byrow=TRUE, dimnames=list(rownames, colnames))
```

```
print(P)
```

#When we execute the above code, it produces the following

output:

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

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

```
[2,] 6 7 8
```

```
[3,] 9 10 11
```

```
[4,] 12 13 14
```

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

```
[1,] 3 7 11
```

```
[2,] 4 8 12
```

```
[3,] 5 9 13
```

```
[4,] 6 10 14
```

```
 col1 col2 col3
```

```
row1 3 4 5
```

```
row2 6 7 8
```

```
row3 9 10 11
```

```
row4 12 13 14
```

4.2 CREATING MATRICES:

Matrix can be created using the `matrix()` function.

Dimension of the matrix can be defined by passing appropriate value for arguments `nrow` and `ncol`.

Providing value for both dimension is not necessary. If one of the dimension is provided, the other is inferred from length of the data.

```
>matrix(1:9, nrow = 3, ncol = 3)
     [,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
> # same result is obtained by providing only one dimension
>matrix(1:9, nrow = 3)
     [,1] [,2] [,3]
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
```

We can see that the matrix is filled column-wise. This can be reversed to row-wise filling by passing `TRUE` to the argument `byrow`.

```
>matrix(1:9, nrow=3, byrow=TRUE) # fill matrix row-wise
     [,1] [,2] [,3]
[1,]  1  2  3
[2,]  4  5  6
[3,]  7  8  9
```

In all cases, however, a matrix is stored in column-major order internally as we will see in the subsequent sections.

It is possible to name the rows and columns of matrix during creation by passing a 2 element list to the argument `dimnames`.

```
> x <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"), c("A","B","C")))
> x
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9
```

These names can be accessed or changed with two helpful functions `colnames()` and `rownames()`.

```
>colnames(x)
[1] "A" "B" "C"
```

```
>rownames(x)

[1] "X" "Y" "Z"

> # It is also possible to change names
>colnames(x) <- c("C1","C2","C3")
>rownames(x) <- c("R1","R2","R3")
> x
   C1 C2 C3
R1  1  4  7
R2  2  5  8
R3  3  6  9
```

Another way of creating a matrix is by using functions `cbind()` and `rbind()` as in column bind and row bind.

```
>cbind(c(1,2,3),c(4,5,6))
  [,1] [,2]
[1,]  1  4
[2,]  2  5
[3,]  3  6

>rbind(c(1,2,3),c(4,5,6))
  [,1] [,2] [,3]
[1,]  1  2  3
[2,]  4  5  6
```

Finally, you can also create a matrix from a vector by setting its dimension using `dim()`.

```
> x <- c(1,2,3,4,5,6)
> x
[1] 1 2 3 4 5 6
> class(x)

[1] "numeric"
> dim(x) <- c(2,3)
> x
  [,1] [,2] [,3]
[1,]  1  3  5
[2,]  2  4  6

> class(x)
[1] "matrix"
```

4.3 ACCESSING ELEMENTS OF A MATRIX:

Elements of a matrix can be accessed by using the column and row index of the element.

We consider the matrix P above to find the specific elements below.

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix.
P <- matrix(c(3:14), nrow=4, byrow=TRUE, dimnames=list(rownames, colnames))

# Access the element at 3rd column and 1st row.
print(P[1,3])

# Access the element at 2nd column and 4th row.
print(P[4,2])

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

# Access only the 3rd column.
print(P[,3])
```

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

```
[1] 5
[1] 13
col1 col2 col3
  6   7   8
row1 row2 row3 row4
  5   8  11  14
```

4.4 MATRIX COMPUTATIONS:

Various mathematical operations are performed on the matrices using the R operators.

The result of the operation is also a matrix.

The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

Matrix Addition & Subtraction

Create two 2x3 matrices.

```
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow=2)
print(matrix1)
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow=2)
print(matrix2)
```

```
# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)
# Subtract the matrices
result <- matrix1 - matrix2
```

```
cat("Result of subtraction","\n")
```

```
print(result)
```

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

```
[,1] [,2] [,3]  
[1,] 3 -1 2  
[2,] 9 4 6  
[,1] [,2] [,3]  
[1,] 5 0 3  
[2,] 2 9 4
```

Result of addition

```
[,1] [,2] [,3]  
[1,] 8 -1 5  
[2,] 11 13 10
```

Result of subtraction

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

Matrix Multiplication & Division

Create two 2x3 matrices.

```
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow=2)
```

```
print(matrix1)
```

```
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow=2)
```

```
print(matrix2)
```

Multiply the matrices.

```
result <- matrix1 * matrix2
```

```
cat("Result of multiplication","\n")
```

```
print(result)
```

Divide the matrices

```
result <- matrix1 / matrix2
```

```
cat("Result of division","\n")
```

```
print(result)
```

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

```
[,1] [,2] [,3]  
[1,] 3 -1 2  
[2,] 9 4 6  
[,1] [,2] [,3]  
[1,] 5 0 3  
[2,] 2 9 4
```

Result of multiplication

```
[,1] [,2] [,3]  
[1,] 15 0 6  
[2,] 18 36 24
```

Result of division

```
[,1] [,2] [,3]  
[1,] 0.6 -Inf 0.6666667
```

```
[2,] 4.5 0.4444444 1.5000000
```

4.5 MATRIX FACILITIES:

In **R programming**, matrices are a fundamental data structure used for mathematical and statistical computations. R provides a wide range of **matrix facilities** to create, manipulate, and perform operations on matrices efficiently. Below are some key matrix facilities in R, explained with examples:

4.5.1 Creating a Matrix

The `matrix()` function is commonly used to create matrices in R.

```
# Create a 3x3 matrix
A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3)
print(A)
```

Output:

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

By default, R fills the matrix **column-wise**.

4.5. 2. Accessing Elements

You can access specific elements, rows, or columns.

```
# Access element in 2nd row, 3rd column
A[2, 3]
# Access entire 1st row
A[1, ]
# Access entire 2nd column
A[, 2]
```

Output:

```
[1] 8
[1] 1 4 7
[1] 4 5 6
```

4.5. 3. Matrix Operations

(a) Addition & Subtraction

```
B <- matrix(1:9, nrow = 3)
```

```
# Matrix Addition
C <- A + B
# Matrix Subtraction
D <- A - B
```

```
print(C)
print(D)
```

Output:

```
[,1] [,2] [,3]
[1,]  2  8 14
[2,]  4 10 16
[3,]  6 12 18

[,1] [,2] [,3]
[1,]  0  0  0
[2,]  0  0  0
[3,]  0  0  0
```

(b) Matrix Multiplication**(i) Element-wise Multiplication (*)**

```
E <- A * B
print(E)
```

Output:

```
[,1] [,2] [,3]
[1,]  1 16 49
[2,]  4 25 64
[3,]  9 36 81
```

(ii) Matrix Product (%*%)

```
F <- A %*% B
print(F)
```

Output:

```
[,1] [,2] [,3]
[1,] 30 66 102
[2,] 36 81 126
[3,] 42 96 150
```

(c) Transpose of a Matrix

```
t_A <- t(A)
print(t_A)
```

Output:

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

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

(d) Determinant of a Matrix

```
det_A <- det(A)
print(det_A)
```

Output:

```
[1] 0
```

4.5.4. Combining Matrices**(a) Row Binding**

```
A_new <- rbind(A, c(10, 11, 12))
print(A_new)
```

Output:

```
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[4,]   10   11   12
```

(b) Column Binding

```
B_new <- cbind(A, c(10, 11, 12))
print(B_new)
```

Output:

```
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

4.5.5. Diagonal Matrix

```
diag_matrix <- diag(c(1, 2, 3))
print(diag_matrix)
```

Output:

```
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

4.5.6. Extract Diagonal Elements

```
diag_elements <- diag(A)
```

```
print(diag_elements)
```

Output:

```
[1] 1 5 9
```

4.5.7. Identity Matrix

```
identity_matrix <- diag(3)
print(identity_matrix)
```

Output:

```
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

4.5.8. Apply Functions to Rows/Columns

(a) Row Sums

```
row_sums <- rowSums(A)
print(row_sums)
```

Output:

```
[1] 12 15 18
```

(b) Column Means

```
col_means <- colMeans(A)
print(col_means)
```

Output:

```
[1] 2 5 8
```

(c) Using apply()

```
# Sum of each row
apply(A, 1, sum)
```

```
# Mean of each column
apply(A, 2, mean)
```

Output:

```
[1] 12 15 18
```

```
[1] 2 5 8
```

4.5.9. Checking Matrix Dimensions

```
dim(A)
nrow(A)
ncol(A)
```

Output:

```
[1] 3 3
```

```
[1] 3
```

```
[1] 3
```

4.5.10. Reshaping a Matrix

```
reshape_A <- matrix(A, nrow = 1)
print(reshape_A)
```

Output:

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

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

Summary Table

Facility	R Function / Operator	Description
Matrix Creation	matrix()	Create a matrix
Access Elements	[]	Access specific elements, rows, columns
Addition/Subtraction	+, -	Element-wise addition and subtraction
Element-wise Multiplication	*	Element-wise multiplication
Matrix Product	%*%	Matrix multiplication
Transpose	t()	Transpose of a matrix
Determinant	det()	Determinant of a square matrix
Inverse	solve()	Inverse of a square matrix
Combine Rows/Columns	rbind(), cbind()	Bind matrices by rows or columns
Diagonal Matrix	diag()	Create a diagonal matrix
Row/Column Sums and Means	rowSums(), colMeans()	Compute sums and means
Apply Function to Rows/Cols	apply()	Apply a function across rows or columns

Facility	R Function / Operator	Description
Matrix Dimensions	dim(), nrow(), ncol()	Get matrix dimensions

These matrix facilities allow R users to perform efficient data manipulation and mathematical computations, making matrices a powerful tool in **data analysis, statistical modeling, and machine learning**.

4.6 FORMING PARTITIONED MATRICES:

4.6.1. Partitioned Matrices

A **partitioned matrix** is a matrix that is divided into smaller sub-matrices or blocks. This is often done to simplify complex matrix operations or organize data efficiently.

Simple Example in R:

```
# Defining sub-matrices (blocks)
A11 <- matrix(c(1, 2, 3, 4), nrow = 2)
A12 <- matrix(c(5, 6), nrow = 2)
A21 <- matrix(c(7, 8), nrow = 1)
A22 <- matrix(c(9), nrow = 1)

# Combining the blocks to form a partitioned matrix
A_upper <- cbind(A11, A12) # Combining A11 and A12 horizontally
A_lower <- cbind(A21, A22) # Combining A21 and A22 horizontally

Partitioned_Matrix <- rbind(A_upper, A_lower) # Combining the two rows vertically

print(Partitioned_Matrix)
```

Output:

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

4.6.2. cbind()

The cbind() function in R is used to **combine two or more matrices, vectors, or data frames by columns**.

Syntax:

```
cbind(matrix1, matrix2, ...)
```

Example 1: Combining Matrices by Columns

```
A <- matrix(1:4, nrow = 2)
```



```
B <- matrix(5:8, nrow = 2)
```

```
C <- cbind(A, B)
print(C)
```

Output:

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

Each matrix is attached **column-wise**.

Example 2: Combining Vectors by Columns

```
v1 <- c(1, 2, 3)
v2 <- c(4, 5, 6)
```

```
C <- cbind(v1, v2)
print(C)
```

Output:

```
      v1 v2
[1,]  1  4
[2,]  2  5
[3,]  3  6
```

4.6.3. rbind() – Row Binding

The `rbind()` function in R is used to **combine two or more matrices, vectors, or data frames by rows**.

Syntax:

```
rbind(matrix1, matrix2, ...)
```

Example 1: Combining Matrices by Rows

```
A <- matrix(1:4, nrow = 2)
B <- matrix(5:8, nrow = 2)
```

```
D <- rbind(A, B)
print(D)
```

Output:

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

Each matrix is attached **row-wise**.

Example 2: Combining Vectors by Rows

```
v1 <- c(1, 2, 3)
```

```
v2 <- c(4, 5, 6)
```

```
D <- rbind(v1, v2)
```

```
print(D)
```

Output:

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

```
v1   1   2   3
```

```
v2   4   5   6
```

4.6.4. Combining rbind() and cbind() Together

Sometimes, you may need to use both functions to form **partitioned matrices**.

Example:

```
A1 <- matrix(c(1, 2, 3, 4), nrow = 2)
```

```
A2 <- matrix(c(5, 6), nrow = 2)
```

```
B1 <- matrix(c(7, 8), nrow = 1)
```

```
B2 <- matrix(c(9), nrow = 1)
```

```
Upper <- cbind(A1, A2)
```

```
Lower <- cbind(B1, B2)
```

```
Partitioned_Matrix <- rbind(Upper, Lower)
```

```
print(Partitioned_Matrix)
```

Key Points to Remember:

Function	Purpose	Example Use Case
cbind()	Combine matrices/vectors by columns	Add new features (columns) to data
rbind()	Combine matrices/vectors by rows	Add new observations (rows) to data
Partitioned Matrix	Divide a matrix into blocks for efficient computation	Handling large data blocks in sections

4.7 SUMMARY:

Matrices are fundamental data structures used in mathematical computations and data analysis. Understanding matrix creation, manipulation, and operations is essential for working efficiently with numerical data. Functions like `matrix()`, `cbind()`, `rbind()`, `diag()`, and `apply()` simplify matrix handling and computations.

4.8 SELF ASSESSMENT QUESTIONS:

1. What is a matrix in R? How is it different from a data frame?
2. What is the difference between `cbind()` and `rbind()`?
3. How do you create an identity matrix in R?
4. Create a 3x3 matrix with numbers from 1 to 9, and:
 - a) Extract the second row.
 - b) Find the transpose of the matrix.
 - c) Extract the diagonal elements.
5. Given two matrices A and B of the same dimension, perform element-wise addition, multiplication, and matrix multiplication.
6. Create a diagonal matrix with values (2, 4, 6) along the diagonal.
7. Use the `apply()` function to compute the sum of each row and the mean of each column of a matrix.
8. Combine two matrices both row-wise and column-wise.

4.9 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Golemund : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON 5

R-LISTS

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the importance of Lists
- Students should have a solid understanding about the concept of Lists
- Further, the student will be familiar with data frames.

STRUCTURE

5.1 Creating a List

5.2 Naming List Elements

5.3 Accessing List Elements

5.4 Manipulating List Elements

5.5 Merging Lists

5.6 Converting List to Vector

5.7 Data Frames

5.7.1 Create Data Frame

5.8 Extract Data from Data Frame

5.8.1 Expand Data Frame

5.8.2 Add Row

5.9 Attach () data Frames

5.10 Conclusion

5.11 Self Assessment Questions

5.12 Further Readings

5.1 CREATING A LIST:

Lists are the R objects which contain elements of different types like - numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

Following is an example to create a list containing strings, numbers, vectors and a logical values

```
# Create a list containing strings, numbers, vectors and a logical values.  
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)  
print(list_data)
```

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

```
[[1]]  
[1] "Red"  
[[2]]
```

```
[1] "Green"
[[3]]
[1] 21 32 11
[[4]]
[1] TRUE
[[5]]
[1] 51.23
[[6]]
[1] 119.1
```

5.2 NAMING LIST ELEMENTS:

The list elements can be given names and they can be accessed using these names.

Create a list containing a vector, a matrix and a list.

```
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow=2),
list("green",12.3))
```

Give names to the elements in the list.

```
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
```

Show the list.

```
print(list_data)
```

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

```
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"
$A_Matrix
[,1] [,2] [,3]
[1,] 3 5 -2
[2,] 9 1 8
$A_Inner_list
$A_Inner_list[[1]]
[1] "green"
$A_Inner_list[[2]]
[1] 12.3
```

5.3 ACCESSING LIST ELEMENTS:

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

We continue to use the list in the above example:

Create a list containing a vector, a matrix and a list.

```
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow=2),
list("green",12.3))
```

Give names to the elements in the list.

```
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
```

Access the first element of the list.

```
print(list_data[1])
```

Access the third element. As it is also a list, all its elements will be printed.

```
print(list_data[3])
```

```
# Access the list element using the name of the element.
print(list_data$A_Matrix)
When we execute the above code, it produces the following result:
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"
$A_Inner_list
$A_Inner_list[[1]]
[1] "green"
$A_Inner_list[[2]]
[1] 12.3
[,1] [,2] [,3]
[1,] 3 5 -2
[2,] 9 1 8
```

5.4 MANIPULATING LIST ELEMENTS:

We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

Create a list containing a vector, a matrix and a list.

```
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow=2),
list("green",12.3))
# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
# Add element at the end of the list.
list_data[4] <- "New element"
print(list_data[4])
# Remove the last element.
list_data[4] <- NULL
# Print the 4th Element.
print(list_data[4])
# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

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

```
[[1]]
[1] "New element"
$
NULL
$`A Inner list`
[1] "updated element"
```

5.5 MERGING LISTS:

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")
# Merge the two lists.
merged.list <- c(list1,list2)
```

```
# Print the merged list.
```

```
print(merged.list)
```

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

```
[[1]]  
[1] 1  
[[2]]  
[1] 2  
[[3]]  
[1] 3  
[[4]]  
[1] "Sun"  
[[5]]  
[1] "Mon"  
[[6]]  
[1] "Tue"
```

5.6 CONVERTING LIST TO VECTOR:

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion, we use the **unlist()** function. It takes the list as input and produces a vector.

```
# Create lists.
```

```
list1 <- list(1:5)
```

```
print(list1)
```

```
list2 <- list(10:14)
```

```
print(list2)
```

```
# Convert the lists to vectors.
```

```
v1 <- unlist(list1)
```

```
v2 <- unlist(list2)
```

```
R Programming
```

```
59
```

```
print(v1)
```

```
print(v2)
```

```
# Now add the vectors
```

```
result <- v1+v2
```

```
print(result)
```

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

```
[[1]]  
[1] 1 2 3 4 5  
[[1]]  
[1] 10 11 12 13 14  
[1] 1 2 3 4 5  
[1] 10 11 12 13 14  
[1] 11 13 15 17 19
```

5.7 DATA FRAMES:

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

5.7.1 Create Data Frame

```
# Create the data frame.
emp.data <- data.frame(
  emp_id = c(1:5),
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
  salary = c(623.3,515.2,611.0,729.0,843.25),
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),
  stringsAsFactors=FALSE
)
```

Print the data frame.

```
print(emp.data)
```

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

```
emp_id emp_name salary start_date
1 1 Rick 623.30 2012-01-01
2 2 Dan 515.20 2013-09-23
3 3 Michelle 611.00 2014-11-15
4 4 Ryan 729.00 2014-05-11
5 5 Gary 843.25 2015-03-27
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

Create the data frame.

```
emp.data <- data.frame(
  emp_id = c(1:5),
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
  salary = c(623.3,515.2,611.0,729.0,843.25),
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),
  stringsAsFactors=FALSE
)
```

Get the structure of the data frame.

```
str(emp.data)
```

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

```
'data.frame': 5 obs. of 4 variables:
```

```
$ emp_id : int 1 2 3 4 5
```



```
$ emp_name : chr "Rick" "Dan" "Michelle" "Ryan" ...  
$ salary : num 623 515 611 729 843  
$ start_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-
```

Summary of Data in Data Frame:

The statistical summary and nature of the data can be obtained by applying **summary()** function.

```
# Create the data frame.  
emp.data <- data.frame(  
  emp_id = c(1:5),  
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),  
  salary = c(623.3,515.2,611.0,729.0,843.25),  
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-  
11","2015-03-27")),  
  stringsAsFactors=FALSE  
)  
# Print the summary.  
print(summary(emp.data))
```

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

```
emp_id emp_name salary start_date  
Min. :1 Length:5 Min. :515.2 Min. :2012-01-01  
1st Qu.:2 Class :character 1st Qu.:611.0 1st Qu.:2013-09-23  
Median :3 Mode :character Median :623.3 Median :2014-05-11  
Mean :3 Mean :664.4 Mean :2014-01-14  
3rd Qu.:4 3rd Qu.:729.0 3rd Qu.:2014-11-15  
Max. :5 Max. :843.2 Max. :2015-03-27
```

5.8 EXTRACT DATA FROM DATA FRAME:

Extract specific column from a data frame using column name.

```
# Create the data frame.  
emp.data <- data.frame(  
  emp_id = c(1:5),  
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),  
  salary = c(623.3,515.2,611.0,729.0,843.25),  
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-  
11","2015-03-27")),  
  stringsAsFactors=FALSE  
)  
  
# Extract Specific columns.  
result <- data.frame(emp.data$emp_name,emp.data$salary)  
print(result)
```

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

```
emp.data.emp_name emp.data.salary
```

```
1 Rick 623.30
```

```
2 Dan 515.20
```

```
3 Michelle 611.00
```

```
4 Ryan 729.00
```

```
5 Gary 843.25
```

Extract the first two rows and then all columns

```
# Create the data frame.
```

```
emp.data <- data.frame(
```

```
emp_id = c(1:5),
```

```
emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
```

```
salary = c(623.3,515.2,611.0,729.0,843.25),
```

```
start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),stringsAsFactors=FALSE)
```

```
# Extract first two rows.
```

```
result <- emp.data[1:2,]
```

```
print(result)
```

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

```
emp_id emp_name salary start_date
```

```
1 1 Rick 623.3 2012-01-01
```

```
1 2 Dan 515.2 2013-09-23
```

Extract 3rd and 5th row with 2nd and 4th column

```
# Create the data frame.
```

```
emp.data <- data.frame(
```

```
emp_id = c(1:5),
```

```
emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
```

```
salary = c(623.3,515.2,611.0,729.0,843.25),
```

```
start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),stringsAsFactors=FALSE)
```

```
# Extract 3rd and 5th row with 2nd and 4th column.
```

```
result <- emp.data[c(3,5),c(2,4)]
```

```
print(result)
```

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

```
emp_name start_date
```

```
3 Michelle 2014-11-15
```

```
5 Gary 2015-03-27
```

5.8.1 Expand Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

```
# Create the data frame.
emp.data <- data.frame(
  emp_id = c(1:5),
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
  salary = c(623.3,515.2,611.0,729.0,843.25),
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),stringsAsFactors=FALSE)
```

```
# Add the "dept" column.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)
```

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

```
emp_id emp_name salary start_date dept
1 1 Rick 623.30 2012-01-01 IT
2 2 Dan 515.20 2013-09-23 Operations
3 3 Michelle 611.00 2014-11-15 IT
4 4 Ryan 729.00 2014-05-11 HR
5 5 Gary 843.25 2015-03-27 Finance
```

5.8.2 Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function. In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the first data frame.
emp.data <- data.frame(
  emp_id = c(1:5),
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
  salary = c(623.3,515.2,611.0,729.0,843.25),
  start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),
  dept=c("IT","Operations","IT","HR","Finance"),
  R Programming
  78
  stringsAsFactors=FALSE
)
# Create the second data frame
emp.newdata <- data.frame(
  emp_id = c(6:8),
  emp_name = c("Rasmi","Pranab","Tusar"),
  salary = c(578.0,722.5,632.8),
  start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),
```

```
dept = c("IT","Operations","Fianance"),
stringsAsFactors=FALSE
)
# Bind the two data frames.
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

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

```
emp_id emp_name salary start_date dept
1 1 Rick 623.30 2012-01-01 IT
2 2 Dan 515.20 2013-09-23 Operations
3 3 Michelle 611.00 2014-11-15 IT
4 4 Ryan 729.00 2014-05-11 HR
5 5 Gary 843.25 2015-03-27 Finance
6 6 Rasmi 578.00 2013-05-21 IT
7 7 Pranab 722.50 2013-07-30 Operations
8 8 Tusar 632.80 2014-06-17 Fianance
```

5.9 ATTACH DATA FRAMES:

The `attach()` function offers a solution to this: it takes a data frame as an argument and places it in the search path at position 2.

So unless there are variables in position 1 that are exactly the same as the ones from the data frame that you have inputted, the variables are considered as variables that can be immediately called on.

Note that the search path is in fact the order in which R accesses files. You can look this up by entering the `search()` function.

```
# Look up the search path
search()
# Attach the `writers_df`

attach(writers_df)

# Alternatively, use `with()` to attach `writers_df`

with(writers_df, c("Age.At.Death", "Age.As.Writer", "Name", "Surname", "Gender",
"Death"))

# Return `writers_df`

writers_df

> # Look up the search path

>search()

[1] ".GlobalEnv"      "package:stats"   "package:graphics"
[4] "package:grDevices" "package:utils"   "package:datasets"
[7] "package:methods" "Autoloads"       "package:base"
```

```
> # Attach the `writers_df`
```

```
> attach(writers_df)
```

The following objects are masked `_by_`.GlobalEnv:

Age.As.Writer, Age.At.Death, Death, Gender, Name, Surname

The following objects are masked from `writers_df` (pos = 3):

Age.As.Writer, Age.At.Death, Death, Gender, Name, Surname

```
> # Alternatively, use `with()` to attach `writers_df`
```

```
> with(writers_df, c("Age.At.Death", "Age.As.Writer", "Name", "Surname", "Gender",
"Death"))
```

```
[1] "Age.At.Death" "Age.As.Writer" "Name"          "Surname"
```

```
[5] "Gender"       "Death"
```

```
> # Return `writers_df`
```

```
> writers_df
```

	Age.At.Death	Age.As.Writer	Name	Surname	Gender	Death
1	22	16	John	Doe	MALE	2015-05-10
2	40	18	Edgar	Poe	MALE	1849-10-07
3	72	36	Walt	Whitman	MALE	1892-03-26
4	41	36	Jane	Austen	FEMALE	1817-07-18

Detach ()

`detach ()` function just reverses the function of `attach()`. It removes the specified variable from the global environment, so you cannot access the variable directly as before.

5.10 SUMMARY:

Understanding lists and data frames is crucial for efficient data management and analysis in R. Lists allow for storing diverse types of data in a single structure, whereas data frames are fundamental for handling tabular data, which is widely used in data analysis. Mastery of these data structures equips students with the capability to manipulate, merge, and transform data efficiently, laying the groundwork for more advanced data analysis and statistical operations in R.

5.11 SELF ASSESSMENT QUESTIONS:

1. What is a list in R, and how is it different from a vector?
2. Explain the process of naming elements in a list.
3. How can you access and manipulate elements in a list?

4. Describe the steps involved in merging two lists in R.
5. What are data frames in R? How are they different from matrices?
6. Write an R script to create a list containing a vector, a matrix, and a character string. Access the second element of the list.
7. Create a data frame containing the names, ages, and marks of five students. Extract the 'marks' column from the data frame.
8. Write a code snippet to add a new row to an existing data frame.

5.12 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Grolemund : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON 6

READING AND GETTING DATA INTO R USING FILES

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the Process of Reading and Importing Data into R.
- Students should have a solid understanding about the reading and getting data into r using files.
- The student will learn a reading and getting data into r using files.
- Further, the student will be familiar reading and getting data into r using files.

STRUCTURE:

6.1 Reading and getting data into R using files

6.2 Using the combine Command for Making Data

6.3 Reading Bigger data Files

6.4 Alternative Commands for Reading Data in R

6.5 Saving Data using files

6.5.1 Save () command

6.5.2 Load () command

6.6 Writing data using files

6.6.1 Write. table ()

6.6.2 File. choose()

6.7 Reading a data using files

6.8 Conclusion

6.9 Self Assessment Questions

6.10 Further Readings

6.1 READING AND GETTING DATA INTO R USING FILES:

So far you have looked at some simple math. More often you will have sets of data to examine (that is, samples) and will want to create more complex series of numbers to work on. You cannot perform any analyses if you do not have any data so getting data into R is a very important task. This next section focuses on ways to create these complex samples and get data into R, where you are able to undertake further analyses.

6.3 USING THE COMBINE COMMAND FOR MAKING DATA:

The simplest way to create a sample is to use the `c()` command. You can think of this as short for combine or concatenate, which is essentially what it does. The command takes the following form:

c(item.1, item.2, item.3, item .n)

Everything in the parentheses is joined up to make a single item. More usually you will assign the joined-up items to a named object:

sample.name = c(item.1, item.2, item.3, item.n)

This is much like you did when making simple result objects, except now your sample objects consist of several bits rather than a single value.

6.3 READING BIGGER DATA FILES:

The scan() command is helpful to read a simple vector. More often though, you will have complicated data files that contain multiple items (in other words two-dimensional items containing both rows and columns). Although it is possible to enter large amounts of data directly into R, it is more likely that you will have your data stored in a spreadsheet. When you are sent data items, the spreadsheet is also the most likely format you will receive. R provides the means to read data that is stored in a range of text formats, all of which the spreadsheet is able to create. The read.csv () command.

In most cases you will have prepared data in a spreadsheet. Your dataset could be quite large and it would be tedious to use the clipboard. When you have more complex data it is better to use a new command—read.csv ():

names to **read.csv()**

As you might expect, this looks for a CSV file and reads the enclosed data into R. You can add a variety of additional instructions to the command. For example:

read.csv(file, sep = ',', header = TRUE, row.names)

You can replace the file with any filename as before. By default the separator is set to a comma but you can alter this if you need to. This command expects the data to be in columns, and for each column to have a helpful name. The instruction header = TRUE, the default, reads the first row of the CSV file and sets this as a name for each column. You can override this with header = FALSE.

The row.names part allows you to specify row names for the data; generally this will be a column in the dataset (the first one is most usual and sensible). You can set the rownames to be one of the columns by setting row.names = n, where n is the column number. Some simple example data are shown in Table 2-2. Here you can see two columns; each one is a variable. The first column is labelled abund; this is the abundance of some water-living organism. The second column is labelled flow and represents the flow of water where the organism was found.

Simple Data from a Two Column Spreadsheet

ABUND	FLOW
9	2
25	3
15	5
2	9

14	14
25	24
24	29
47	34

In this case there are only two columns and it would not take too long to use the `scan()` command to transfer the data into R. However, it makes sense to keep the two columns together and import them to R as a single entity. To do so, perform the following steps:

1. If you have a file saved in a proprietary format (for example, XLS), save the data as a CSV File instead.

2. Now assign the file a sensible name and use the `read.csv()` command as follows:

```
>fw = read.csv(file.choose())
```

3. Select the file from the browser window. If you are using Linux, the filename must be typed in full. Because the `read.csv()` command is expecting the data to be separated with commas, you do not need to specify that. The data has headings and because this is also the default, you do not need to tell R anything else.

4. `>fw`

Abund flow

```
1 9 2
2 25 3
3 15 5
4 2 9
5 14 14
6 25 24
7 24 2
8 47 34
```

You can see that each row is labelled with a simple index number; these have no great relevance but can be useful when there are a lot of data.

In the general, the `read.csv()` command is pretty useful because the CSV format is most easily produced by a wide variety of computer programs, spreadsheets, and is eminently portable. Using CSV means that you have fewer options to type into R and consequently less typing.

6.4 ALTERNATIVE COMMANDS FOR READING DATA IN R:

There are many other formats besides CSV in which data can exist and in which other characters, including spaces and tabs, can separate data. Consequently, the `read.table()` command is actually the basic R command for reading data. It enables you to read most formats of plain-text data. However, R provides variants of the command with certain defaults to make it easier when specifying some common data formats, like `read.csv()` for CSV files. Since most data is CSV though, the `read.csv()` is the most useful of these variants. But you may run into alternative formats, and the following list outlines:

The basic `read.table()` as well as other commands you can use to read various types of data:

In this case you have to specify the additional instructions explicitly. The defaults are set to `header = FALSE`, `sep = " "` (a single space), and `dec = "."`,

For example:

```
data1 data2 data3
1      2      4
```

In the following example the data are separated by simple spaces. The `read.table()` command is a more generalized command and you could use this at any time to read your data.

```
4 5 3
3 4 5
3 6 6
4 5 9
```

```
>my.ssv = read.table(file.choose(), header = TRUE)
```

```
>my.ssv = read.csv(file.choose(), sep = ' ')
```

The next example shows data separated by tabs. If you have tab-separated values you can use the `read.delim()` command. In this command R assumes that you still have column heading names but this time the separator instruction is set to `sep = "\t"` (a tab character) by default:

```
data1 data2 data3
1      2      4
4      5      3
3      4      5
3      6      6
4      5      9
```

```
>my.tsv = read.delim(file.choose())
```

```
>my.tsv = read.csv(file.choose(), sep = '\t')
```

```
>my.tsv = read.table(file.choose(), header = TRUE, sep = '\t')
```

The next example also shows data separated by tabs. In some countries the decimal point character is not a period but a comma, and a semicolon often separates data values. If you have a file like this you can use another variant, `read.csv2()`. Here the defaults are set to `sep = ";"`, `header = TRUE`, and `dec = ","`.

```
day data1 data2 data3
mon 1 2 4
tue 4 5 3
wed 3 4 5
thu 3 6 6
fri 4 5 9
```

```
>my.list = read.delim(file.choose(), row.names = 1)
```

```
>my.list = read.csv(file.choose(), row.names = 1, sep = '\t')
```

```
>my.list = read.table(file.choose(), row.names = 1, header = TRUE, sep = '\t')
```

6.4 SAVING DATA USING FILES:

It is not really convenient to quit R every time you want to save your work to disk. Sometimes, if you are working on several items or projects at a time you may even want to save these separately.

Fortunately, R provides a solution; you can save individual objects, or indeed all the objects, to disk at any time using the `save()` command.

6.5.1 Save () command:

The save() command operates like so:

save(list, file = 'filename')

You need to specify a filename and it must be in quotes. The file will be saved to the current working directory by default. The list instruction can be in one of two forms: you can simply type the names of the objects you want to save separated with commas or you can link to a list of names created by some other means. Look at the examples that follow:

```
>save(bf, bf.lm, bf.beta, file = 'Desktop/butterfly.RData')  
>save(list=ls(pattern='^bf'),file = 'Desktop/butterfly.RData')
```

In the first case three objects were specified (bf, bf.lm, and bf.beta), and in the second example the ls() command was used to create a list of objects beginning with bf. In both cases, the output file was saved to the Desktop folder rather than the default.

6.5.2 Load () command:

When you save a file to disk, R saves the data in a binary format. This means that the file cannot be read by a regular word processor or text editor. You can read one of these binary files from within R using the load() command:

load(file = 'filename.Rdata')

You need to put the filename in quotes (single or double, as long as the pair match) and remember to include the extension. The usual extension to use is .Rdata. If the file is not in your working directory the full path must be entered (all in the quotes). Alternatively, you can use the file.choose() instruction and select your file if you are using Windows or Macintosh operating systems.

load(file = file.choose())

Once the file is read, any data objects that were saved in it are available and can be seen by using the ls() command.

It is also possible to load binary data items directly from your operating system by double-clicking the file you want. In Windows and Macintosh systems the .Rdata file extension should automatically become associated with R when you install the program. This is a useful way to open R because the only data that is loaded will be the data within the .Rdata file.

6.6 WRITING DATA USING FILES:

If you have a vector, you can use the write() command. The basic form of the command is the following:

write(x, file = "data", ncolumns = if(is.character(x)) 1 else 5, sep = " ")

This looks a bit complicated because the ncolumns = part contains a conditional statement. This is because the if() statement creates a file with multiple columns according to the type of data. If the data are text, a single column is created. If the data are numeric, five columns are

created (you can alter the number of columns). The items are separated by a space by default; you can change this by altering the `sep =` instruction. For example, the following code snippet contains a list of numbers.

The `write()` command sees that these are numeric and creates five columns by default. The data are separated with commas.

```
> data7
[1] 23.0 17.0 12.5 11.0 17.0 12.0 14.5 9.0 11.0 9.0 12.5 14.5 17.0 8.0 21.0

> write(data7, file = 'Desktop/data7.txt', sep = ',')
```

The resulting file looks like the following if viewed in a basic text editor:

```
23,17,12.5,11,17
12,14. ,9,11,9
12.5,14.5,17,8,21
```

If you want to create a single column you set the `ncolumns =` instruction to 1. If you want to create a single row you need to know how many items there are and set the number of columns to this value. You can do this automatically like so:

```
> write(data7, file = 'Desktop/data7.txt', sep = ',', ncolumns = length(data7))
```

Here a command called `length()` was used, which determines how “long” the vector of data is. The resulting file looks like the following:

```
23,17,12.5,11,17,12,14.5,9,11,9,12.5,14.5,17,8,21
```

6.6.1 Write. `table()`:

If you have a matrix object or a data frame, you need to use the `write.table()` command. The basic command has various instructions that can be set as follows:

```
write.table(mydata, file = 'filename', row.names = TRUE, sep = ' ', col.names=TRUE)
```

If you want to make a CSV file, you could use the alternative `write.csv()` command. This is essentially the same but the default settings are slightly different:

```
write.csv(mydata, file = 'filename', row.names = TRUE, sep = ',', col.names = TRUE)
```

The `write.table()` and `write.csv()` commands are most useful to save complex data items that contain multiple columns.

6.6.2 File. `choose()`:

The `file.choose()` instruction is useful because you can select files from different directories without having to alter the working directory or type the names in full.

Using the `file.choose()` instruction does not work on Linux operating systems.

6.7 READING A DATA USING FILES:

To read a file with the `scan()` command you simply add `file = 'filename'` to the command. For example:

```
> data6 = scan(file = 'test data.txt')
Read 15 items
> data6
[1] 23.0 17.0 12.5 11.0 17.0 12.0 14.5 9.0 11.0 9.0 12.5 14.5 17.0 8.0 21.0
```

In this example the data file is called `test data.txt`, which is plain text, and the numerical values are separated by spaces. Note that the filename must be enclosed in quotes (single or double). Of course you can use the `what =` and `sep =` instructions as appropriate.

6.8 SUMMARY:

R provides a flexible and diverse set of tools for data input and output, accommodating various file types and dataset sizes. When working with large datasets, functions like `fread()` from **data.table** and `read_csv()` from **readr** offer significantly better performance compared to base R functions. The `save()` and `load()` commands enable efficient storage and retrieval of R objects, helping to preserve workspaces for future use. Additionally, the `file.choose()` function allows users to interactively select files, eliminating the need to manually specify file paths.

6.9 SELF ASSESSMENT QUESTIONS:

1. What is the purpose of the `read.csv()` function in R? Provide an example.
2. Explain the use of the `c()` function in R. How is it different from `read.table()`?
3. How would you optimize reading a large dataset in R? Mention any two functions.
4. What are the alternatives to `read.table()` in R for reading data? List at least three.
5. Explain the difference between `save()` and `write.table()` in R.
6. Write an R script to:
7. Read a CSV file.
8. Display the first few rows.
9. Save the data to an `.RData` file.
10. How can `file.choose()` simplify the process of reading and writing data in R?
11. What is the difference between `load()` and `read.csv()` functions in R?

6.10 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): *Beginning R – The Statistical Programming Language*, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): *An Introduction to R*
- 3) J.P. Lander(2014): *R for Everyone*, Pearson Publications
- 4) Garrett Golemund : *Hands-On Programming with R*
- 5) Michael J. Crawley: *The R Book*

LESSON -7

CONTROL STATEMENTS

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the importance of Decision Making Statements
- Students should have a solid understanding about the Decision Making Statements
- .To know the concepts of Decision Making Statements
- To acquire knowledge about Control Statements

STRUCTURE

7.1 Introduction

7.2 if-statements

7.3 if-else statements

7.4 if-else-if statement

7.5 nested-if statements

7.6 switch statement

7.7 Conclusion

7.8 Self Assessment Questions

7.9 Further Readings

7.1 INTRODUCTION:

Decision-making statements, also known as conditional statements, are a fundamental component of programming languages, including R. They enable programmers to make decisions based on conditions, allowing code to execute different paths or actions depending on specific criteria.

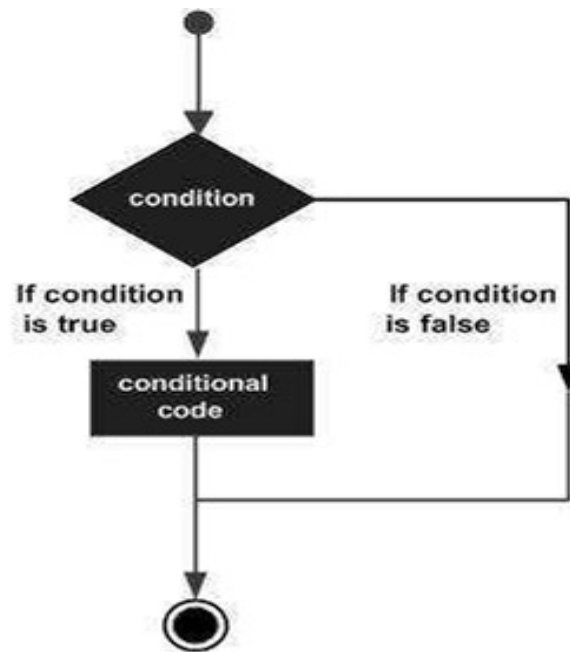
Decision-making statements are essential in programming because they:

1. **Enable conditional execution:** Decision-making statements allow code to execute specific blocks of code only when certain conditions are met.
2. **Improve code flexibility:** Decision-making statements enable code to adapt to different situations and inputs.
3. **Enhance code readability:** Decision-making statements make code more readable by providing a clear structure and organization.

Key Elements of Decision-Making Statements

1. **Condition:** a logical expression that evaluates to TRUE or FALSE
2. **Action:** a block of code that executes when the condition is TRUE
3. **Alternative action:** a block of code that executes when the condition is FALSE

By mastering decision-making statements, you'll be able to write more flexible, efficient, and effective code.



R provides the following types of decision making statements. Click the following links to check their detail.

Statement	Description
if statement	An if statement consists of a Boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
switch statement	A switch statement allows a variable to be tested for equality against a list of values.

7.2 IF STATEMENT:

The conditional if statement is used to test an expression. If the test_expression is TRUE, the statement gets executed. But if it's FALSE, nothing happens.

```
# syntax of if statement
if (test_expression) {
  statement
}
```

Example:

```
x<-c(8,3,-2,5)
# without curly braces
if(any(x<0))print("x contains negative numbers")
```

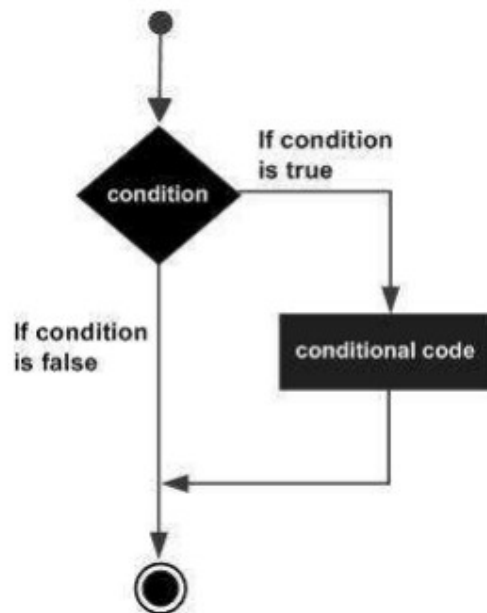
```
## [1] "x contains negative numbers"

# with curly braces produces same result
if(any(x<0)){
  print("x contains negative numbers")
}
## [1] "x contains negative numbers"

# an if statement in which the test expression is FALSE

# does not produce any output
y<-c(8,3,2,5)

if(any(y<0)){
  print("y contains negative numbers")
}
```



7.3 IF...ELSE STATEMENT:

The conditional if...else statement is used to test an expression similar to the if statement. However, rather than nothing happening if the test_expression is FALSE, the else part of the function will be evaluated.

```
# syntax of if...else statement
if(test_expression){
  statement1
}else{
  statement2
}
```


Example:

This test results in statement 1 being executed

```
x<-c(8,3,-2,5)
if(any(x<0)){
  print("x contains negative numbers")
}else{
  print("x contains all positive numbers")
}
```

Output

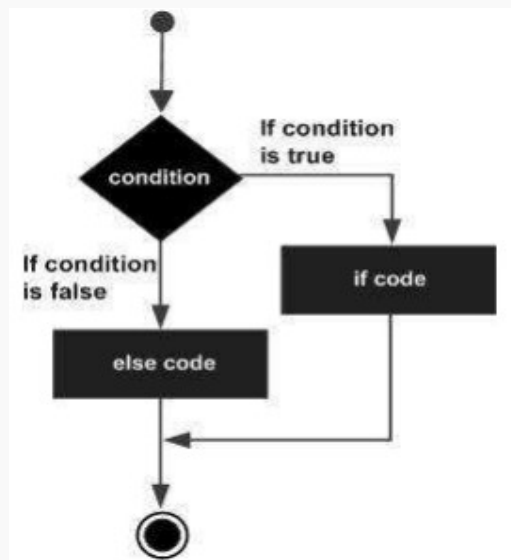
```
[1] "x contains negative numbers"
```

This test results in statement 2 (or the else statement) being executed

```
y<-c(8,3,2,5)
if(any(y<0)){
  print("y contains negative numbers")
}else{
  print("y contains all positive numbers")
}
```

Output

```
[1] "y contains all positive numbers"
```



7.4 IF-ELSE-IF STATEMENT:

The if-else if statement is a control structure in R that allows you to execute different blocks of code based on multiple conditions.

Syntax:

```
if (condition1) {  
  # code to be executed if condition1 is TRUE  
} else if (condition2) {  
  # code to be executed if condition1 is FALSE and condition2 is TRUE  
} else {  
  # code to be executed if all conditions are FALSE  
}
```

How it Works:

1. The first condition (condition1) is checked.
2. If condition1 is TRUE, the code inside the if block is executed.
3. If condition1 is FALSE, the second condition (condition2) is checked.
4. If condition2 is TRUE, the code inside the else if block is executed.
5. If all conditions are FALSE, the code inside the else block is executed.

Example:

```
x <- 5  
if (x > 10) {  
  print("x is greater than 10")  
} else if (x == 5) {  
  print("x is equal to 5")  
} else {  
  print("x is less than 5")  
}
```

output:

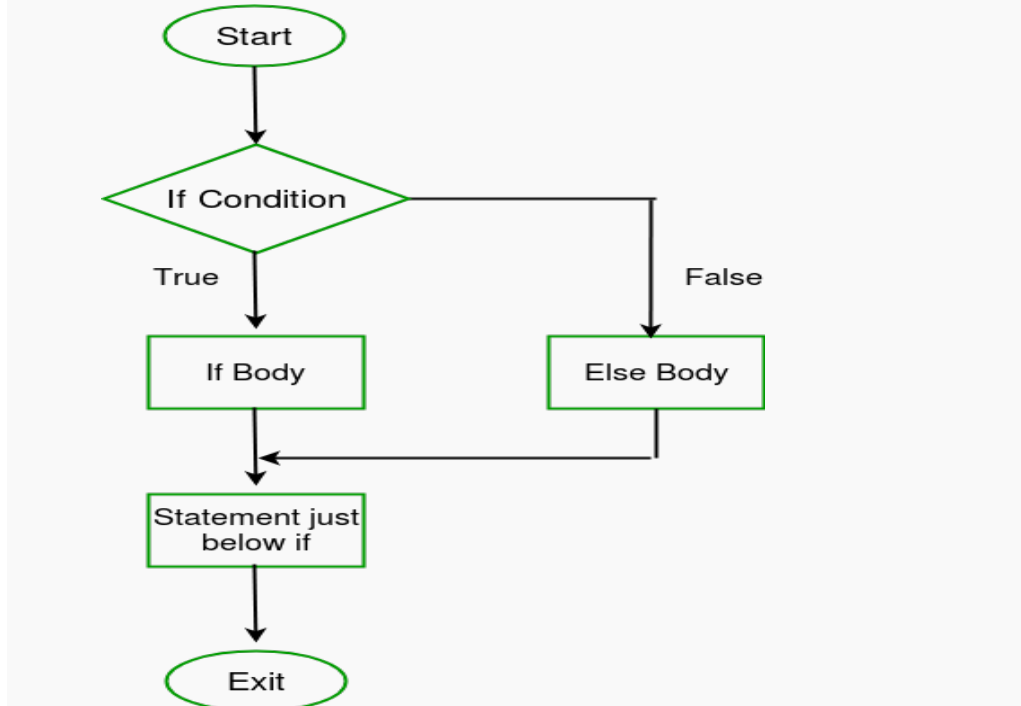
This code will print "x is equal to 5".

EXAMPLE 2

```
num <- 10  
if (num > 10) {  
  print("Number is greater than 10")  
} else if (num < 10) {  
  print("Number is less than 10")  
} else {  
  print("Number is exactly 10")  
}
```

Explanation

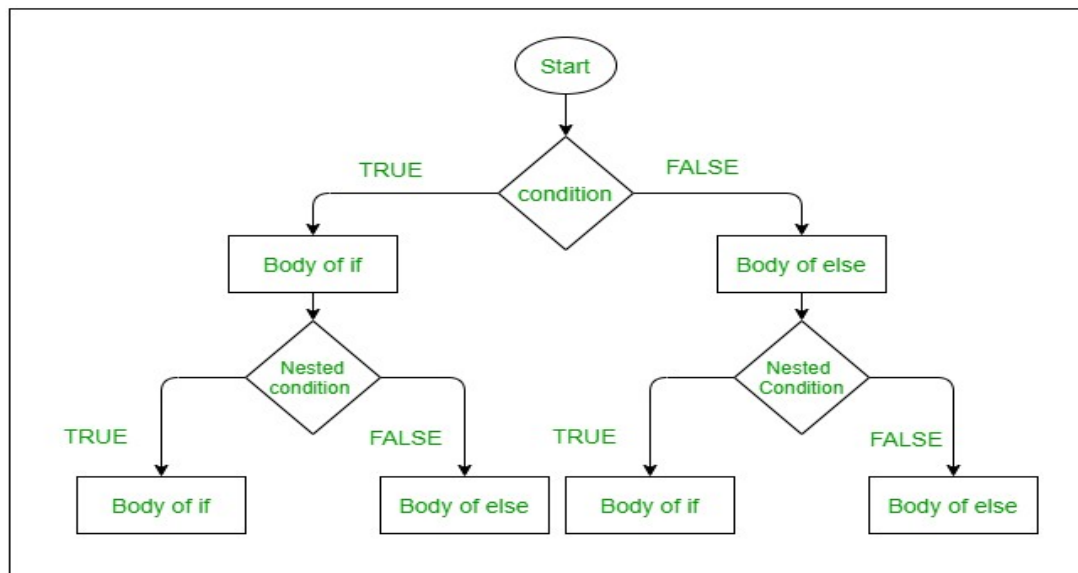
1. If num > 10, it prints "Number is greater than 10".
2. If num < 10, it prints "Number is less than 10".
3. If neither condition is true (i.e., num == 10), it prints "Number is exactly 10".

FLOW CHART**7.5 NESTED IF- ELSE STATEMENT:**

Placing one If Statement inside another If Statement is called as Nested If Else in R Programming. The R If else statement allows us to print different statements depending upon the expression result (TRUE, or FALSE). Sometimes we have to check further when the condition is TRUE. In these situations, we can use this Nested If Else concept

The basic syntax of the Nested If Else Statement in R Programming language is as follows:

```
if (Boolean_Expression 1) {  
  #Boolean_Expression 1 result is TRUE then, it will check for Boolean_Expression 2  
  if (Boolean_Expression 2) {  
    #Boolean_Expression 2 result is TRUE, then these statements will be executed  
    Boolean_Expression 2 True statements  
  } else {  
    #Boolean_Expression 2 result is FALSE then, these statements will be executed  
    Boolean_Expression 2 False statements  
  } else {  
    #If the Boolean_Expression 1 result is FALSE, these statements will be executed  
    Boolean_Expression 1 False statements  
  }  
}
```

Flow chart**7.6 SWITCH STATEMENT:**

The working functionality of the switch case in R programming is almost same as R If Statement. As we said before, Switch statement may have n number of options so, switch case compares the expression value with the values assigned in the position. If both the expression value and case value match then statements present in that position will execute. Let us see the syntax of switch case for better understanding.

```

switch(expression,

      case1 = value1,

      case2 = value2,

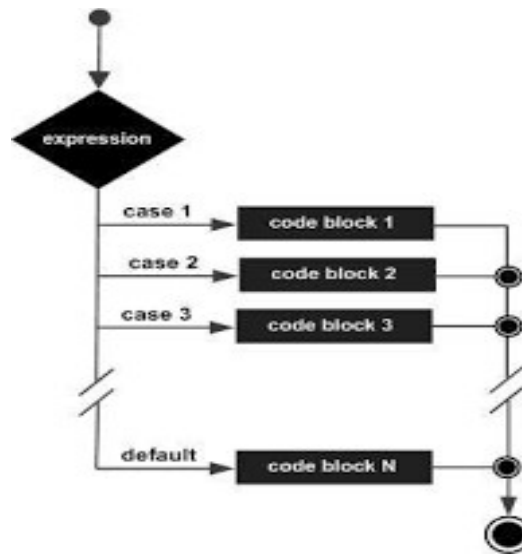
      case3 = value3,

      ...

)
  
```

Explanation:

- **expression:** Evaluates to a value (typically a string or numeric index).
- **case1, case2, ...:** Values associated with each case. If expression matches a case, the corresponding value is returned.
- If expression is numeric, it selects the position (e.g., 1 selects the first value).
- If expression is a character string, it selects the value associated with the matching name.

Flow chart**Example 1: Using Character Matching**

```
x <- "apple"
```

```
result <- switch(x,  
  apple = "You chose Apple!",  
  banana = "You chose Banana!",  
  orange = "You chose Orange!",  
  "Unknown choice"  
)
```

```
print(result)
```

Output:

```
[1] "You chose Apple!"
```

Example 2: Using Numeric Index

```
choice <- 2
```

```
result <- switch(choice,  
  "Option 1 selected",  
  "Option 2 selected",  
  "Option 3 selected"  
)
```

```
print(result)
```

Output:

```
[1] "Option 2 selected"
```

7.7 SUMMARY:

Conditional statements in R are essential for controlling the flow of execution based on certain conditions. They enable programs to make decisions and execute specific blocks of code depending on whether a condition is true or false. The if statement executes a block of code if a condition is true, while the if-else statement provides an alternative block if the condition is false. The if-else-if structure is used when multiple conditions need to be checked sequentially. Nested if statements allow for more complex decision-making by placing one condition inside another. Additionally, the switch() statement is a useful alternative for selecting from multiple options based on a value, offering a cleaner solution than lengthy if-else chains. Together, these statements improve the flexibility and efficiency of R programs, allowing them to handle various situations and data conditions dynamically.

7.8 SELF ASSESSMENT QUESTIONS:

1. What is the purpose of an if statement in R? Provide an example.
2. Explain the difference between if and if-else statements with suitable examples.
3. How does an if-else-if statement work in R? Write a program to check whether a number is positive, negative, or zero.
4. Write an R program using **nested if statements** to check if a number is even and greater than 10.
5. Describe the **switch() statement** in R. How is it different from if-else? Provide an example.
6. Write a program in R to print the grade of a student based on the following marks:
 - a) Marks ≥ 90 : "A"
 - b) Marks ≥ 75 : "B"
 - c) Marks ≥ 50 : "C"
 - d) Marks < 50 : "Fail"

7.9 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Golemund : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON -8

LOOPING STATEMENTS

OBJECTIVES:

After studying this unit, you should be able to:

- To understanding the concepts of looping statements
- Will be able to write own R-scripts
- To acquire knowledge about R programming
- To understand the purpose and objectives Decision Making Statements

STRUCTURE:

8.1 Introduction

8.2 For Loop

8.3 While Loop

8.4 Repeat Statement

8.5 Break Statement

8.6 Next Statement

8.7 Functions

8.7.1 Function Components

8.7.2 Function Arguments

8.7.3 Multi Arguments Function

8.7.4 Writing a Function in R

8.8 User Defined Functions

8.9 Conclusion

8.10 Self Assessment Questions

8.11 Further Readings

8.1 INTRODUCTION:

Looping statements are a core concept in programming that allows a programmer to execute a block of code multiple times based on certain conditions. They are essential for performing repetitive tasks without the need to write the same code over and over again.

Why Are Loops Important

Loops help in automating repetitive tasks, improving code efficiency, and making programs more concise and readable. Instead of manually repeating code, a loop can handle the repetition, reducing errors and making the code more maintainable.

8.2 FOR LOOP:

A **for loop** in R is used to iterate over a sequence (such as a vector, list, or range of numbers). It repeats a set of instructions for each element in the sequence. The for loop is extremely useful when you need to perform repetitive tasks in your R programs.

Why Use a For Loop in R

For loops are ideal when the number of iterations is known or finite. They help automate repetitive tasks, making the code more concise, efficient, and easier to maintain.

Basic Syntax of For Loop in R:

The basic structure of a for loop in R is:

```
for (variable in sequence) {
```

```
  # Code to execute
```

```
}
```

- **variable:** The loop variable, which will take each value in the sequence.
- **sequence:** A sequence of values (can be a vector, list, or range).

How Does a For Loop

1. The loop starts by initializing the **variable** to the first value in the **sequence**.
2. It then executes the block of code inside the loop.
3. After each iteration, the variable is updated to the next value in the sequence.
4. The loop continues until all elements of the sequence have been processed.

Example 1

```
for (i in 1:5) {  
  print(i)  
}
```

Explanation:

- The 1:5 creates a sequence from 1 to 5.
- The loop runs five times, and in each iteration, the value of i is printed.

Output:

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


Example 2

```
fruits <- c("apple", "banana", "cherry")
for (fruit in fruits) {
  print(fruit)
}
```

The loop iterates over the fruits vector, printing each fruit in the list.

```
[1] "apple"
```

```
[1] "banana"
```

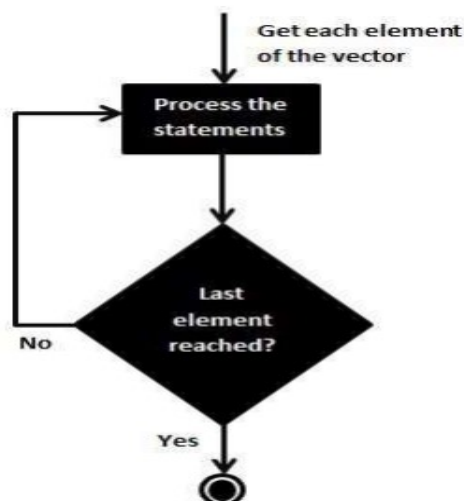
```
[1] "cherry"
```

The for loop is used to execute repetitive code statements for a particular number of times. The general syntax is provided below where *i* is the counter and as *i* assumes each sequential value defined (1 through 100 in this example) the code in the body will be performed for that *i*th value.

```
# syntax of for loop
for(iin1:100){
<do stuff here with i>
}
```

Example:

```
for(iin2010:2016){
output<-paste("The year is",i)
print(output)
}
## [1] "The year is 2010"
## [1] "The year is 2011"
## [1] "The year is 2012"
## [1] "The year is 2013"
## [1] "The year is 2014"
## [1] "The year is 2015"
## [1] "The year is 2016"
```

Flowchart

8.3 WHILE LOOP:

While loops begin by testing a condition. If it is true, then they execute the statement. Once the statement is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits. It's considered a best practice to include a counter object to keep track of total iterations

```
# syntax of while loop
counter<-1
while(test_expression){
statement
counter<-counter+1
}
```

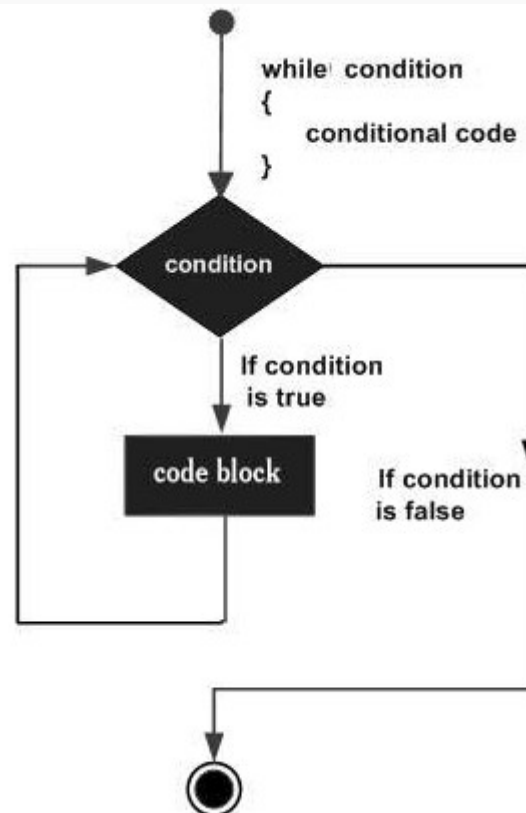
Example:

```
counter<-1
x<-5
set.seed(3)

while(x>=3&& x<=8){
  coin<-rbinom(1,1,0.5)
  if(coin==1){## random walk
    x<-x+1
  }else{
    x<-x-1
  }
  cat("On iteration",counter," , x =",x,"\n")
  counter<-counter+1
}
```

Output:

```
## On iteration 1 , x = 4
## On iteration 2 , x = 5
## On iteration 3 , x = 4
## On iteration 4 , x = 3
## On iteration 5 , x = 4
## On iteration 6 , x = 5
## On iteration 7 , x = 4
## On iteration 8 , x = 3
## On iteration 9 , x = 4
## On iteration 10 , x = 5
## On iteration 11 , x = 6
## On iteration 12 , x = 7
## On iteration 13 , x = 8
## On iteration 14 , x = 9
```

FLOW CHART**8.4 REPEAT LOOP:**

A repeat loop is used to iterate over a block of code multiple number of times. There is test expression in a repeat loop to end or exit the loop. Rather, we must put a condition statement explicitly inside the body of the loop and use the break function to exit the loop. Failing to do so will result into an infinite loop. The repeat statement is a control structure in R that allows you to execute a block of code repeatedly until a certain condition is met.

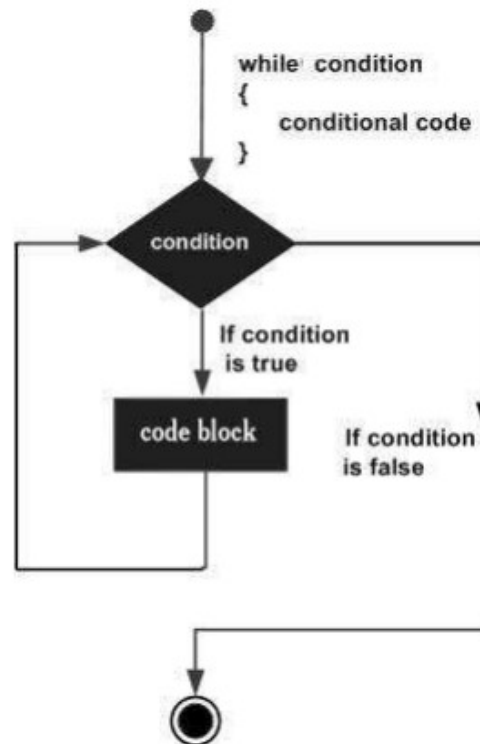
```
# syntax of repeat loop
counter<-1
repeat{
  statement
  if(test_expression){
    break
  }
  counter<-counter+1
}
```

Example:

```
counter<-1
x<-NULL
repeat{
  x<-c(x,round(runif(1,min=1,max=25)))
  if(all(1:25%in%x)){
    break
  }
}
```

```
counter<-counter+1
}
counter
## [1] 75
```

Flow chart



8.5 BREAK ARGUMENTS:

The break argument is used to exit a loop immediately, regardless of what iteration the loop may be on. break arguments are typically embedded in an if statement in which a condition is assessed, if TRUE break out of the loop, if FALSE continue on with the loop. In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

In this example, the for loop will iterate for each element in x; however, when it gets to the element that equals 3 it will break out and end the for loop process.

```
x<-1:5
```

```
for(iinx){
  if(i==3){
    break
  }
  print(i)
}
```

Output:

```
## [1] 1
```

```
## [1] 2
```

Example:

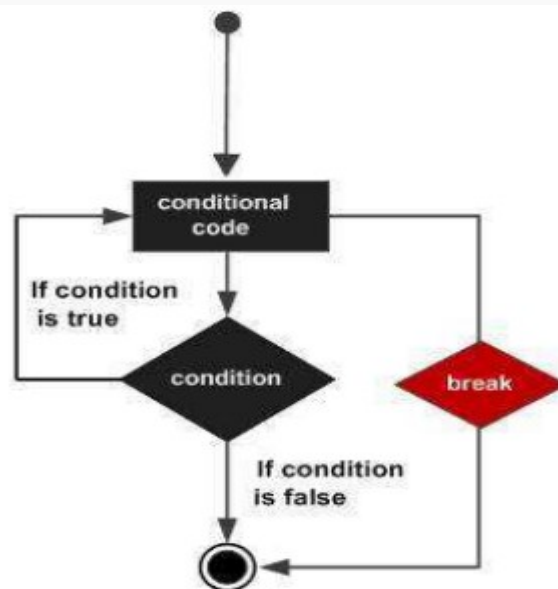
```
x<-1:5
```

```
for(iinx){  
  if(i==3){  
    next  
  }  
  print(i)  
}
```

Output:

```
[1] 1  
[1] 2  
[1] 4  
[1] 5
```

FLOW CHART



8.6 NEXT STATEMENT:

The next statement in R is used within loops to skip the current iteration and move directly to the next one. It is useful when certain conditions need to be met before continuing with the next iteration.

Usage of next in Loops

The next statement is typically used in for and while loops. When the condition for next is satisfied, the rest of the statements in that iteration are skipped, and the loop proceeds to the next iteration.

Example 1

```
for (i in 1:10) {  
  if (i %% 2 == 0) { # Check if the number is even  
    next # Skip even numbers  
  }  
  print(i) # Print only odd numbers  
}
```

Explanation

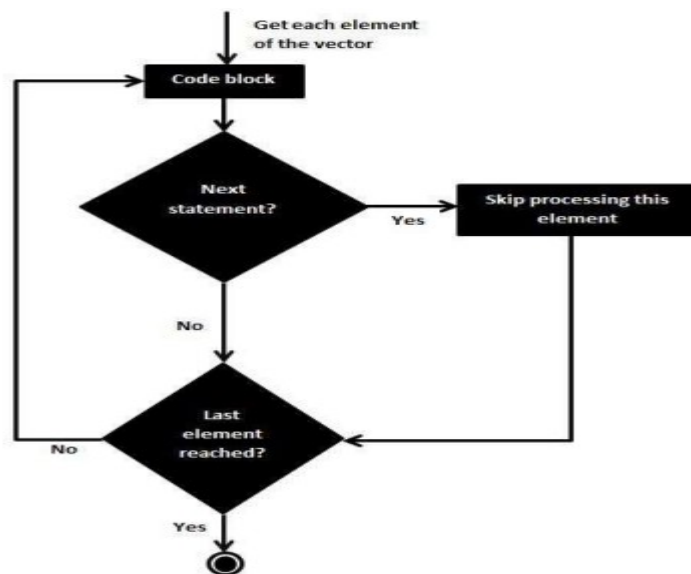
- The loop iterates through numbers **1 to 10**.
- If **i** is **even** ($i \% 2 == 0$), the next statement is executed, skipping the `print(i)` command.
- As a result, only **odd numbers** are printed.

Example 2

```
x <- 0
while (x < 10) {
  x <- x + 1
  if (x %% 2 == 0) {
    next # Skip even numbers
  }
  print(x)
}
```

Use Cases of next Statement

1. **Skipping specific values:** Useful when certain values need to be ignored during iteration.
2. **Avoiding unnecessary computations:** Helps improve efficiency by skipping unwanted operations.
3. **Handling special conditions:** Useful for avoiding errors or unwanted processing.

Flow chart**8.7 FUNCTIONS:**

A **function**, in a programming environment, is a set of instructions. A programmer builds a function to avoid **repeating** the same task, or reduce **complexity**.

A function should be

- written to carry out a specified a tasks
- may or may not include arguments
- contain a body
- May or may not return one or more values.

A general approach to a function is to use the argument part as **inputs**, feed the **body** part and finally return an **output**. The Syntax of a function is the following:

```
Function (arglist) {
#Function body
}
```

8.7.1 Function components

The different parts of a function are –

- **Function Name** – This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** – An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** – The function body contains a collection of statements that defines what the function does.
- **Return Value** – The return value of a function is the last expression in the function body to be evaluated.

8.7.2 Function Arguments:

It's useful to distinguish between the formal arguments and the actual arguments of a function. The formal arguments are a property of the function, whereas the actual or calling arguments can vary each time you call the function. This section discusses how calling arguments are mapped to formal arguments, how you can call a function given a list of arguments, how default arguments work, and the impact of lazy evaluation.

Functions have named arguments which potentially have default values.

- 1) The formal arguments are the arguments included in the function definition
- 2) The formals function returns a list of all the formal arguments of a function
- 3) Not every function call in R makes use of all the formal arguments Function arguments
- 4) can be missing or might have default values

R functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent

```
mydata <- rnorm(100)
sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
```

```
function (formula, data, subset, weights, na.action, method = "qr", model = TRUE, x
= FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset,
...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
```

```
lm(y ~ x, mydata, 1:100, model = FALSE)
```

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Function arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is

- Check for exact match for a named argument
- Check for a partial match
- Check for a positional match

8.7.3 Multi arguments function

We can write a function with more than one argument. Consider the function called "times". It is a straightforward function multiplying two variables.

```
times <- function(x,y) {
  x*y
}
times(2,4)
```

Output:

```
## [1] 8.
```

8.7.4 Writing a function in R

In some occasion, we need to write our own function because we have to accomplish a particular task and no ready made function exists. A user-defined function involves a **name**, **arguments** and a **body**.

```
function.name <- function(arguments)
{
  computations on the arguments
  some other code
}
```

Note: A good practice is to name a user-defined function different from a built-in function. It avoids confusion.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

8.8 USER DEFINED FUNCTIONS:

One of the great strengths of R is the user's ability to add functions. In fact, many of the functions in R are actually functions of functions. The structure of a function is given below.

```
myfunction<-function(arg1,arg2,... ){  
  statements  
  return(object)  
}
```

Objects in the function are local to the function. The object returned can be any data type. Here is an example.

```
mysummary <- function(x,npar=TRUE,print=TRUE) {  
  if (!npar) {  
    center <- mean(x); spread <- sd(x)  
  } else {  
    center <- median(x); spread <- mad(x)  
  }  
  if (print & !npar) {  
    cat("Mean=", center, "\n", "SD=", spread, "\n")  
  } else if (print & npar) {  
    cat("Median=", center, "\n", "MAD=", spread, "\n")  
  }  
  result <- list(center=center,spread=spread)  
  return(result)  
}
```

8.9 SUMMARY:

We explored various fundamental control flow structures and functions in R programming, which are essential for creating efficient and readable code. We covered loops such as the **for loop** and **while loop**, each of which serves a specific purpose in controlling the flow of execution based on conditions. The **repeat statement** was also discussed, offering another way to repeat actions, with the possibility of breaking the loop using the **break statement** or skipping iterations using the **next statement**.

We also delved into **functions**, which are critical for encapsulating repetitive tasks and ensuring code reusability. By understanding the components of a function, including **function arguments**, we can write versatile functions that take multiple arguments. Learning how to define user-defined functions in R also allows us to create tailored solutions for specific tasks, providing a deeper understanding of how to structure programs effectively.

8.10 SELF ASSESSMENT QUESTIONS:

1. Write a for loop in R that prints all even numbers between 1 and 20.
2. Explain the concept of **function arguments** and give an example of a function with multiple arguments in R.
3. What are the key components of a function in R? Provide an example function that accepts two parameters and returns their sum.
4. Write a while loop in R that continuously asks the user for input until they enter the word "stop".
5. Create a user-defined function in R that takes a list of numbers and returns the mean and standard deviation of those numbers.
6. Using a repeat loop, create a program that keeps asking the user to guess a number between 1 and 100, and ends when the correct number is guessed.

8.11 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language, Wiley India Pvt Ltd.
- 2) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 3) J.P. Lander(2014):R for Everyone, Pearson Publications
- 4) Garrett Grolemond : Hands-On Programming with R
- 5) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON -9

R- FUNCTIONS

OBJECTIVES:

After studying this unit, you should be able to:

- Will be able to handle the data analysis using the R-statistical tools
- The student will learn how to perform graphical presentation of the data
- Understand the concepts of R-functions
- Able to write Their own R-codes with and without using Built-in functions

STRUCTURE:

9.1 Built-in functions

9.2 General functions

9.2.1 diff () function

9.2.2 length () function

9.3 Statistical functions

9.4 Scoping:

9.4.1 Scoping Rules:

9.4.2 Environment Scoping

9.5 One argument function

9.6. Calling functions

9.7 R Codes for small standard statistical problems

9.8 Apply functions

9.8.1 apply()

9.8.2 sapply()

9.8.3 lapply()

9.8.4 tapply()

9.9 Conclusion

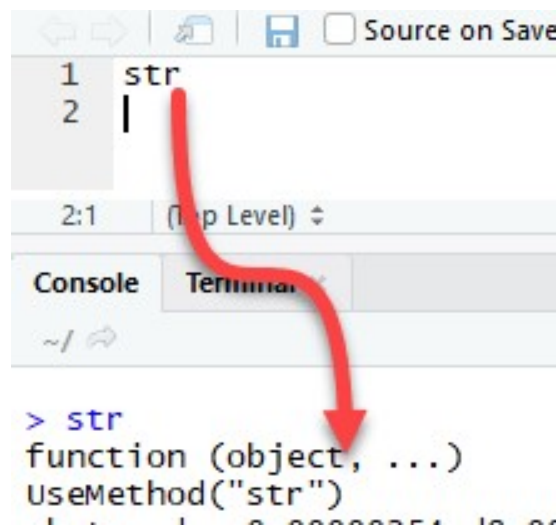
9.10 Self Assessment Questions

9.11 Further Readings

9.1 BUILT-IN FUNCTIONS:

There is a lot of built-in function in R. R matches your input parameters with its function arguments; either by value or by position, then executes the function body. Function arguments can have default values: if you do not specify these arguments, R will take the default value.

Note: It is possible to see the source code of a function by running the name of the function itself in the console.



We will see three groups of function in action

- General function
- Maths function
- Statistical function

9.2 GENERAL FUNCTIONS:

We are already familiar with general functions like `cbind()`, `rbind()`, `range()`, `sort()`, `order()` functions. Each of these functions has a specific task, takes arguments to return an output. Following are important functions one must know-

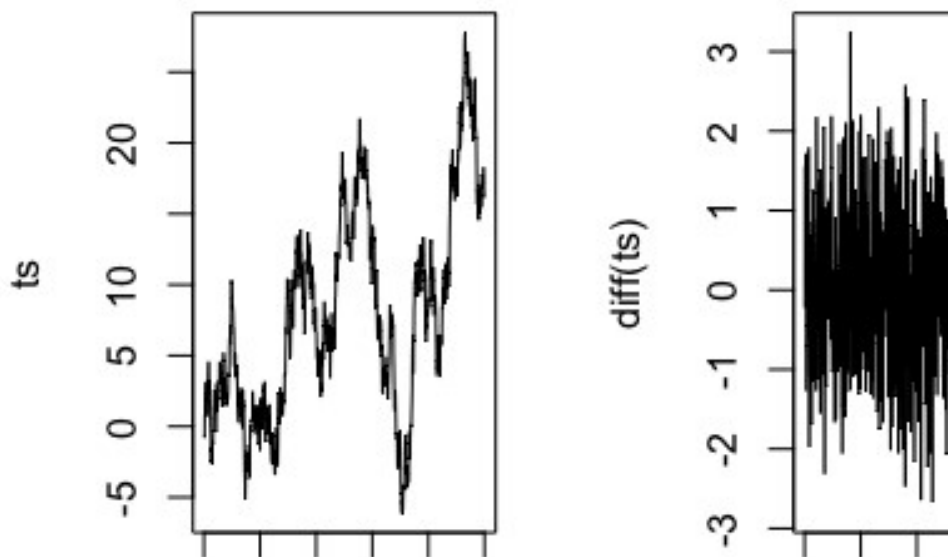
9.2.1 diff() function:

If you work on **time series**, you need to stationary the series by taking their **lag values**. A **stationary process** allows constant mean, variance and autocorrelation over time. This mainly improves the prediction of a time series. It can be easily done with the function `diff()`. We can build a random time-series data with a trend and then use the function `diff()` to stationary the series. The `diff()` function accepts one argument, a vector, and return suitable lagged and iterated difference.

Note: We often need to create random data, but for learning and comparison we want the numbers to be identical across machines. To ensure we all generate the same data, we use the `set.seed()` function with arbitrary values of 123. The `set.seed()` function is generated through the process of pseudorandom number generator that make every modern computers to have the same sequence of numbers. If we don't use `set.seed()` function, we will all have different sequence of numbers.

```
set.seed(123)
## Create the data
x = rnorm(1000)
ts <- cumsum(x)
```

```
## Stationary the serie  
diff_ts <- diff(ts)  
par(mfrow=c(1,2))  
## Plot the series  
plot(ts, type='l')  
plot(diff(ts), type='l')
```



9.2.2 length() function

In many cases, we want to know the **length** of a vector for computation or to be used in a for loop. The `length()` function counts the number of rows in vector `x`. The following codes import the cars dataset and return the number of rows.

Note: `length()` returns the number of elements in a vector. If the function is passed into a matrix or a data frame, the number of columns is returned.

```
dt <- cars  
## number columns  
length(dt)
```

Output:

```
## [1] 1  
## number rows  
length(dt[,1])
```

Output:

```
[1] 50
```

Math functions

R has an array of mathematical functions.

Operator	Description
<code>abs(x)</code>	Takes the absolute value of x
<code>log(x,base=y)</code>	Takes the logarithm of x with base y; if base is not specified, returns the natural logarithm
<code>exp(x)</code>	Returns the exponential of x
<code>sqrt(x)</code>	Returns the square root of x
<code>factorial(x)</code>	Returns the factorial of x (x!)

```
# sequence of number from 44 to 55 both including incremented by 1
x_vector <- seq(45,55, by = 1)
#logarithm
log(x_vector)
```

```
[1] 3.806662 3.828641 3.850148 3.871201 3.891820 3.912023 3.931826 3.951244 3.970292
3.988984 4.007333
```

#exponential

```
exp(x_vector)
```

Output:

```
3.493427e+19 9.496119e+19 2.581313e+20 7.016736e+20 1.907347e+21 5.184706e+21
1.409349e+22 3.831008e+22 1.041376e+23 2.830753e+23 7.694785e+23
```

#squared root

```
sqrt(x_vector)
```

Output:

```
6.708204 6.782330 6.855655 6.928203 7.000000 7.071068 7.141428 7.211103 7.280110
7.348469 7.416198
```

#factorial

factorial(x_vector)

Output:

```
1.196222e+56 5.502622e+57 2.586232e+59 1.241392e+61 6.082819e+62
3.041409e+64 1.551119e+66 8.065818e+67 4.274883e+69 2.308437e+71
1.269640e+73
```

9.3 STATISTICAL FUNCTIONS:

R standard installation contains wide range of statistical functions. In this tutorial, we will briefly look at the most important function..

Basic statistic functions

Operator	Description
mean(x)	Mean of x
median(x)	Median of x
var(x)	Variance of x
sd(x)	Standard deviation of x
scale(x)	Standard scores (z-scores) of x
quantile(x)	The quartiles of x
summary(x)	Summary of x: mean, min, max etc..

```
speed <- dt$speed
speed
# Mean speed of cars dataset
mean(speed)
```

Output:

```
## [1] 15.4
# Median speed of cars dataset
median(speed)
```

Output:

```
## [1] 15
# Variance speed of cars dataset
var(speed)
```

Output:

```
## [1] 27.95918
# Standard deviation speed of cars dataset
sd(speed)
```

Output:

```
## [1] 5.287644
# Standardize vector speed of cars dataset
head(scale(speed), 5)
```

Output:

```
##      [,1]
## [1,] -2.155969
## [2,] -2.155969
## [3,] -1.588609
## [4,] -1.588609
## [5,] -1.399489
# Quantile speed of cars dataset
quantile(speed)
```

Output:

```
## 0% 25% 50% 75% 100%
##  4 12 15 19 25
# Summary speed of cars dataset
summary(speed)
```

Output:

```
##  Min. 1st Qu.  Median    Mean 3rd Qu.   Max.
##  4.0   12.0   15.0   15.4   19.0   25.0
```

Up to this point, we have learned a lot of R built-in functions.

Note: Be careful with the class of the argument, i.e. numeric, Boolean or string. For instance, if we need to pass a string value, we need to enclose the string in quotation mark: "ABC" .

9.4 SCOPING:

Scoping is the set of rules that govern how R looks up the value of a symbol. In the example below, scoping is the set of rules that R applies to go from the symbol x to its value 10:

```
x <- 10

x

## [1] 10
```


Understanding scoping allows you to:

- build tools by composing functions, as described in functional programming.
- overrule the usual evaluation rules and do non-standard evaluation, as described in non-standard evaluation.

R has two types of scoping: **lexical scoping**, implemented automatically at the language level, and **dynamic scoping**, used in select functions to save typing during interactive analysis. We discuss lexical scoping here because it is intimately tied to function creation. Dynamic scoping is described in more detail in scoping issues.

Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called. With lexical scoping, you don't need to know how the function is called to figure out where the value of a variable will be looked up. You just need to look at the function's definition.

The “lexical” in lexical scoping doesn't correspond to the usual English definition (“of or relating to words or the vocabulary of a language as distinguished from its grammar and construction”) but comes from the computer science term “lexing”, which is part of the process that converts code represented as text to meaningful pieces that the programming language understands.

There are four basic principles behind R's implementation of lexical scoping:

- name masking
- functions vs. variables
- a fresh start
- dynamic lookup

9.4.1 Scoping Rules:

The scoping rules for R are the main feature that make it different from the original S language.

- 1) The scoping rules determine how a value is associated with a free variable in a function
- 2) R uses lexical scoping or static scoping. A common alternative is dynamic scoping.
- 3) Related to the scoping rules is how R uses the search list to bind a value to a symbol
- 4) Lexical scoping turns out to be particularly useful for simplifying statistical computations

Consider the following function.

```
f <- function(x, y) {  
  x^2 + y / z  
}
```

This function has 2 formal arguments x and y. In the body of the function there is another symbol z. In this case z is called a free variable.

The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned inside the function body).

9.4.2 Environment Scoping

In R, the **environment** is a **collection** of objects like functions, variables, data frame, etc.

R opens an environment each time Rstudio is prompted.

The top-level environment available is the **global environment**, called `R_GlobalEnv`. And we have the **local environment**.

We can list the content of the current environment.

```
ls(environment())
```

Output

```
## [1] "diff_ts"      "dt"           "speed"        "square_function"  
## [5] "ts"           "x"            "x_vector"
```

You can see all the variables and function created in the `R_Global Env`.

The above list will vary for you based on the historic code you execute in R Studio.

Note that `n`, the argument of the `square_function` function is **not in this global environment**.

A **new** environment is created for each function. In the above example, the function `square_function()` creates a new environment inside the global environment.

To clarify the difference between **global** and **local environment**, let's study the following example

This function takes a value `x` as an argument and adds it to `y` defined outside and inside the function

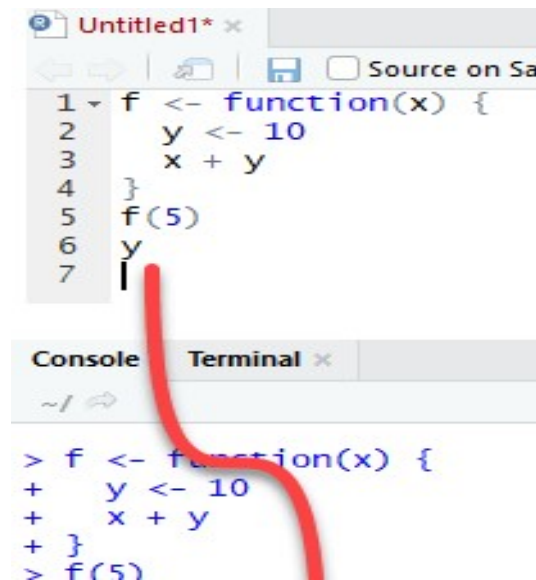


```
1 y <- 10  
2 f <- function(  
3   x + y}  
4 f(5)  
5 y  
6  
Console Terminal x  
> f(5)  
[1] 15  
> ## [1] 15
```

The function `f` returns the output 15. This is because `y` is defined in the global environment. Any variable defined in the global environment can be used locally. The variable `y` has the value of 10 during all function calls and is accessible at any time.

Let's see what happens if the variable `y` is defined inside the function.

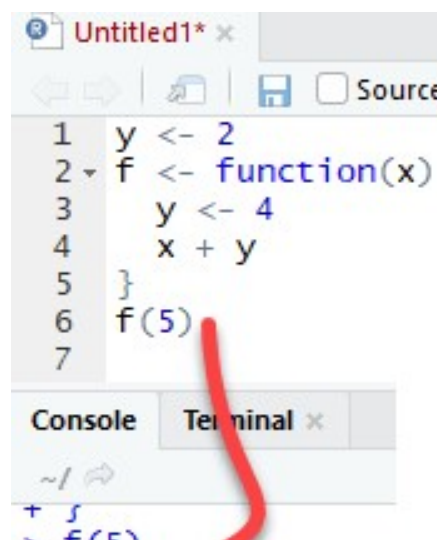
We need to drop `y` prior to run this code using `rm y`



```
1 f <- function(x) {  
2   y <- 10  
3   x + y  
4 }  
5 f(5)  
6 y  
7  
  
> f <- function(x) {  
+   y <- 10  
+   x + y  
+ }  
> f(5)
```

The output is also 15 when we call `f(5)` but returns an error when we try to print the value `y`. The variable `y` is not in the global environment.

Finally, R uses the most recent variable definition to pass inside the body of a function. Let's consider the following example:



```
1 y <- 2  
2 f <- function(x)  
3   y <- 4  
4   x + y  
5 }  
6 f(5)  
7  
  
+ f(5)
```

R ignores the `y` values defined outside the function because we explicitly created a `y` variable inside the body of the function.

9.5 ONE ARGUMENT FUNCTION:

In the next snippet, we define a simple square function. The function accepts a value and returns the square of the value.

```
square_function<- function(n)
{
# compute the square of integer `n`
n^2
}
# calling the function and passing value 4
square_function(4)
```

Code Explanation:

- The function is named `square_function`; it can be called whatever we want.
- It receives an argument "n". We **didn't specify the type of variable so that the user can pass an integer, a vector or a matrix**
- The function takes the input "n" and returns the square of the input.

When you are done using the function, we can remove it with the `rm()` function.

after you create the function

```
rm(square_function)
square_function
```

On the console, we can see an error message :Error: object 'square_function' not found telling the function does not exist.

Every operation is a function call

“To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call."

9.6. CALLING FUNCTIONS:

When calling a function you can specify arguments by position, by complete name, or by partial name. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

```
f<-function(abcdef, bcde1, bcde2) {
list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
str(f(1, 2, 3))
## List of 3
```

```
## $ a : num 1
## $ b1: num 2
## $ b2: num 3
str(f(2, 3, abcdef=1))
## List of 3
## $ a : num 1
## $ b1: num 2
## $ b2: num 3
# Can abbreviate long argument names:
str(f(2, 3, a=1))
## List of 3
## $ a : num 1
## $ b1: num 2
## $ b2: num 3
# But this doesn't work because abbreviation is ambiguous
str(f(1, 3, b=1))
## Error in f(1, 3, b = 1): argument 3 matches multiple formal arguments
```

Generally, you only want to use positional matching for the first one or two arguments; they will be the most commonly used, and most readers will know what they are. Avoid using positional matching for less commonly used arguments, and only use readable abbreviations with partial matching. (If you are writing code for a package that you want to publish on CRAN you can not use partial matching, and must use complete names.) Named arguments should always come after unnamed arguments. If a function uses ... (discussed in more detail below), you can only specify arguments listed after ... with their full name.

These are good calls:

```
mean(1:10)
mean(1:10, trim =0.05)
```

This is probably overkill:

```
mean(x =1:10)
```

And these are just confusing:

```
mean(1:10, n = T)
mean(1:10, , FALSE)
mean(1:10, 0.05)
```

```
mean(, TRUE, x =c(1:10, NA))
```

Calling a function given a list of arguments

Suppose you had a list of function arguments:

```
args <-list(1:10, na.rm =TRUE)
```

How could you then send that list to mean()? You need do.call():

```
do.call(mean, args)
## [1] 5.5
# Equivalent to
mean(1:10, na.rm =TRUE)
## [1] 5.5
```

Default and missing arguments

Function arguments in R can have default values.

```
f <-function(a =1, b =2) {
  c(a, b)
}
f()
## [1] 1 2
```

Since arguments in R are evaluated lazily (more on that below), the default value can be defined in terms of other arguments:

```
g <-function(a =1, b = a *2) {
  c(a, b)
}
g()
## [1] 1 2
g(10)
## [1] 10 20
```

Default arguments can even be defined in terms of variables created within the function. This is used frequently in base R functions, but I think it is bad practice, because you can't understand what the default values will be without reading the complete source code.

```
h <-function(a =1, b = d) {  
  d <-(a +1) ^2  
  c(a, b)  
}  
  
h()  
## [1] 1 4  
  
h(10)  
## [1] 10 121
```

You can determine if an argument was supplied or not with the `missing()` function.

```
i <-function(a, b) {  
  c(missing(a), missing(b))  
}  
  
i()  
## [1] TRUE TRUE  
  
i(a =1)  
## [1] FALSE TRUE  
  
i(b =2)  
## [1] TRUE FALSE  
  
i(1, 2)  
## [1] FALSE FALSE
```

Sometimes you want to add a non-trivial default value, which might take several lines of code to compute. Instead of inserting that code in the function definition, you could use `missing()` to conditionally compute it if needed. However, this makes it hard to know which arguments are required and which are optional without carefully reading the documentation. Instead, I usually set the default value to `NULL` and use `is.null()` to check if the argument was supplied.

9.7 R CODES FOR SMALL STANDARD STATISTICAL PROBLEMS :

#R-code for finding arithmetic mean, standard deviation (SD), coefficient of variation (CV)

```
cat ("\n enter sample values:");  
  
S=scan();  
  
n=length(S);  
  
mean=0;SD=0;
```

```
for(x in S)
{
mean=mean+x;SD=SD+x*x;
mean=mean/n;
SD=SD/n-mean*mean;
SD=sqrt(SD);
CV=100*SD/mean;
cat("mean of the given sample=",mean);
cat("\n SD of the given sample=",SD);
cat("\n CV of the given sample=",CV);
}
```

Output:

enter sample values:1: 2

2: 4

3: 6

4: 8

5: 11

6: 19

7: 22

8: 28

9:

Read 8 items

mean of the given sample= 12.5

SD of the given sample= 8.803408

CV of the given sample= 70.42727

#R-code for obtaining Range and Median

```
cat("\n Enter sample:");
```

```
x=scan();
```

```
n=length(x);
```



```
for(i in 1:(n-1))for(j in (i+1):n){if(x[i]>x[j]){t=x[i];x[i]=x[j];x[j]=t;}}
print("sorted sample:");print(x);
if(n%%2==0)median=(x[n/2]+x[n/2+1])/2 else median=x[(n+1)/2]
cat("\n median of the given sample=",median);
cat("\n Range of the given sample=",x[n]-x[1]);
cat("\n\n");
```

Output:

Enter sample: 1: 2

2: 4

3: 6

4: 8

5: 11

6: 19

7: 22

8: 28

Read 8 items

[1] "sorted sample:"

[1] 2 4 6 8 11 19 22 28

median of the given sample= 9.5

Range of the given sample= 26

#R program for Correlation coefficient of given bivariate sample

```
cat("\n Enter sample x:");
```

```
x=scan();
```

```
cat("\n Enter sample y:");
```

```
y=scan();
```

```
#x=rnorm(30);
```

```
#y=rnorm(30);
```

```
n=length(x);
```

```
#demonstration of for loop
```

```
mx=my=0;
sxx=syy=sxy=0;#i=1;
for(i in 1:n){mx=mx+x[i];my=my+y[i];
sxx=sxx+x[i]*x[i];syy=syy+y[i]*y[i];
sxy=sxy+x[i]*y[i];}
mx=mx/n;my=my/n;
vx=sxx/n-mx*mx;
vy=syy/n-my*my;cov=sxy/n-mx*my;
r=cov/sqrt(vx*vy);
cat("\n Covariance of x&Y=",cov);
cat("\n correlation of the given data=",r);
cat("\n\n");
```

Output:

Enter sample x: 1: 2

2: 4

3: 6

4: 8

Read 4 items

Enter sample y: 1: 3

2: 5

3: 7

4: 9

Read 4 items

Covariance of x &y = 5

correlation of the given data= 1

#R- program for one sample t-test

```
cat("\n Enter sample:");x=scan();
```

```
cat("\n population mean mu0=");mu0=scan();
```

```
n=length(x);
```

```
#demonstration of for loop
mean=s=0;
for(x in X){mean=mean+x;s=s+x*x;}
mean=mean/n;s=s-n*mean**2;
s=sqrt(s/(n-1));
t=(mean-mu0)/s*sqrt(n);
cat("\n mean=",mean);
cat("\n s=",s);cat("\n mu0=",mu0);
cat("\n t-critical value (at 5% LOS)=",qt(0.975,n-1));cat("DF=",n-1);
cat("\n\n t value based on one sample=",t);

if(abs(t)<qt(0.975,n-1))cat("\n\n The given has been drawn from the Normal population with
mean=",mu0) else cat("The given sample has not been drawn from the Normal population
with mean=",mu0)

cat("\n\n");
```

Output:

Enter sample: 1: 36

2: 37

3: 37

4: 40

5: 41

6: 42

7: 43

8: 44

9: 46

10: 47

11: 48

12: 48

13: 51

14: 52

15: 53

16: 59

17: 55

18: 55

19: 56

20: 60

Read 20 items

population mean $\mu_0=1$: 47.5

Read 1 item

mean= 3.9

s= 4.266146

$\mu_0= 47.5$

t- critical value (at 5% LOS)= 2.093024DF= 19

t value based on one sample= -45.70522

The given sample has not been drawn from the Normal population with mean= 47.5

9.8 APPLY FUNCTIONS:

The apply family of functions in R(e.g., apply, lapply, sapply, tapply etc.) allows for streamlined operations over various data structures. To perform group-wise manipulation using these functions, you often work with data frames or lists, grouping your data by a particular column or criteria.

Common scenario for Group manipulations using apply family of functions given below:

9.8.1 apply()

The apply function is extremely useful for manipulating grouped data within matrices or arrays. It works by applying a function to rows or columns of a matrix, or over dimensions of a multi-dimensional array. For instance, if you have a matrix of data where rows represent individuals and columns represent different variables, you can use apply to compute statistics (like mean or sum) for each row or column. This eliminates the need for cumbersome loops, streamlining your code. One limitation, however, is that apply is primarily suited for homogeneous data structures. When working with grouped data, you can reshape the data into an array and process groups using the desired dimension (MARGIN = 1 for rows, MARGIN = 2 for columns). For advanced group manipulation, combining apply with other tools like split can provide

even more flexibility.

Syntax:

apply(X, MARGIN, FUN, ...)

- X: The matrix or array.
- MARGIN: 1 for rows, 2 for columns.
- FUN: The function to apply.
- ...: Additional arguments to the function.

Example:

```
data=matrix(1:12,4,byrow=T)

data
      [,1] [,2] [,3]
[1,]  1    2    3
[2,]  4    5    6
[3,]  7    8    9
[4,] 10   11   12

apply(data,1,sum)
[1]  6 15 24 33

apply(data,2,sum)
[1] 22 26 30
```

In this example by using the **apply()** function to perform the various statistical functions for different dimensions. Here we use `sum()` for the matrix shows the sum of each row elements for `MARGIN=1` and each column elements for `MARGIN=2`.

9.8.2 sapply()

`sapply` is a simplified version of `lapply` that is ideal for grouped data when you want the output in a more compact form, such as a vector or matrix. This function applies a specified function over a vector or list of grouped data, trying to simplify the results automatically. For instance, in data analysis, if you have a list of numeric vectors (each representing a group), you can use `sapply` to calculate the mean, variance, or other summary statistics for each group.

The function adapts well to grouped data because it handles vectors or lists easily, converting them into a unified output format. However, for more complex group structures, `sapply` might fail to simplify the output as expected.

Syntax:

```
lapply(X, FUN, ...)
```

- X: A list or vector.
- FUN: The function to apply.

Example:

```
data()
```

```
D=women
```

```
D
```

```
  height weight
```

```
1  58  115
```

```
2  59  117
```

```
3  60  120
```

```
4  61  123
```

```
5  62  126
```

```
6  63  129
```

```
7  64  132
```

```
8  65  135
```

```
9  66  139
```

```
10 67  142
```

```
11 68  146
```

```
12 69  150
```

```
13 70  154
```

```
14 71  159
```

```
15 72  164
```

```
sapply(D,mean) height weight 65.0000 136.7333
```

Generally, in R there are so many data sets stored. We can use those data sets to perform the statistical computations by using **sapply()** function.

In this example, the output will print like this i.e., average of the heights and weights for the given data. In this example the output will be in horizontal values.

9.8.3 lapply()

lapply is especially powerful for handling grouped data stored in lists. It applies a function to each element of a list (or vector) and always returns a list as output. This is particularly useful for working with grouped datasets where each group is represented as a separate list element. For example, you can use **lapply** to clean, transform, or analyze data within each group independently. Its flexibility allows it to handle groups of varying sizes and structures without assuming any simplification, making it ideal for more complex or nested data manipulation tasks.

Syntax:

```
lapply(X, function,..., simplify = TRUE)
```

- simplify = TRUE: Attempts to simplify output.

Example:

From the example of sapply function

```
lapply(D,sd)
```

```
$height
```

```
[1] 4.472136
```

```
$weight
```

```
[1] 15.49869
```

Here also we use those data sets to perform the computations by using **lapply()**function.

9.8.4 tapply()

tapply is uniquely designed for grouped data manipulation and is one of the most intuitive functions for this purpose. It splits a vector into groups based on a factor (or multiple factors) and applies a function to each group. This makes it invaluable for summarizing data by categories, such as computing group means, medians, or counts. Unlike apply, tapply handles vectors rather than matrices and allows for more direct grouping operations. It's commonly used in exploratory data analysis, where summarizing data by groups is a frequent task. One drawback is that it may produce outputs as arrays, which might require additional formatting for further analysis.

Syntax:

```
tapply(X, INDEX, function, ...)
```

- X: The vector to split.
- INDEX: A factor or list of factors.
- function: The function to apply.

Example:1

```
x=matrix(sample(1:100,45),5,9)
```

```
x
```

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

```
[1,] 60 43 93 63 2 71 6 49 1
```

```
[2,] 62 30 34 46 72 79 40 35 5
```

```
[3,] 80 61 57 78 75 59 82 67 96
```

```
[4,] 9 74 38 70 21 37 28 47 99
```

```
[5,] 11 55 84 18 88 91 32 20 90
```

Example:2

```
x <- c(1, 2, 3, 4, 5, 6)
```

```
group <- c("A", "A", "B", "B", "C", "C")
```

```
tapply(x, group, mean) # apply mean function to each subset
```

Output:

```
  A  B  C  
1.5 3.5 5.5
```

9.9 SUMMARY:

Mastering R's built-in functions, scoping rules, and the apply family of functions is essential for efficient data manipulation, statistical analysis, and writing reliable code. These concepts form the foundation for solving complex data analysis tasks and developing robust statistical models in R.

9.10 SELF ASSESSMENT QUESTIONS:

1. What are built-in functions in R? Give two examples.
2. Explain the role of mathematical and string manipulation functions in R with examples.
3. What is the purpose of the diff() function? Write a sample R code demonstrating its use.
4. How does the length() function work in R? Provide an example.
5. List any four statistical functions in R and explain their use.
6. What is an environment in R? How does it affect variable searching in a function?
7. Explain the concept of a one-argument function with an example.
8. How are functions called in R? Write an example to call the sqrt() function.
9. Write R code to find the median of a dataset.
10. Differentiate between apply(), sapply(), lapply(), and tapply() functions in R.

9.11 SUGGESTED READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language,
- 2) Wiley India Pvt Ltd.
- 3) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 4) J.P. Lander(2014):R for Everyone, Pearson Publications
- 5) Garrett Golemund : Hands-On Programming with R
- 6) Michael J. Crawley: The R Book

Dr. SYED JILANI

LESSON-10

PROBABILITY DISTRIBUTIONS IN R

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the importance of Probability distribution in R
- Students should have a solid understanding about the Generating samples
- .To know the concepts of Built-in R- functions
- To acquire knowledge about PDF and CDF for the distributions

STRUCTURE:

10.1 Binomial distribution

10.2 Poisson distribution

10.3 Normal distribution

10.4 Exponential distribution

10.5 Weibull distribution

10.6 Logistic distribution

10.7 Conclusion

10.8 Self Assessment Questions

10.9 Further Readings

10.1 BINOMIAL DISTRIBUTION:

Introduction: The Binomial Distribution is a discrete probability distribution representing the number of successes in a fixed number of independent Bernoulli trials, each having the same probability of success.

A binomially distributed random variable X follows:

$$X \sim \text{Bin}(n, p)$$

where:

- n = number of trials
- p = probability of success in each trial

Probability mass function (PMF): The Probability Mass Function (PMF) is given by:

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}, \quad k = 0, 1, 2, \dots, n$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the binomial coefficient.

Cumulative distribution function (CDF): The Cumulative Distribution Function (CDF)

gives the probability that X takes a value less than or equal to a given value:

$$F(X) = P(X \leq k) = \sum_{i=0}^k P(X = i)$$

Procedure in R: R provides built-in functions to work with the binomial distribution:

Function	Description
<code>dbinom(x, size, prob)</code>	Computes the probability mass function (PMF).
<code>pbinom(q, size, prob)</code>	Computes the cumulative probability (CDF).
<code>qbinom(p, size, prob)</code>	Computes the quantile function (inverse CDF).
<code>rbinom(n, size, prob)</code>	Generates random samples from a binomial distribution.

Example in R:

Set parameters for the binomial distribution

`n <- 10` # Number of trials

`p <- 0.5` # Probability of success

1. Compute PMF (Probability Mass Function)

`x_values <- 0:n` # Possible outcomes

`pmf_values <- dbinom(x_values, size=n, prob=p)`

`print(pmf_values)`

```
> print(pmf_values)
[1] 0.0009765625 0.0097656250 0.0439453125 0.1171875000 0.2050781250 0.2460937500
[7] 0.2050781250 0.1171875000 0.0439453125 0.0097656250 0.0009765625
```

2. Compute CDF (Cumulative Probability)

`cdf_values <- pbinom(x_values, size=n, prob=p)`

`print(cdf_values)`

```
> print(cdf_values)
[1] 0.0009765625 0.0107421875 0.0546875000 0.1718750000 0.3769531250 0.6230468750
[7] 0.8281250000 0.9453125000 0.9892578125 0.9990234375 1.0000000000
```

3. Compute Quantile Function

`quantile_value <- qbinom(0.5, size=n, prob=p)` # 50th percentile

`print(quantile_value)`

```
> print(quantile_value)
[1] 5
```

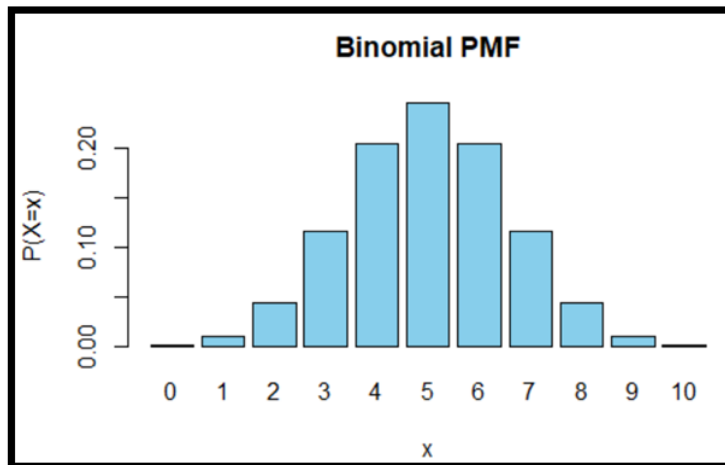
4. Generate Random Samples

```
random_samples <- rbinom(1000, size=n, prob=p) # Generate 1000 samples  
print(head(random_samples))
```

```
> print(head(random_samples))  
[1] 3 5 4 4 6 2
```

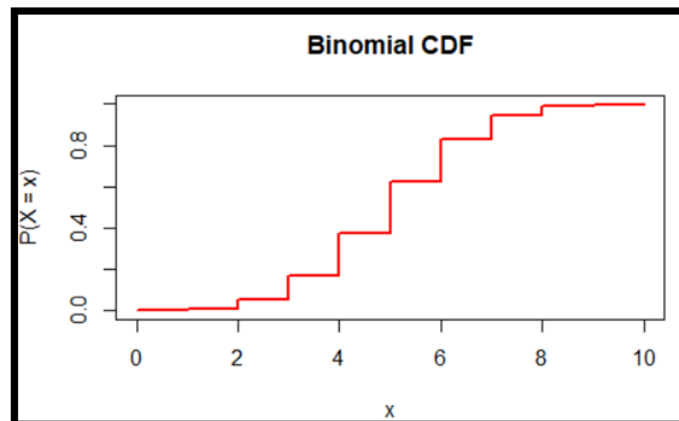
5. Plot PMF (Probability Density Curve)

```
barplot(pmf_values, names.arg=x_values, col="skyblue", main="Binomial PMF", xlab="x",  
ylab="P(X=x)")
```



6. Plot CDF (Cumulative Density Curve)

```
plot(x_values, cdf_values, type="s", col="red", lwd=2, main="Binomial CDF", xlab="x",  
ylab="P(X ≤ x)")
```



Explanation:

1. **PMF Calculation:** The `dbinom()` function calculates probabilities for each possible outcome in the binomial distribution.
2. **CDF Calculation:** The `pbinom()` function calculates cumulative probabilities.
3. **Quantile Calculation:** The `qbinom()` function determines the smallest x such that $P(X \leq x)$ is at least a given probability.

4. **Random Sampling:** The `rbinom()` function generates random numbers from the binomial distribution.
5. **PMF Plot:** The `barplot()` function is used to visualize the probability distribution.
6. **CDF Plot:** The `plot()` function is used to draw the cumulative distribution function as a step function.

Conclusion:

- The binomial distribution is useful for modeling the number of successes in a fixed number of trials.
- R provides built-in functions to compute probabilities, cumulative probabilities, quantiles, and generate random samples.
- Visualizing PMF and CDF helps in understanding the distribution.

10.2 POISSON DISTRIBUTION:

Introduction: The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant mean rate and independently of the time since the last event.

Probability mass function (PMF): The probability of observing k events in an interval is given by:

$$P(X = k) = \frac{e^{-\lambda} \lambda^k}{k!}, \quad k = 0, 1, 2, \dots$$

where:

- λ (lambda) is the expected number of occurrences in the given interval (mean rate),
- e is Euler's number (~ 2.71828).

Estimating lambda value: The value of λ (mean rate of occurrences) can be estimated from real-world data using:

$$\lambda = \frac{\text{Total number of occurrences}}{\text{Total number of intervals}}$$

Example: Suppose a call center receives 50 calls in 10 hours. The estimated λ is:

$$\lambda = \frac{50}{10} = 5$$

Cumulative distribution function (CDF): The CDF gives the probability of obtaining at most k occurrences:

$$F(k) = P(X \leq k) = \sum_{i=0}^k \frac{e^{-\lambda} \lambda^i}{i!}$$

Quantile function: The quantile function finds the value of k for which the probability $P(X \leq k)$ is equal to a given probability p .

Generating random samples: Random numbers following a Poisson distribution can be generated to simulate real-world Poisson processes.

Procedure in R: In R, the Poisson distribution is handled using built-in functions:

Function	Description
dpois(x, lambda)	Computes the PMF (probability mass function) at x .
ppois(x, lambda)	Computes the CDF (cumulative probability) at x .
qpois(p, lambda)	Computes the quantile function (inverse CDF).
rpois(n, lambda)	Generates n random samples from a Poisson distribution.

Example implementation in R:

```
# Set lambda (mean rate)
lambda <- 4
```

```
# Define the range of x values
x_values <- 0:15
```

```
# Compute PMF (Probability Mass Function)
pmf_values <- dpois(x_values, lambda)
```

```
pmf_values
```

```
> pmf_values
[1] 1.831564e-02 7.326256e-02 1.465251e-01 1.953668e-01 1.953668e-01 1.562935e-01
[7] 1.041956e-01 5.954036e-02 2.977018e-02 1.323119e-02 5.292477e-03 1.924537e-03
[13] 6.415123e-04 1.973884e-04 5.639669e-05 1.503912e-05
```

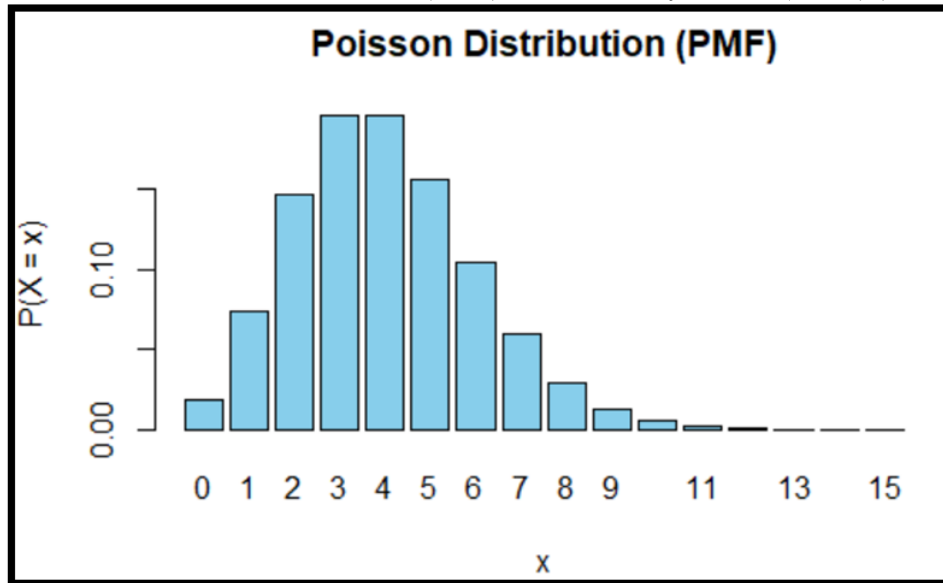
```
# Compute CDF (Cumulative Distribution Function)
cdf_values <- ppois(x_values, lambda)
```

```
> cdf_values
[1] 0.01831564 0.09157819 0.23810331 0.43347012 0.62883694 0.78513039 0.88932602
[8] 0.94886638 0.97863657 0.99186776 0.99716023 0.99908477 0.99972628 0.99992367
[15] 0.99998007 0.99999511
```

```
# Generate 1000 random samples from Poisson distribution
set.seed(123) # For reproducibility
random_samples <- rpois(1000, lambda)
```

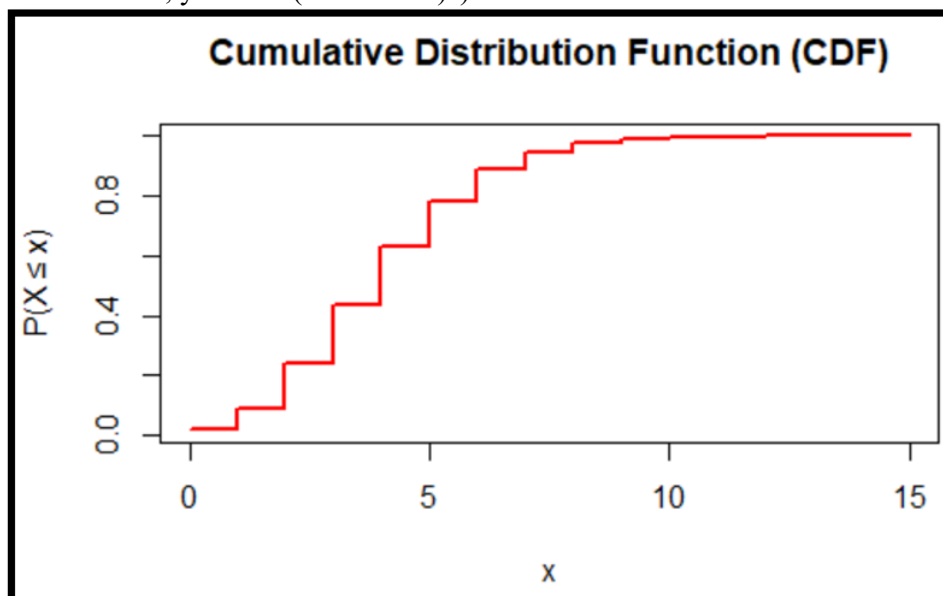
```
# Plot the PMF
barplot(pmf_values, names.arg = x_values, col = "skyblue",
```

```
main = "Poisson Distribution (PMF)", xlab = "x", ylab = "P(X = x)")
```



```
# Plot the CDF
```

```
plot(x_values, cdf_values, type = "s", col = "red", lwd = 2,  
     main = "Cumulative Distribution Function (CDF)",  
     xlab = "x", ylab = "P(X ≤ x)")
```

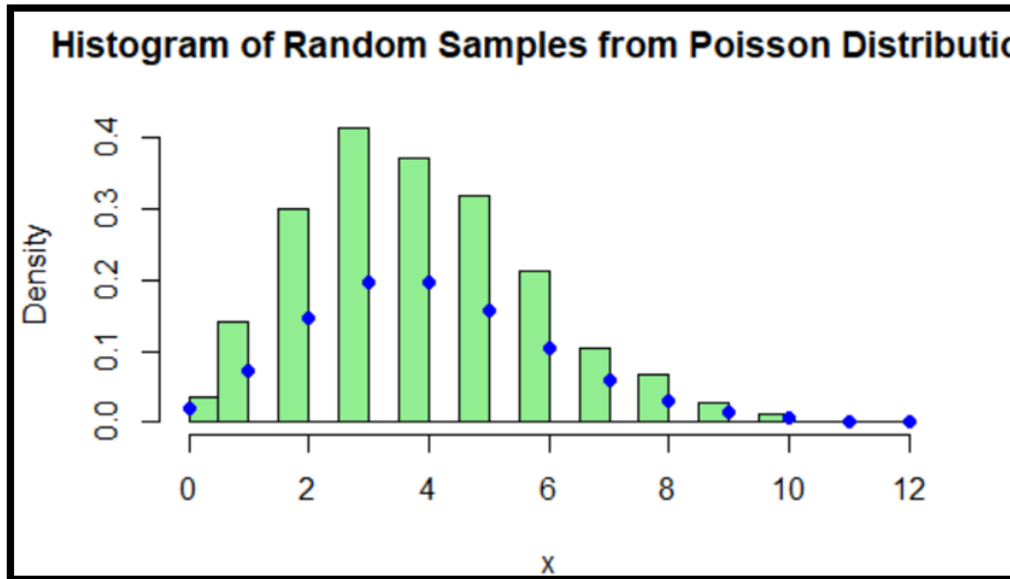


```
# Histogram of generated samples
```

```
hist(random_samples, breaks = 20, col = "lightgreen", probability = TRUE,  
     main = "Histogram of Random Samples from Poisson Distribution",  
     xlab = "x", ylab = "Density")
```

```
# Overlay PMF on the histogram
```

```
lines(x_values, dpois(x_values, lambda), type = "p", col = "blue", pch = 16)
```



Explanation of the code:

1. **Define lambda:** Set the Poisson parameter $\lambda = 4$.
2. **Compute PMF:** Use `dpois()` to get probability values for given x .
3. **Compute CDF:** Use `ppois()` to get cumulative probabilities.
4. **Generate random samples:** Use `rpois()` to generate 1000 values.
5. **Plot PMF:** A bar chart is created using `barplot()`.
6. **Plot CDF:** A step plot is drawn using `plot()` with type = "s".
7. **Plot Histogram:** A histogram of generated samples is overlaid with theoretical PMF.

Conclusion:

- The Poisson distribution is useful for modeling the count of occurrences over a fixed interval.
- R provides built-in functions for computing probabilities, cumulative probabilities, quantiles, and generating random samples.
- Visualizing PMF and CDF helps understand the distribution.

10.3 NORMAL DISTRIBUTION:

Introduction: The **Normal Distribution** is one of the most widely used probability distributions in statistics. It is a continuous probability distribution, defined by its mean (μ) and standard deviation (σ). The Normal Distribution is symmetric about its mean, meaning the left and right sides of the graph are mirror images of each other. It is described by the probability density function (PDF) as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where:

- μ is the mean of the distribution,
- σ is the standard deviation,
- e is the base of the natural logarithm.

Built-in R functions for Normal Distribution:

R provides several built-in functions to work with the Normal Distribution:

1. **dnorm(x, mean, sd)**: Computes the **probability density function** (PDF) of the Normal Distribution for the given x value.
2. **pnorm(q, mean, sd)**: Computes the **cumulative distribution function** (CDF) for the given quantile q.
3. **qnorm(p, mean, sd)**: Computes the **quantile** for the given cumulative probability p.
4. **rnorm(n, mean, sd)**: Generates n random samples from a Normal Distribution with specified mean and sd.

Computing PDF, CDF, and Quantiles:

To compute the **PDF**, **CDF**, and **Quantiles**, we use the following functions:

- **PDF**: The dnorm() function calculates the value of the probability density for a given point x. Example:

```
# Compute PDF for x = 1, mean = 0, sd = 1
dnorm(1, mean = 0, sd = 1)
```

```
> # Compute PDF for x = 1, mean = 0, sd = 1
> dnorm(1, mean = 0, sd = 1)
[1] 0.2419707
```

- **CDF**: The pnorm() function calculates the cumulative probability up to a given quantile q. Example:

```
# Compute CDF for q = 1, mean = 0, sd = 1
pnorm(1, mean = 0, sd = 1)
```

```
> # Compute CDF for q = 1, mean = 0, sd = 1
> pnorm(1, mean = 0, sd = 1)
[1] 0.8413447
```

- **Quantiles**: The qnorm() function computes the quantile value for a given cumulative probability p. Example:

```
# Compute quantile for p = 0.95, mean = 0, sd = 1
qnorm(0.95, mean = 0, sd = 1)
```

```
> # Compute quantile for p = 0.95, mean = 0, sd = 1
> qnorm(0.95, mean = 0, sd = 1)
[1] 1.644854
```


Generating samples:

To generate random samples from a Normal Distribution, we use the `rnorm()` function. It takes the number of samples, the mean, and the standard deviation as input:

```
# Generate 100 random samples from a Normal Distribution with mean = 0, sd = 1
samples <- rnorm(100, mean = 0, sd = 1)
```

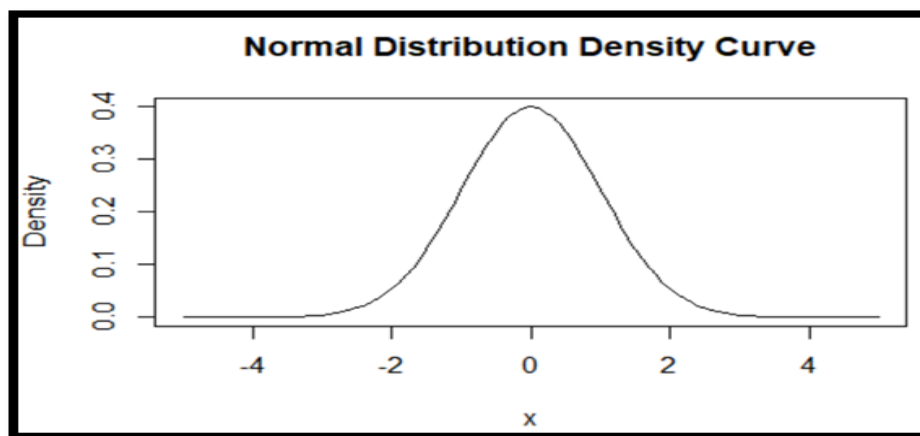
Plotting Density and Cumulative Density Curves:

R also provides functions to visually represent the **Density Curve** and the **Cumulative Density Curve** for the Normal Distribution:

- **Density Plot:** The `plot()` function along with `dnorm()` can be used to plot the density of the Normal Distribution.

Example:

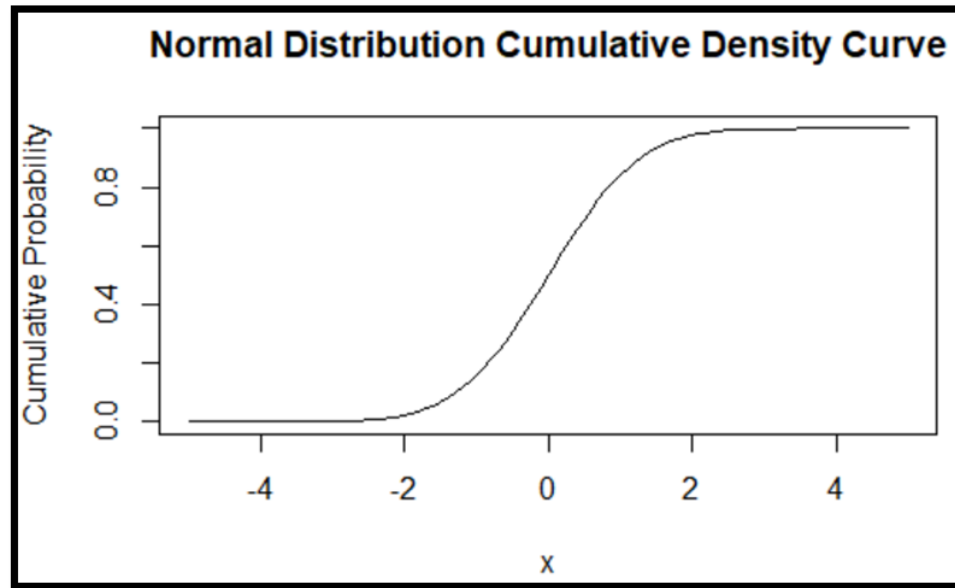
```
# Plot the Normal Density Curve for mean = 0, sd = 1
x <- seq(-5, 5, by = 0.1)
y <- dnorm(x, mean = 0, sd = 1)
plot(x, y, type = "l", main = "Normal Distribution Density Curve", ylab = "Density",
      xlab = "x")
```



- **Cumulative Density Plot:** To plot the cumulative density, use `pnorm()`:

Example:

```
# Plot the Cumulative Density Curve for mean = 0, sd = 1
y_cdf <- pnorm(x, mean = 0, sd = 1)
plot(x, y_cdf, type = "l", main = "Normal Distribution Cumulative Density Curve",
      ylab = "Cumulative Probability", xlab = "x")
```



10.4 EXPONENTIAL DISTRIBUTION:

Introduction:

The **Exponential distribution** is widely used to model the time between independent events that occur at a constant average rate. It is a continuous probability distribution with a single parameter, λ (**lambda**), which represents the rate of occurrence.

Mathematically, the **Probability Density Function (PDF)** of an exponential distribution is given by:

$$f(x) = \lambda e^{-\lambda x}, \quad x \geq 0, \lambda > 0$$

The **Cumulative Distribution Function (CDF)** is given by:

$$F(x) = 1 - e^{-\lambda x}, \quad x \geq 0$$

The mean of an exponential distribution is $1/\lambda$, and the variance is $1/\lambda^2$.

Exponential Distribution using R-software:

R provides built-in functions to compute various properties of the exponential distribution:

- **dexp(x, rate = λ)** → Computes the Probability Density Function (PDF).
- **pexp(x, rate = λ)** → Computes the Cumulative Distribution Function (CDF).
- **qexp(p, rate = λ)** → Computes the Quantile Function (inverse CDF).
- **rexp(n, rate = λ)** → Generates random samples from an exponential distribution.

Estimating the Rate Parameter (λ) for an Exponential Distribution:

The **rate parameter (λ)** in an **Exponential Distribution** represents the average rate at which events occur. It is the reciprocal of the mean ($\lambda = 1/\text{mean}$).

Methods to estimate λ :**Using the sample mean**

- The Maximum Likelihood Estimator (MLE) for λ is:

$$\hat{\lambda} = \frac{1}{\bar{x}}$$

where \bar{x} is the sample mean.

Using the method of moments

- Since the expected value of an exponential distribution is $E(X) = 1/\lambda$, we estimate λ as:

$$\hat{\lambda} = \frac{1}{\text{mean}(\text{sample data})}$$

Estimating λ in R with an example:***Step 1: Generate Sample Data***

We generate a random sample from an **Exponential Distribution** with a known λ and then estimate it.

```
# Set seed for reproducibility
set.seed(123)
```

```
# Generate 20 random samples from an exponential distribution with  $\lambda = 2$ 
true_lambda <- 2
sample_data <- rexp(20, rate = true_lambda)
```

```
# Display the sample data
sample_data
```

Step 2: Estimate λ Using the Sample Mean

```
# Compute the sample mean
sample_mean <- mean(sample_data)
```

```
# Estimate lambda as 1/sample_mean
estimated_lambda <- 1 / sample_mean
```

```
# Display the estimated lambda
estimated_lambda
```

Step 3: Compare Estimated λ with True λ

```
cat("True lambda:", true_lambda, "\nEstimated lambda:", estimated_lambda)
```

Interpretation

- The estimated λ should be **close** to the true value (**2 in this example**), but it may vary due to randomness.
- As the **sample size increases**, the estimate becomes more accurate.

Conclusion:

We estimated λ (rate parameter) using the **MLE method** by computing **1/mean(sample data)**. This method is simple and effective for determining the **rate of occurrence** of events in an **Exponential Distribution**.

Computing PDF, CDF, Quantile and Random Sampling in R:**1. Computing Probability Density Function (PDF)**

The **PDF** gives the likelihood of observing a particular value in an exponential distribution.

R Code:

```
# Define rate parameter ( $\lambda$ )
lambda <- 2
```

```
# Compute PDF for a range of x values
x_values <- seq(0, 2, by = 0.1)
pdf_values <- dexp(x_values, rate = lambda)

pdf_values
```

```
> pdf_values
[1] 2.00000000 1.63746151 1.34064009 1.09762327 0.89865793 0.73575888 0.60238842
[8] 0.49319393 0.40379304 0.33059778 0.27067057 0.22160632 0.18143591 0.14854716
[15] 0.12162013 0.09957414 0.08152441 0.06674654 0.05464744 0.04474154 0.03663128
```

```
# Display the computed PDF values
data.frame(x_values, pdf_values)
```

```
> data.frame(x_values, pdf_values)
  x_values pdf_values
1      0.0 2.00000000
2      0.1 1.63746151
3      0.2 1.34064009
4      0.3 1.09762327
5      0.4 0.89865793
6      0.5 0.73575888
7      0.6 0.60238842
8      0.7 0.49319393
9      0.8 0.40379304
10     0.9 0.33059778
11     1.0 0.27067057
12     1.1 0.22160632
13     1.2 0.18143591
14     1.3 0.14854716
15     1.4 0.12162013
16     1.5 0.09957414
17     1.6 0.08152441
18     1.7 0.06674654
19     1.8 0.05464744
20     1.9 0.04474154
21     2.0 0.03663128
```

2. Computing Cumulative Distribution Function (CDF)

The **CDF** gives the probability that a random variable is less than or equal to a given value.

R Code:

```
# Compute CDF values for the same x_values
```

```
cdf_values <- pexp(x_values, rate = lambda)
```

```
# Display computed CDF values
```

```
data.frame(x_values, cdf_values)
```

```
> data.frame(x_values, cdf_values)
  x_values cdf_values
1      0.0 0.0000000
2      0.1 0.1812692
3      0.2 0.3296800
4      0.3 0.4511884
5      0.4 0.5506710
6      0.5 0.6321206
7      0.6 0.6988058
8      0.7 0.7534030
9      0.8 0.7981035
10     0.9 0.8347011
11     1.0 0.8646647
12     1.1 0.8891968
13     1.2 0.9092820
14     1.3 0.9257264
15     1.4 0.9391899
16     1.5 0.9502129
17     1.6 0.9592378
18     1.7 0.9666267
19     1.8 0.9726763
20     1.9 0.9776292
21     2.0 0.9816844
```

3. Computing Quantiles

The **Quantile function** gives the value corresponding to a given cumulative probability.

R Code:

```
# Compute quantiles for given probabilities
```

```
probabilities <- c(0.1, 0.25, 0.5, 0.75, 0.9)
```

```
quantiles <- qexp(probabilities, rate = lambda)
```

```
# Display computed quantiles
```

```
data.frame(probabilities, quantiles)
```

```
> data.frame(probabilities, quantiles)
 probabilities quantiles
1          0.10 0.05268026
2          0.25 0.14384104
3          0.50 0.34657359
4          0.75 0.69314718
5          0.90 1.15129255
```

4. Generating Random Samples

We can generate random samples from an **Exponential distribution** using `rexp()`.

R Code:

```
# Generate 10 random samples from exponential distribution
set.seed(123) # Set seed for reproducibility
random_samples <- rexp(10, rate = lambda)

# Display the generated random samples
random_samples
```

```
> random_samples
[1] 0.42172863 0.28830514 0.66452743 0.01578868 0.02810549 0.15825061 0.15711365
[8] 0.07263340 1.36311823 0.01457672
```

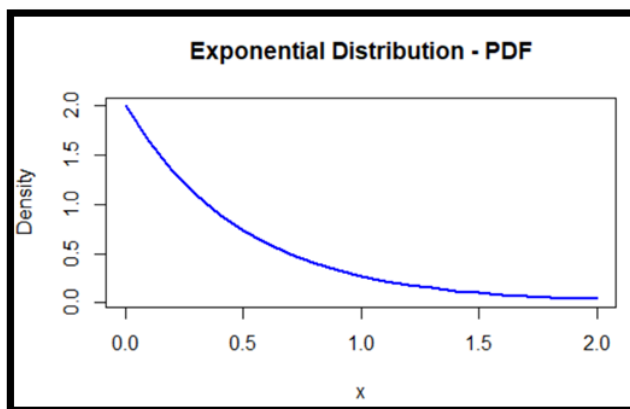
Plotting Density and Cumulative Density Curves:

1. Plotting the PDF (Density Curve)

To visualize the **Probability Density Function**, we can use the `plot()` function.

R Code:

```
# Plot the Exponential PDF
plot(x_values, pdf_values, type = "l", col = "blue", lwd = 2,
     main = "Exponential Distribution - PDF",
     xlab = "x", ylab = "Density")
```

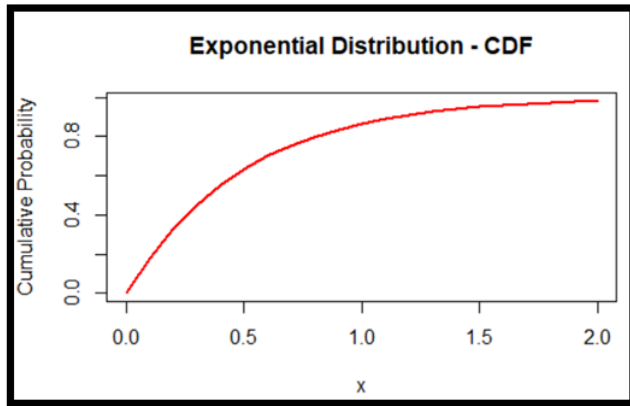


2. Plotting The CDF (Cumulative Density Curve)

To visualize the **Cumulative Distribution Function**, we plot the CDF values.

R Code:

```
# Plot the Exponential CDF
plot(x_values, cdf_values, type = "l", col = "red", lwd = 2,
     main = "Exponential Distribution - CDF",
     xlab = "x", ylab = "Cumulative Probability")
```



Conclusion:

This lesson covered the **Exponential Distribution**, its properties, and how to compute PDF, CDF, quantiles, and generate random samples in **R**. We also demonstrated how to plot **density and cumulative density curves** for better visualization.

By understanding and applying these concepts, we can model and analyze **time-to-event** data effectively using R.

10.5 WEIBULL DISTRIBUTION:

Introduction: The **Weibull distribution** is a continuous probability distribution used in reliability analysis, life data analysis, and survival analysis. It is characterized by two parameters:

1. **Shape parameter (k or shape)** – Determines the shape of the distribution.
2. **Scale parameter (λ or scale)** – Stretches or compresses the distribution.

The probability density function (PDF) of a Weibull distribution is given by:

$$f(x; k, \lambda) = \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k}, \quad x > 0$$

The cumulative distribution function (CDF) is:

$$F(x; k, \lambda) = 1 - e^{-(x/\lambda)^k}, \quad x > 0$$

The quantile function is the inverse of the CDF.

Setting Weibull Shape and Scale Parameters in R:

Manually defining Weibull parameters: If the shape (k) and scale (λ) parameters are already known, they can be assigned directly in R:

```
shape_param <- 2 # Shape parameter (k)
scale_param <- 5 # Scale parameter ( $\lambda$ )
```

Estimating Weibull parameters from data: If data is available, the Weibull parameters can be estimated using the `fitdistrplus` package.

Step 1: Install and load necessary libraries

```
install.packages("fitdistrplus") # Install package if not already installed
library(fitdistrplus)
```

Step 2: Generate sample data from a Weibull Distribution

```
set.seed(123) # For reproducibility
data <- rweibull(100, shape = 2, scale = 5) # Simulating data
```

Step 3: Fit Weibull Distribution to the data

```
fit <- fitdist(data, "weibull")
summary(fit)
```

```
> summary(fit)
Fitting of the distribution ' weibull ' by maximum likelihood
Parameters :
      estimate Std. Error
shape 2.040070  0.1563357
scale 4.997746  0.2581219
Loglikelihood: -218.497   AIC:  440.9941   BIC:  446.2044
Correlation matrix:
      shape    scale
shape 1.0000000 0.3152897
scale 0.3152897 1.0000000
```

Output: The summary(fit) command will display the estimated shape and scale parameters based on the provided data.

Using SurvivalPackage: Another method to estimate Weibull parameters is by using the survival package:

```
install.packages("survival")
library(survival)

weibull_model <- survreg(Surv(data) ~ 1, dist = "weibull")
summary(weibull_model)
```

```
Call:
survreg(formula = Surv(data) ~ 1, dist = "weibull")
              Value Std. Error      z      p
(Intercept)  1.6091     0.0516 31.1 <2e-16
Log(scale)   -0.7130     0.0766 -9.3 <2e-16

Scale= 0.49

Weibull distribution
Loglik(model)= -218.5   Loglik(intercept only)= -218.5
Number of Newton-Raphson Iterations: 6
n= 100
```

The summary(weibull_model) function provides the estimated parameters based on survival analysis techniques.

Conclusion:

This lesson demonstrated how to manually define and estimate Weibull distribution parameters in R using statistical tools. Understanding these concepts is essential for applications in reliability engineering and survival analysis.

Procedure in R:

R provides built-in functions to work with the Weibull distribution:

Function	Description
dweibull(x, shape, scale)	Computes the probability density function (PDF).
pweibull(x, shape, scale)	Computes the cumulative distribution function (CDF).
qweibull(p, shape, scale)	Computes the quantile function (inverse of CDF).
rweibull(n, shape, scale)	Generates random samples from a Weibull distribution.

Brief example to get Location (μ) and Scale (s) parameters

To estimate the location and scale parameters from a given dataset, we can use the `fitdistr` function from the MASS package in R.

```
# Load required package
library(MASS)

# Generate a sample dataset
set.seed(123)
data <- rlogis(100, location = 5, scale = 2)

# Estimate parameters
fit <- fitdistr(data, "logistic")
print(fit)
```

```
> print(fit)
  location      scale
 4.9820634  1.9290582
(0.3342806) (0.1607817)
```

This will output estimates for the location (μ) and scale (s) parameters based on the given dataset.

Example in R:

```
# Set Weibull distribution parameters
shape_param <- 2 # Shape parameter (k)
scale_param <- 3 # Scale parameter (lambda)

# Define x values for plotting
x_vals <- seq(0, 10, length.out = 100)

# Compute PDF and CDF values
pdf_vals <- dweibull(x_vals, shape = shape_param, scale = scale_param)
```

```
cdf_vals <- pweibull(x_vals, shape = shape_param, scale = scale_param)
```

```
# Generate random samples
```

```
set.seed(123) # For reproducibility
```

```
samples <- rweibull(1000, shape = shape_param, scale = scale_param)
```

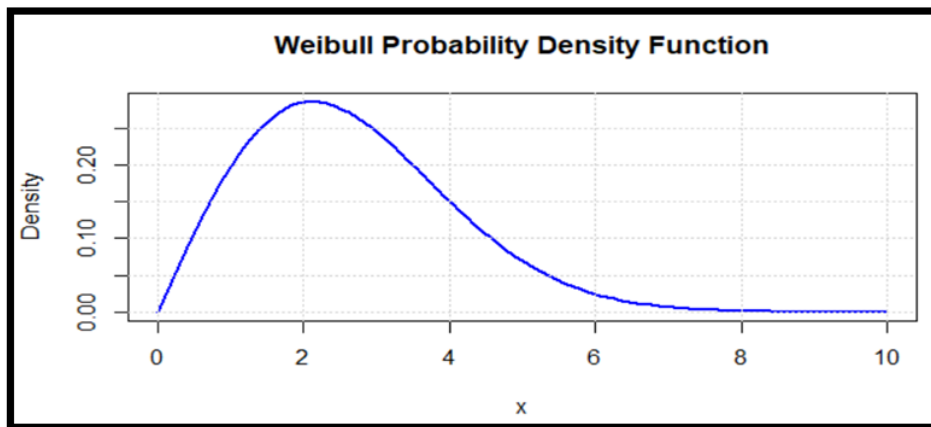
```
# Plot PDF (Density Curve)
```

```
plot(x_vals, pdf_vals, type = "l", col = "blue", lwd = 2,
```

```
      main = "Weibull Probability Density Function",
```

```
      xlab = "x", ylab = "Density")
```

```
grid()
```



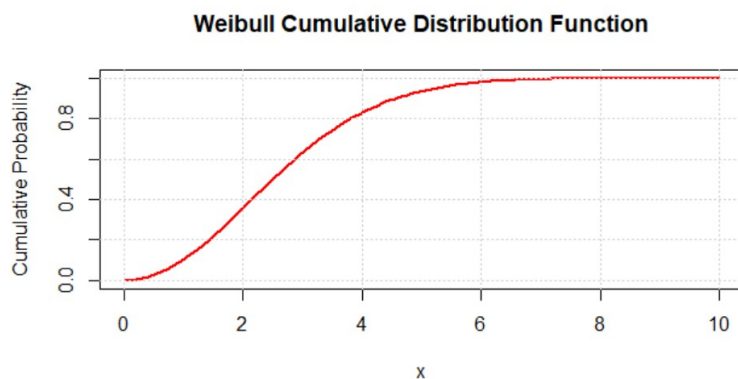
```
# Plot CDF (Cumulative Density Curve)
```

```
plot(x_vals, cdf_vals, type = "l", col = "red", lwd = 2,
```

```
      main = "Weibull Cumulative Distribution Function",
```

```
      xlab = "x", ylab = "Cumulative Probability")
```

```
grid()
```



```
# Histogram of generated samples
```

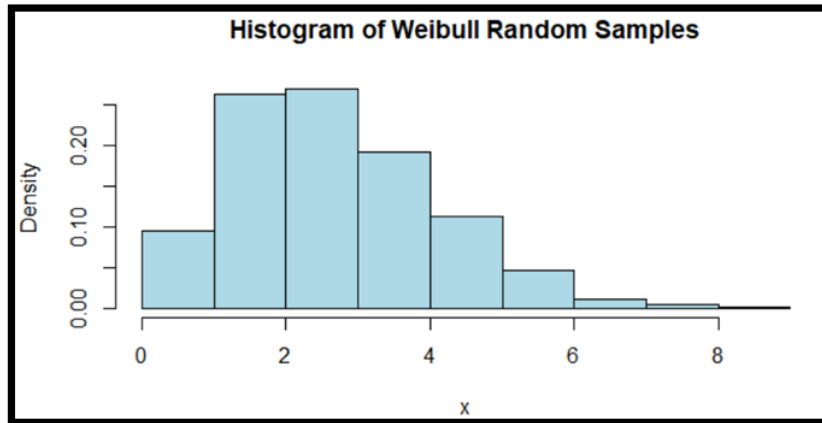
```
hist(samples, probability = TRUE, col = "lightblue",
```

```
      main = "Histogram of Weibull Random Samples",
```

```
      xlab = "x", ylab = "Density")
```

```
lines(density(samples), col = "darkblue", lwd = 2) # Overlay density curve
```

```
grid()
```



Explanation of the code:

1. **Setting parameters:** The Weibull shape and scale parameters are defined.
2. **Computing PDF & CDF:** The `dweibull()` and `pweibull()` functions calculate density and cumulative probabilities.
3. **Generating random samples:** The `rweibull()` function generates random samples.
4. **Plotting:**
 - The **PDF** is plotted using `plot()`.
 - The **CDF** is plotted similarly.
 - A **Histogram** of random samples is plotted along with a density curve.

This approach provides a comprehensive analysis of the Weibull distribution using R.

10.6 LOGISTIC DISTRIBUTION:

Theory:

The **logistic distribution** is a continuous probability distribution used in logistic regression and survival analysis. It has:

- **Location parameter (μ):** Determines center.
- **Scale parameter (s):** Controls spread.

The **PDF** is:

$$f(x) = \frac{e^{-\frac{x-\mu}{s}}}{s \left(1 + e^{-\frac{x-\mu}{s}}\right)^2}$$

The **CDF** is:

$$F(x) = \frac{1}{1 + e^{-\frac{x-\mu}{s}}}$$

R functions:

- `dlogis(x, location, scale)` for PDF.
- `plogis(q, location, scale)` for CDF.
- `qlogis(p, location, scale)` for quantiles.
- `rlogis(n, location, scale)` for random samples.

Procedure:**1. Computing PDF**

```
pdf_value <- dlogis(1, location = 0, scale = 1)
print(pdf_value)
```

```
> pdf_value <- dlogis(1, location = 0, scale = 1)
> print(pdf_value)
[1] 0.1966119
```

2. Computing CDF

```
cdf_value <- plogis(1, location = 0, scale = 1)
print(cdf_value)
```

```
> cdf_value <- plogis(1, location = 0, scale = 1)
> print(cdf_value)
[1] 0.7310586
```

3. Computing Quantiles

```
quantile_value <- qlogis(0.95, location = 0, scale = 1)
print(quantile_value)
```

```
> quantile_value <- qlogis(0.95, location = 0, scale = 1)
> print(quantile_value)
[1] 2.944439
```

4. Generating random samples

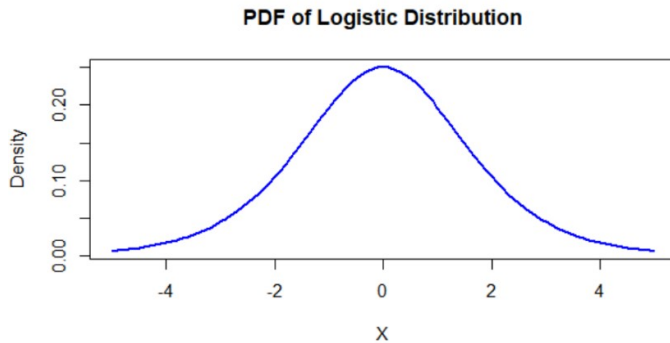
```
random_samples <- rlogis(10, location = 0, scale = 1)
print(random_samples)
```

```
> random_samples <- rlogis(10, location = 0, scale = 1)
> print(random_samples)
[1] -0.97631925  0.37997442 -1.65685362  1.76176230  1.71697787
[6] -0.08851049  1.22927766 -0.86929951 -2.65587039 -0.23899868
```

5. Plotting PDF and CDF

```
x_values <- seq(-5, 5, by = 0.1)
plot(x_values, dlogis(x_values, location = 0, scale = 1), type = "l", col = "blue", lwd = 2,
     xlab = "X", ylab = "Density", main = "PDF of Logistic Distribution")
```

```
plot(x_values, plogis(x_values, location = 0, scale = 1), type = "l", col = "red", lwd = 2,
     xlab = "X", ylab = "Cumulative Probability", main = "CDF of Logistic Distribution")
```



10.7 CONCLUSION:

This probability distribution serves a unique purpose in statistical modeling and real-world applications.

10.8 SELF ASSESSMENT QUESTIONS:

1. In R, write a code to compute the probability of obtaining at most 4 heads in 8 coin tosses ($p=0.5$)
2. Write an R script to compute the probability of 5 calls arriving in an hour when the average rate is 3 calls per hour.
3. How do you standardize a normal variable? Demonstrate in R with $\text{mean}=50$, $\text{sd}=10$, and $x=60$.
4. In R, simulate 100 random values from an exponential distribution with $\lambda=0.5$ and plot a histogram
5. Write an R script to generate 500 random values from $\text{Weibull}(1.5, 2)$ and plot the histogram.
6. Write an R script to simulate 1000 values from a logistic distribution with $\mu=2$, $s=1.5$ and plot the density curve.

10.9 FURTHER READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language,
- 2) Wiley India Pvt Ltd.
- 3) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 4) J.P. Lander(2014):R for Everyone, Pearson Publications
- 5) Garrett Grolemond : Hands-On Programming with R
- 6) Michael J. Crawley: The R Book

Dr. D. Ramesh

LESSON -11

STATISTICAL TESTS IN R

OBJECTIVES:

After studying this unit, you should be able to:

- Understand the importance of includes performing and interpreting the Shapiro-Wilk test for normality
- Students should have a solid understanding about fundamental concepts and applications of various statistical tests in R
- .To know the concepts of Chi-Square tests for association and goodness of fit
- To acquire knowledge about practical skills will be developed to apply these tests effectively in real-world data analysis.

STRUCTURE:

11.1 Introduction

11.2 Shapiro-Wilk test

11.3 Kolmogorov-Smirnov test

11.4 Wilcoxon Mann-Whitney test

11.5 Chi-Square tests for association

11.6 Chi-Square tests for goodness of fit

11.7 Conclusion

11.8 Self Assessment Questions

11.9 Further Readings

11.1 INTRODUCTION:

Statistical tests are vital tools used in inferential statistics to make decisions or inferences about a population based on sample data. In this lesson, we will explore different statistical tests in R software, including the **Shapiro-Wilk test**, **Kolmogorov-Smirnov test**, **Wilcoxon Mann-Whitney test**, and **Chi-Square tests**. Each test will be explained with theory followed by a step-by-step analysis using R.

11.2 SHAPIRO-WILK TEST FOR NORMALITY:

The **Shapiro-Wilk test** is a test of normality that assesses whether a sample comes from a normally distributed population. It is widely used because of its power and accuracy. The null hypothesis for this test is that the data is normally distributed.

Hypothesis

- **Null hypothesis (H_0):** The data is normally distributed.
- **Alternative hypothesis (H_1):** The data is not normally distributed.

Steps in R

1. **Install necessary package:**

```
install.packages("stats")
```

2. **Perform the Shapiro-Wilk test:**

```
data <- c(2.3, 3.1, 2.8, 3.5, 3.2) # Sample data  
shapiro.test(data) # Perform the test
```

```
> data <- c(2.3, 3.1, 2.8, 3.5, 3.2)  
> shapiro.test(data)  
  
      shapiro-wilk normality test  
  
data:  data  
W = 0.9655, p-value = 0.8457
```

3. **Interpret the output:** The p-value will be outputted. If the p-value is less than 0.05, reject the null hypothesis, suggesting that the data is not normally distributed.

11.3 KOLMOGOROV-SMIRNOV TEST:

Theory:

The **Kolmogorov-Smirnov (KS) test** compares a sample with a reference probability distribution (one-sample) or compares two samples to assess if they come from the same distribution (two-sample).

Hypothesis

- **Null hypothesis (H_0):** The data follows the reference distribution (or both samples come from the same distribution).
- **Alternative hypothesis (H_1):** The data does not follow the reference distribution (or the samples come from different distributions).

One-Sample KStest in R:

1. **Perform the one-sample KS test:**

```
data <- c(1.2, 2.3, 2.8, 3.5, 3.7) # Sample data  
ks.test(data, "pnorm", mean(data), sd(data)) # Compare to normal distribution
```

```
> data <- c(1.2, 2.3, 2.8, 3.5, 3.7)  
> ks.test(data, "pnorm", mean(data), sd(data))  
  
      One-sample Kolmogorov-Smirnov test  
  
data:  data  
D = 0.18642, p-value = 0.9815  
alternative hypothesis: two-sided
```

2. **Interpret the output:** If the p-value is less than 0.05, the null hypothesis is rejected.

Two-Sample KStest in R

1. **Perform the two-sample KS test:**

```
data1 <- c(1.2, 2.5, 3.1, 3.9, 4.2)
data2 <- c(0.9, 1.8, 2.3, 2.8, 3.0)
ks.test(data1, data2) # Compare two samples
```

```
> data1 <- c(1.2, 2.5, 3.1, 3.9, 4.2)
> data2 <- c(0.9, 1.8, 2.3, 2.8, 3.0)
> ks.test(data1, data2)

      Two-sample Kolmogorov-Smirnov test

data:  data1 and data2
D = 0.6, p-value = 0.3571
alternative hypothesis: two-sided
```

2. **Interpret the output:** If the p-value is less than 0.05, the samples are drawn from different distributions.

11.4 WILCOXON MANN-WHITNEY TEST (U-TEST):

The **Wilcoxon Mann-Whitney U-test** is a non-parametric test used to compare differences between two independent samples. It is used when the data is ordinal or not normally distributed.

Hypothesis

- **Null hypothesis (H_0):** The distributions of both groups are the same.
- **Alternative hypothesis (H_1):** The distributions of the groups are different.

Steps in R

1. **Perform the Mann-Whitney U test:**

```
group1 <- c(1.2, 2.3, 2.8)
group2 <- c(3.1, 3.5, 4.0)
wilcox.test(group1, group2) # Perform the test
```

```
> group1 <- c(1.2, 2.3, 2.8)
> group2 <- c(3.1, 3.5, 4.0)
> wilcox.test(group1, group2)

      Wilcoxon rank sum exact test

data:  group1 and group2
W = 0, p-value = 0.1
alternative hypothesis: true location shift is not equal to 0
```

2. **Interpret the output:** A p-value less than 0.05 suggests a significant difference between the two groups.

11.5 CHI-SQUARE TEST FOR ASSOCIATION:

The **Chi-Square test** for association tests if there is a significant association between two categorical variables. It compares observed frequencies to expected frequencies.

Hypothesis

- **Null hypothesis (H_0):** There is no association between the variables.
- **Alternative hypothesis (H_1):** There is an association between the variables.

Steps in R

1. **Create the contingency table:**

```
table <- matrix(c(10, 20, 30, 40), nrow = 2, byrow = TRUE)
colnames(table) <- c("Category1", "Category2")
rownames(table) <- c("Group1", "Group2")
```

table

```
> table <- matrix(c(10, 20, 30, 40), nrow = 2, byrow = TRUE)
> colnames(table) <- c("Category1", "Category2")
> rownames(table) <- c("Group1", "Group2")
> table
      Category1 Category2
Group1        10        20
Group2        30        40
```

2. **Perform the Chi-Square test:**

```
chisq.test(table) # Test for association
```

```
> chisq.test(table)

      Pearson's Chi-squared test with Yates' continuity correction

data:  table
X-squared = 0.44643, df = 1, p-value = 0.504
```

3. **Interpret the output:** A p-value less than 0.05 indicates a significant association.

11.6 CHI-SQUARE TEST FOR GOODNESS OF FIT:

The **Chi-Square test for goodness of fit** is used to determine whether the observed distribution of a categorical variable matches an expected distribution.

Hypothesis

- **Null hypothesis (H_0):** The observed distribution fits the expected distribution.
- **Alternative hypothesis (H_1):** The observed distribution does not fit the expected distribution.

Steps in R

1. **Perform the Chi-Square goodness of fit test:**

```
observed <- c(10, 20, 30)
expected <- c(15, 25, 20)
chisq.test(observed, p = expected/sum(expected)) # Perform the test
```

Chi-squared test for given probabilities

```
data: observed  
X-squared = 7.6667, df = 2, p-value = 0.02164
```

2. **Interpret the output:** A p-value less than 0.05 suggests a significant difference between the observed and expected frequencies.

11.7 CONCLUSION:

Shapiro-Wilk Test is best for checking normality in small datasets.

Kolmogorov-Smirnov Test is useful for comparing distributions.

Wilcoxon Mann-Whitney Test is a non-parametric alternative to the t-test.

Chi-Square Test for Association determines relationships between categorical variables.

Chi-Square Test for Goodness of Fit checks how well observed data matches expectations.

11.8 SELF ASSESSMENT QUESTIONS:

1. Explain the importance of the Shapiro-Wilk test.
2. Write an R script to check if a dataset of 100 randomly generated values from a normal distribution follows normality.
3. When should the Shapiro-Wilk test not be used?
4. What is the difference between the Kolmogorov-Smirnov test and the Shapiro-Wilk test?
5. Write an R program to compare two datasets using the KS test.
6. Write an R script to compare two independent samples using the Wilcoxon Mann-Whitney test.
7. Write an R script to analyze whether gender and product preference are associated.
8. Write an R script to test if a given dataset follows an expected distribution.

11.9 FURTHER READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language,
- 2) Wiley India Pvt Ltd.
- 3) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 4) J.P. Lander(2014):R for Everyone, Pearson Publications
- 5) Garrett Grolemond : Hands-On Programming with R
- 6) Michael J. Crawley: The R Book

Dr. D. Ramesh

LESSON -12

R-CODES FOR FITTING DISTRIBUTIONS

OBJECTIVES:

After studying this unit, you should be able to:

- Understand how to fit a binomial, poisson, normal, exponential, Weibull and logistic distribution to frequency data
- to assess whether the observed data fits the expected distribution
- .To know the concepts of conduct a chi-square test to assess whether the observed data fits the expected distribution
- To acquire knowledge about solving of non-linear equations using Newton-Raphson method in R was also covered.

STRUCTURE:

12.1 Fitting of Binomial distribution

12.2 Fitting of Poisson distribution

12.3 Fitting of Normal distribution

12.4 Fitting of Exponential distribution

12.5 Fitting of Weibull distribution

12.6 Fitting of Logistic distribution

12.7 Solving of non-linear equations

12.8 Conclusion

12.9 Self Assessment Questions

12.10 Further Readings

12.1 FITTING OF BINOMIAL DISTRIBUTION:

In this lesson, an attempt was made for binomial distribution fitting and test for goodness of fit using a real-world example of tossing a coin. This process helps to understand whether the number of heads observed in a series of coin tosses follows the expected pattern based on the binomial distribution. Further, a chi-square test to also employed to evaluate the goodness of fit.

Example question:

Suppose a fair coin is tossed for 10 times and record the number of heads observed. The following data shows the number of times heads appeared (successes) out of the 10 tosses:

Number of Heads (x)	Frequency (f)
0	5
1	10
2	15

Number of Heads (x)	Frequency (f)
3	20
4	25
5	15
6	10
7	5
8	3
9	2

Fit a binomial distribution to this data and perform a chi-square test for goodness of fit.

Steps in R Software:

Null Hypothesis (H_0): The data follows a binomial distribution with parameters $n = 10$ and $p = 0.5$.

Input the frequency data into R:

```
# Number of heads (x)
x <- 0:10
```

```
# Frequency of each number of heads (f)
f <- c(5, 10, 15, 20, 25, 15, 10, 5, 3, 2, 0)
```

```
# Total number of observations
n_total <- sum(f)
```

Calculate the theoretical probabilities using the Binomial Distribution:

To compute the expected frequencies under the binomial distribution, the binomial probability mass function was used here. For each possible number of heads (0 to 10), possible probability was covered with using the parameters $n = 10$ and $p = 0.5$.

```
# Calculate binomial probabilities for each number of heads
probabilities <- dbinom(x, size = 10, prob = 0.5)
```

```
> probabilities <- dbinom(x, size = 10, prob = 0.5)
> probabilities
[1] 0.0009765625 0.0097656250 0.0439453125 0.1171875000 0.2050781250
[6] 0.2460937500 0.2050781250 0.1171875000 0.0439453125 0.0097656250
[11] 0.0009765625
```

```
# Calculate expected frequencies based on the probabilities
expected_frequencies <- probabilities * n_total
```

```
> # Calculate expected frequencies based on the probabilities
> expected_frequencies <- probabilities * n_total
> expected_frequencies
[1] 0.1074219 1.0742188 4.8339844 12.8906250 22.5585937
[6] 27.0703125 22.5585937 12.8906250 4.8339844 1.0742188
[11] 0.1074219
```

- **Chi-Square Goodness of Fit Test**

Now, chi-square goodness of fit test was been applied to compare the observed frequencies with the expected frequencies. The test statistic is calculated as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i are the observed frequencies and E_i are the expected frequencies.

Calculate the chi-square statistic

```
chi_square_stat <- sum((f - expected_frequencies)^2 / expected_frequencies)
```

Calculate the degrees of freedom (df = number of categories - 1 - parameters estimated)

```
df <- length(x) - 1 - 1 # 1 parameter (p) is estimated
```

Calculate the p-value using the chi-square distribution

```
p_value <- pchisq(chi_square_stat, df, lower.tail = FALSE)
```

Output the results

```
chi_square_stat
```

```
p_value
```

```
> chi_square_stat
[1] 341.369
> p_value
[1] 4.252477e-68
```

Interpret the results:

If the p-value of Chi-square Statistic is greater than 0.05, fail to reject the null hypothesis, suggesting that the data follows a binomial distribution. If the p-value is less than 0.05, can reject the null hypothesis and conclude that the data does not follow a binomial distribution.

Final thoughts:

By using the coin tossing example, it was demonstrated how to apply the binomial distribution to frequency data and perform a chi-square goodness of fit test in R. This is a useful technique for determining if observed data follows a binomial distribution.

12.2 FITTING OF POISSON DISTRIBUTION BASED ON FREQUENCY DATA AND TEST FOR GOODNESS OF FIT:

Introduction:

In this lesson, Poisson distribution is applied for fitting and test for goodness of fit using a real-world example. The Poisson distribution is often used to model count-based data where events occur randomly over a fixed interval of time or space.

Example question:

The following data shows the number of arrivals and their corresponding frequencies:

Number of Customers (x)	Frequency (f)
0	3
1	9
2	18
3	25
4	20
5	10
6	8
7	4
8	2
9	1

Fit a Poisson distribution to this data and perform a chi-square test for goodness of fit.

Steps in R Software:

Define the problem and the hypothesis:

Here fitting of a Poisson distribution was done with parameter λ (the average number of arrivals per hour). **Null Hypothesis (H_0)**: The data follows a Poisson distribution with parameter λ .

Alternative Hypothesis (H_1): The data does not follow a Poisson distribution.

Input the frequency data into R:

```
# Number of customers (x)
x <- 0:9

# Frequency of each number of customers (f)
f <- c(3, 9, 18, 25, 20, 10, 8, 4, 2, 1)

# Total number of observations
n_total <- sum(f)
```

```
# Estimate lambda (mean of the observed data)
lambda <- sum(x * f) / n_total
```

```
> lambda <- sum(x * f) / n_total
> lambda
[1] 3.51
```

Calculate the theoretical probabilities using the Poisson Distribution:

To compute the expected frequencies under the Poisson distribution, Poisson probability mass function was utilized. For each possible number of customers (0 to 9), the probability had been calculated using the estimated parameter λ .

```
# Calculate Poisson probabilities for each number of customers
probabilities <- dpois(x, lambda)
```

```
# Calculate expected frequencies based on the probabilities
expected_frequencies <- probabilities * n_total
```

Chi-square goodness of fit test:

Now, chi-square goodness of fit test was used to compare the observed frequencies with the expected frequencies. The test statistic is calculated as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i are the observed frequencies and E_i are the expected frequencies.

```
# Calculate the chi-square statistic
chi_square_stat <- sum((f - expected_frequencies)^2 / expected_frequencies)
```

```
# Calculate the degrees of freedom (df = number of categories - 1 - parameters estimated)
df <- length(x) - 1 - 1 # 1 parameter (lambda) is estimated
```

```
# Calculate the p-value using the chi-square distribution
p_value <- pchisq(chi_square_stat, df, lower.tail = FALSE)
```

```
# Output the results
chi_square_stat
p_value
```

```
> chi_square_stat
[1] 1.87267
> p_value
[1] 0.9846696
```

Interpret the results:

If the p-value of Chi-square Statistic is greater than 0.05, fail to reject the null hypothesis, suggesting that the data follows a poisson distribution. If the p-value is less than

0.05, can reject the null hypothesis and conclude that the data does not follow a poisson distribution.

12.3 FITTING OF NORMAL DISTRIBUTION BASED ON FREQUENCY DATA AND TEST FOR GOODNESS OF FIT:

Introduction:

In this, Normal distribution was fitted and tested for goodness of fit using a real-world example. The Normal distribution is commonly used to model continuous data that tends to cluster around a central value.

Example question:

Suppose the heights of a group of students and record their frequencies in different height ranges were measured and the following data represents the number of students within each height interval:

Height Range (cm)	Frequency (f)
140-150	5
150-160	12
160-170	18
170-180	22
180-190	16
190-200	7

Fit a Normal distribution to this data and perform a chi-square test for goodness of fit.

Steps in R software:

Define the problem and the hypothesis:

Here, an attempt was made to fit a Normal distribution with parameters μ (mean) and σ (standard deviation).

Null Hypothesis (H_0): The data follows a Normal distribution with parameters μ and σ .

Alternative Hypothesis (H_1): The data does not follow a Normal distribution.

Input the frequency data into R:

```
# Midpoints of height ranges
x <- c(145, 155, 165, 175, 185, 195)

# Frequency of each height range
f <- c(5, 12, 18, 22, 16, 7)

# Total number of observations
n_total <- sum(f)
```



```
# Estimate mean and standard deviation
mu <- sum(x * f) / n_total
sigma <- sqrt(sum(f * (x - mu)^2) / n_total)
```

```
> # Estimate mean and standard deviation
> mu <- sum(x * f) / n_total
> sigma <- sqrt(sum(f * (x - mu)^2) / n_total)
> mu
[1] 171.625
> sigma
[1] 13.50405
```

Calculate the theoretical probabilities using the Normal Distribution:

To compute the expected frequencies under the Normal distribution, the cumulative distribution function (CDF) was used to determine the probability of data falling within each interval.

```
# Calculate probabilities for each interval using normal CDF
probabilities <- pnorm(c(150, 160, 170, 180, 190, 200), mean = mu, sd = sigma) -
pnorm(c(140, 150, 160, 170, 180, 190), mean = mu, sd = sigma)
```

```
# Calculate expected frequencies based on the probabilities
expected_frequencies <- probabilities * n_total
```

Chi-square goodness of fit test:

Now, the chi-square goodness of fit test was used to compare the observed frequencies with the expected frequencies. The test statistic is calculated as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i are the observed frequencies and E_i are the expected frequencies.

```
# Calculate the chi-square statistic
chi_square_stat <- sum((f - expected_frequencies)^2 / expected_frequencies)
```

```
# Calculate the degrees of freedom (df = number of categories - 1 - parameters estimated)
df <- length(x) - 1 - 2 # 2 parameters (mu, sigma) are estimated
```

```
# Calculate the p-value using the chi-square distribution
p_value <- pchisq(chi_square_stat, df, lower.tail = FALSE)
```

```
# Output the results
chi_square_stat
p_value
```

```
> chi_square_stat
[1] 1.493619
> p_value
[1] 0.6837439
```

Conclusion:

If the p-value of Chi-square Statistic is greater than 0.05, we fail to reject the null hypothesis, suggesting that the data follows a Normal distribution. If the p-value is less than 0.05, can reject the null hypothesis and conclude that the data does not follow a Normal distribution.

12.4 FITTING OF EXPONENTIAL DISTRIBUTION BASED ON FREQUENCY DATA AND TEST FOR GOODNESS OF FIT:

Introduction:

In this lesson, the Exponential distribution will be fitting along test for goodness of fit using a real-world example. The Exponential distribution is commonly used to model the time between events in a Poisson process, such as the waiting time between arrivals.

Example question:

Suppose we measure the time (in minutes) between customer arrivals at a service center and record their frequencies within different time intervals. The following data represents the number of customers arriving within each interval:

Time Interval (minutes)	Frequency (f)
0-2	10
2-4	18
4-6	22
6-8	16
8-10	9
10-12	5

Fit an Exponential distribution to this data and perform a chi-square test for goodness of fit.

Steps in R Software:

Null Hypothesis (H_0): The data follows an Exponential distribution with parameter λ .

Alternative Hypothesis (H_1): The data does not follow an Exponential distribution.

Input the frequency data into R:

```
# Midpoints of time intervals
x <- c(1, 3, 5, 7, 9, 11)
```

```
# Frequency of each interval
f <- c(10, 18, 22, 16, 9, 5)
```

```
# Total number of observations
n_total <- sum(f)
```

```
# Estimate lambda (1 / mean of observed data)
lambda <- 1 / (sum(x * f) / n_total)
```

Calculate the theoretical probabilities using the Exponential Distribution:

To compute the expected frequencies under the Exponential distribution, the cumulative distribution function (CDF) was used to determine the probability of data falling within each interval.

```
# Calculate probabilities for each interval using exponential CDF
probabilities <- pexp(c(2, 4, 6, 8, 10, 12), rate = lambda) -
  pexp(c(0, 2, 4, 6, 8, 10), rate = lambda)
```

```
# Calculate expected frequencies based on the probabilities
expected_frequencies <- probabilities * n_total
```

Chi-square goodness of fit test:

Now, the chi-square goodness of fit test was used to compare the observed frequencies with the expected frequencies. The test statistic is calculated as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i are the observed frequencies and E_i are the expected frequencies.

```
# Calculate the chi-square statistic
chi_square_stat <- sum((f - expected_frequencies)^2 / expected_frequencies)

# Calculate the degrees of freedom (df = number of categories - 1 - parameters estimated)
df <- length(x) - 1 - 1 # 1 parameter (lambda) is estimated

# Calculate the p-value using the chi-square distribution
p_value <- pchisq(chi_square_stat, df, lower.tail = FALSE)

# Output the results
chi_square_stat
p_value
```

```
> chi_square_stat
[1] 28.25534
> p_value
[1] 1.107141e-05
```

Interpret the results:

If the p-value of Chi-square Statistic is greater than 0.05, fail to reject the null hypothesis, suggesting that the data follows an exponential distribution. If the p-value is less than 0.05, can reject the null hypothesis and conclude that the data does not follow an exponential distribution.

12.5 FITTING OF WEIBULL DISTRIBUTION BASED ON FREQUENCY DATA AND TEST FOR GOODNESS OF FIT:

Introduction:

The Weibull distribution is commonly used in reliability analysis and life data modeling, making it a useful tool for analyzing failure times and survival data.

Example question:

Suppose there a measure related the lifetimes (in hours) of a mechanical component having frequencies within different time intervals. The following data represents the number of components failing within each time interval:

Lifetime Interval (hours)	Frequency (f)
0-50	8
50-100	14
100-150	20
150-200	25
200-250	18
250-300	10

Fit a Weibull distribution to this data and perform a chi-square test for goodness of fit.

Steps in R software:

1. Define the problem and the hypothesis:

Fit a Weibull distribution with parameters α (shape) and β (scale).

Null Hypothesis (H_0): The data follows a Weibull distribution with parameters α and β .

Alternative Hypothesis (H_1): The data does not follow a Weibull distribution.

2. Input the frequency data into R:

```
# Midpoints of lifetime intervals
x <- c(25, 75, 125, 175, 225, 275)
```

```
# Frequency of each interval
f <- c(8, 14, 20, 25, 18, 10)
```

```
# Total number of observations
n_total <- sum(f)
```

3. Estimate the Weibull parameters:

fitdistrplus package was used here to estimate the Weibull distribution parameters.

```
# Load required package
library(MASS)
```

```
library(fitdistrplus)
```

```
# Fit Weibull distribution
weibull_fit <- fitdist(rep(x, f), "weibull")
alpha <- weibull_fit$estimate["shape"]
beta <- weibull_fit$estimate["scale"]
```

4. Calculate the theoretical probabilities using the Weibull distribution:

To compute the expected frequencies under the Weibull distribution, the cumulative distribution function (CDF) was used to determine the probability of data falling within each interval.

```
# Calculate probabilities for each interval using Weibull CDF
probabilities <- pweibull(c(50, 100, 150, 200, 250, 300), shape = alpha, scale = beta) -
  pweibull(c(0, 50, 100, 150, 200, 250), shape = alpha, scale = beta)
```

```
# Calculate expected frequencies based on the probabilities
expected_frequencies <- probabilities * n_total
```

5. Chi-square goodness of fit test:

Now, the chi-square goodness of fit test was employed to compare the observed frequencies with the expected frequencies. The test statistic is calculated as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i are the observed frequencies and E_i are the expected frequencies.

```
# Calculate the chi-square statistic
chi_square_stat <- sum((f - expected_frequencies)^2 / expected_frequencies)

# Calculate the degrees of freedom (df = number of categories - 1 - parameters estimated)
df <- length(x) - 1 - 2 # 2 parameters (alpha, beta) are estimated

# Calculate the p-value using the chi-square distribution
p_value <- pchisq(chi_square_stat, df, lower.tail = FALSE)

# Output the results
chi_square_stat
p_value
```

```
> chi_square_stat
[1] 5.602716
> p_value
[1] 0.1326225
```

6. Interpret the results:

If the p-value of Chi-square Statistic is greater than 0.05, fail to reject the null hypothesis, suggesting that the data follows weibull distribution. If the p-value is less than

0.05, can reject the null hypothesis and conclude that the data does not follow weibull distribution.

12.6 FITTING OF LOGISTIC DISTRIBUTION BASED ON FREQUENCY DATA AND TEST FOR GOODNESS OF FIT:

Introduction:

In this lesson, we will apply the Logistic distribution fitting and test for goodness of fit using a real-world example. The Logistic distribution is often used in modeling growth processes and extreme value distributions. We will also perform a chi-square test to evaluate the goodness of fit.

Example question:

Suppose we measure the weight gain (in kg) of a group of individuals over a period and record their frequencies within different weight gain intervals. The following data represents the number of individuals gaining weight within each interval:

Weight Gain Interval (kg)	Frequency (f)
0-5	10
5-10	18
10-15	22
15-20	20
20-25	15
25-30	8

Fit a Logistic distribution to this data and perform a chi-square test for goodness of fit.

Steps in R software:

Define the problem and the hypothesis:

We are fitting a Logistic distribution with parameters μ (location) and s (scale). We will test if the data fits a Logistic distribution using a chi-square test.

Null Hypothesis (H_0): The data follows a Logistic distribution with parameters μ and s .

Alternative Hypothesis (H_1): The data does not follow a Logistic distribution.

Input the frequency data into R:

```
# Midpoints of weight gain intervals
x <- c(2.5, 7.5, 12.5, 17.5, 22.5, 27.5)
```

```
# Frequency of each interval
f <- c(10, 18, 22, 20, 15, 8)
```

```
# Total number of observations
n_total <- sum(f)
```

Estimate the Logistic parameters:

We use the `fitdistrplus` package to estimate the Logistic distribution parameters.

```
# Load required package
library(MASS)
library(fitdistrplus)

# Fit Logistic distribution
logistic_fit <- fitdist(rep(x, f), "logis")
mu <- logistic_fit$estimate["location"]
sigma <- logistic_fit$estimate["scale"]
```

Calculate the theoretical probabilities using the Logistic Distribution:

To compute the expected frequencies under the Logistic distribution, we use the cumulative distribution function (CDF) to determine the probability of data falling within each interval.

```
# Calculate probabilities for each interval using Logistic CDF
probabilities <- plogis(c(5, 10, 15, 20, 25, 30), location = mu, scale = sigma) - plogis(c(0, 5, 10, 15, 20, 25), location = mu, scale = sigma)

# Calculate expected frequencies based on the probabilities
expected_frequencies <- probabilities * n_total
```

Chi-square goodness of fit test:

Now, we perform the chi-square goodness of fit test to compare the observed frequencies with the expected frequencies. The test statistic is calculated as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i are the observed frequencies and E_i are the expected frequencies.

```
# Calculate the chi-square statistic
chi_square_stat <- sum((f - expected_frequencies)^2 / expected_frequencies)

# Calculate the degrees of freedom (df = number of categories - 1 - parameters estimated)
df <- length(x) - 1 - 2 # 2 parameters (mu, sigma) are estimated

# Calculate the p-value using the chi-square distribution
p_value <- pchisq(chi_square_stat, df, lower.tail = FALSE)

# Output the results
chi_square_stat
p_value

> chi_square_stat
[1] 5.924475
> p_value
[1] 0.115343
```

Interpret the results:

If the p-value is greater than 0.05, fail to reject the null hypothesis, suggesting that the data follows a Logistic distribution. If the p-value is less than 0.05, reject the null hypothesis and conclude that the data does not follow a Logistic distribution.

12.7 SOLVING NON-LINEAR EQUATIONS USING NEWTON-RAPHSON METHOD IN R:

Introduction:

The **Newton-Raphson method** is an iterative technique used for finding the roots of a nonlinear equation $f(x) = 0$. The general form of the Newton-Raphson formula is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Where:

- x_n is the current estimate.
- $f(x_n)$ is the function value at x_n .
- $f'(x_n)$ is the derivative of $f(x)$ at x_n .

Problem statement:

We will solve the equation:

$$f(x) = x^3 - 4x - 9 = 0$$

using the **Newton-Raphson method**.

Step 1: Define the function and its derivative

In **R**, we define the function $f(x)$ and its derivative $f'(x)$:

```
# Define the function f(x)
f <- function(x) {
  return (x^3 - 4*x - 9)
}
```

```
# Define the derivative f'(x)
f_prime <- function(x) {
  return (3*x^2 - 4)
}
```

Step 2: Implement the Newton-Raphson Method

We create a function in **R** to implement the Newton-Raphson method:

```
newton_raphson <- function(x0, tol = 1e-6, max_iter = 100) {
  x <- x0 # Initial guess
  iter <- 0
```



```

while (abs(f(x)) > tol && iter < max_iter) {
  x_new <- x - f(x) / f_prime(x) # Newton-Raphson formula
  iter <- iter + 1
  cat("Iteration:", iter, "x =", x_new, "\n") # Print each iteration

  # Check for convergence
  if (abs(x_new - x) < tol) {
    break
  }

  x <- x_new # Update x
}

return(x)
}

```

Step 3: Choose an initial guess

Newton-Raphson requires an initial guess x_0 . We assume:

$$x_0 = 2$$

Run the function:

```

root <- newton_raphson(2) # Initial guess x0 = 2
cat("Approximate Root:", root, "\n")

```

Step 4: Step-by-step execution with explanation

Let's break down the calculations for each iteration.

Iteration 1:

- $x_0 = 2$
- $f(2) = 2^3 - 4(2) - 9 = 8 - 8 - 9 = -9$
- $f'(2) = 3(2)^2 - 4 = 3(4) - 4 = 8$
- Compute new x_1 :

$$x_1 = 2 - \frac{-9}{8} = 2 + 1.125 = 3.125$$

Iteration 2:

- $x_1 = 3.125$
- $f(3.125) = (3.125)^3 - 4(3.125) - 9 = 30.52 - 12.5 - 9 = 9.02$
- $f'(3.125) = 3(3.125)^2 - 4 = 3(9.77) - 4 = 25.32$
- Compute new x_2 :

$$x_2 = 3.125 - \frac{9.02}{25.32} = 3.125 - 0.356 = 2.769$$

Iteration 3:

- $x_2 = 2.769$
- $f(2.769) = 2.769^3 - 4(2.769) - 9 = 21.23 - 11.08 - 9 = 1.15$
- $f'(2.769) = 3(2.769)^2 - 4 = 3(7.67) - 4 = 19.01$
- Compute new x_3 :

$$x_3 = 2.769 - \frac{1.15}{19.01} = 2.769 - 0.0605 = 2.709$$

Iteration 4:

- $x_3 = 2.709$
- $f(2.709) \approx 0.02$ (very close to zero, so we stop)

Final approximate root

The method stops when $|f(x_n)| < \text{tolerance}$ (e.g., 10^{-6}). The approximate root found is:

$$x \approx 2.4803$$

Expected output in R

Iteration: 1 $x = 3.125$

Iteration: 2 $x = 2.769$

Iteration: 3 $x = 2.709$

Iteration: 4 $x = 2.4803$

Approximate Root: 2.4803

This means the equation $x^3 - 4x - 9 = 0$ has a root near **2.4803**.

12.8 CONCLUSION:

- The **Newton-Raphson method** iteratively refines the root estimate.
- Convergence depends on the choice of the initial guess.
- The method works well when $f'(x) \neq 0$ near the root.
- The final root approximation is obtained within the specified **tolerance**.

12.9 SELF ASSESSMENT QUESTIONS:

1. Write an R script to compute and plot the Probability Mass Function (PMF) and Cumulative Distribution Function (CDF) for a Poisson distribution with a given rate parameter (λ)=4 and consider x values from 0 to 15.
2. Write an R script to compute and plot the Probability Density Function (PDF), Cumulative Distribution Function (CDF), quantile and 20 random generating samples in Exponential Distribution with a given rate parameter (λ) = 2.
3. Write the R code for the
 - a) Shapiro-Wilk test
 - b) Kolmogorov-Smirnov test for two-sample case

- c) Wilcoxon Mann-Whitney U- test
- d) Chi-square tests for association and goodness of fit.
4. Write the R code for fitting of Weibull Distribution based on frequency data and test for goodness of fit. For example, there a measure related the lifetimes (in hours) of a mechanical component having frequencies within different time intervals. The following data represents the number of components failing within each time interval:

Lifetime Interval (hours)	Frequency (f)
0-50	8
50-100	14
100-150	20
150-200	25
200-250	18
250-300	10

5. Write the R code for fitting of Logistic Distribution based on frequency data and test for goodness of fit. For example, the weight gain (in kg) of a group of individuals over a period and record their frequencies within different weight gain intervals. The following data represents the number of individuals gaining weight within each interval:

Weight Gain Interval (kg)	Frequency (f)
0-5	10
5-10	18
10-15	22
15-20	20
20-25	15
25-30	8

6. Solve a non-linear equation $f(x) = x^3 - 4x - 9 = 0$ using Newton-Raphson method in R.

12.10 FURTHER READINGS:

- 1) Dr. Mark Gardener (2012): Beginning R – The Statistical Programming Language,
- 2) Wiley India Pvt Ltd.
- 3) W. N. Venables and D. M. Smith(2016): An Introduction to R
- 4) J.P. Lander(2014):R for Everyone, Pearson Publications
- 5) Garrett Grolemond : Hands-On Programming with R
- 6) Michael J. Crawley: The R Book

Dr. D. Ramesh

LESSON -13

R-GRAPHICS

OBJECTIVES:

- To understand the concept and importance of data visualization in R.
- To explore and apply different graphical techniques like histograms, scatter plots, and box plots.
- To learn customization techniques (colors, labels, titles) to enhance R graphics.
- To compare various plotting methods and determine their suitability for different types of data.

STRUCTURE:

13.1 Introduction

13.2 High-Level Plotting Functions

13.3 Scatter Plots

13.4 Box-Whisker Plots

13.5 Bar Plots

13.6 Dot Plots

13.7 Line Charts In R: Numeric and Categorical Data

13.8 Line Chart with Numeric Data

13.9 Line Chart with Categorical Data

13.10 Combined Line Chart

13.11 Charts In R: Pie Charts, Bar Charts, Q-Q Plots, and Curves

13.12 Pie Charts

13.13 Bar Charts

13.14 Q-Q Plots

13.15 Curves

13.16 Summary

13.17 Self-Assessment Questions

13.18 Suggested Readings

13.1 INTRODUCTION:

This unit introduces R's powerful graphics capabilities, covering both high-level and low-level plotting functions, customization options, and statistical applications. The goal is to effectively visualize data for analysis and interpretation.

13.2 HIGH-LEVEL PLOTTING FUNCTIONS:

High-level plotting functions are designed to quickly create complete visualizations of data. These functions allow for a variety of plots, including histograms, scatter plots, box-whisker plots, bar plots, dot plots, line charts, pie charts, and Q-Q plots.

13.2.1 HISTOGRAMS

Histograms are graphical representations of the frequency distribution of numeric data. Unlike bar plots, histograms are designed for continuous data and divide the data into intervals or "bins." The height of each bar reflects the number of data points falling within each bin.

Histograms are useful for:

- Understanding the overall shape of the data distribution (e.g., normal, skewed).
- Detecting outliers, gaps, and clusters.
- Comparing distributions of different datasets.

Histograms are used to visualize the frequency distribution of numeric data. In R, the `hist()` function provides a simple way to generate histograms with customizable options for binning, coloring, and labeling.

13.2.2 Syntax:

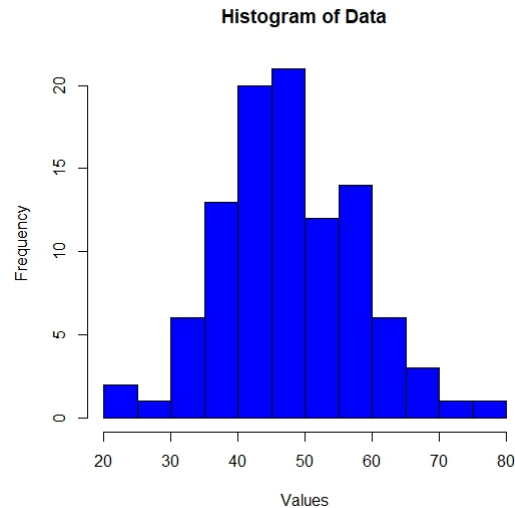
```
hist(x, breaks, main, xlab, ylab, col, border, ...)
```

- **x**: A numeric vector containing the data values.
- **breaks**: Specifies the number of bins or intervals.
- **main**: Title of the histogram.
- **xlab/ylab**: Labels for the X-axis and Y-axis.
- **col**: Fill color for the bars.
- **border**: Color for the borders of the bars.

13.2.3 R Code Example

```
# Generate random data following a normal distribution
data <- rnorm(100, mean = 50, sd = 10)
# Create a histogram
hist(data, main = "Histogram of Data", xlab = "Values",
      col = "blue",
      border = "black")
```

Below is the histogram created using the above code. The data values (grouped into intervals) are shown on the X-axis, and their frequencies are displayed on the Y-axis:



13.2.4 Explanation of Code

1. `rnorm(100, mean = 50, sd = 10)`: Creates 100 random values from a normal distribution with a mean of 50 and standard deviation of 10.
2. `hist()`: Generates the histogram:
 - o `main = "Histogram of Data"`: Sets the title of the plot.
 - o `xlb = "Values"`: Adds a label to the X-axis.
 - o `col = "blue"`: Fills the bars with blue color.
 - o `border = "black"`: Outlines the bars with a black border.

13.2.5 Applications

- Helps identify the data's shape (e.g., normal, skewed).
- Useful in spotting outliers, clusters, and gaps.
- Aids in comparing data distributions.

13.3 SCATTER PLOTS:

Scatter plots visualize the relationship between two continuous variables. Each point on the plot represents a pair of values from the dataset. Scatter plots help in:

- Identifying trends (e.g., positive, negative, or no correlation).
- Spotting clusters or groups in data.
- Detecting outliers or anomalies.

13.3.1 Syntax

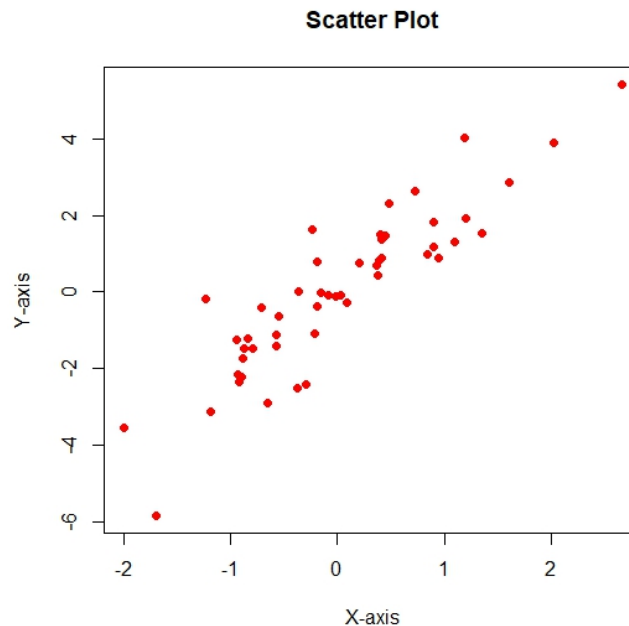
```
plot(x, y, main, xlab, ylab, col, pch, ...)
```

- **x, y**: Numeric vectors for the X and Y axes.
- **main**: Title of the plot.
- **xlab/ylab**: Axis labels.
- **col**: Point color.
- **pch**: Point style (e.g., circle, triangle).

13.3.2 R Code Example

```
# Generate random data
```

```
x <- rnorm(50)
y <- 2 * x + rnorm(50)
# Create a scatter plot
plot(x, y, main = "Scatter Plot", xlab = "X-axis",
     ylab = "Y-axis",
     col = "red",
     pch = 16)
```



13.3.3 Applications

- **Understanding relationships** (e.g., linear or non-linear).
- **Identifying clusters** or grouping patterns.
- **Detecting outliers** or anomalies.

13.4 BOX-WHISKER PLOTS:

Box-whisker plots (box plots) provide a summary of the data distribution by displaying the median, quartiles, and potential outliers. Each box represents the interquartile range (IQR), while whiskers extend to the smallest and largest values within 1.5 times the IQR.

Box plots are particularly useful for:

- Comparing distributions across multiple groups.
- Identifying outliers and variability.
- Visualizing the central tendency and spread of data.

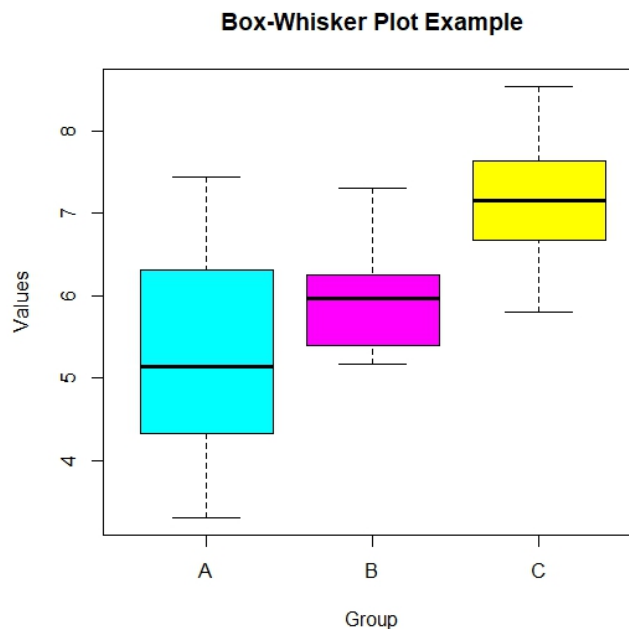
13.4.1 Syntax

```
boxplot(x, main, xlab, ylab, col, border, ...)
```

- **x**: Data vector or a formula (e.g., values ~ group).
- **main**: Title of the plot.
- **xlab/ylab**: Axis labels.
- **col**: Fill color of boxes.
- **border**: Border color.

13.4.2 R Code Example

```
# Generate grouped data
group <- rep(c("A", "B", "C"), each = 10)
values <- c(rnorm(10, 5), rnorm(10, 6), rnorm(10, 7))
# Create a boxplot
boxplot(values ~ group, main = "Box-Whisker Plot Example",
        xlab = "Group", ylab = "Values",
        col = c("cyan", "magenta", "yellow"))
```



13.4.2 Applications

- Comparing distributions between groups.
- Identifying **outliers**.
- Analyzing **variability** within and across groups.

13.5 BAR PLOTS:

Bar plots are graphical representations of categorical data using rectangular bars. The length or height of the bars corresponds to the frequency or magnitude of the category. Bar plots are ideal for:

- Visualizing categorical comparisons.
- Summarizing grouped data.
- Displaying survey results or counts.

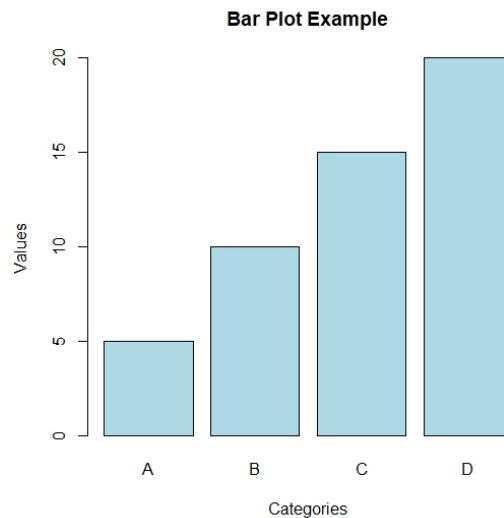
13.5.1 Syntax

```
barplot(height, names.arg, col, main, xlab, ylab, ...)
```

- **height**: Numeric vector of bar heights.
- **names.arg**: Labels for the bars.
- **col**: Bar fill color.
- **main**: Title of the plot.
- **xlab/ylab**: Axis labels.

13.5.2 R Code Example

```
# Data for bar plot
values <-c(5,10,15,20)
labels <-c("A", "B", "C", "D")
# Create a bar plot
barplot(values, names.arg = labels, col = "lightblue",
        main = "Bar Plot Example", xlab = "Categories",
        ylab = "Values")
```



13.5.3 Applications

- Visualizing **categorical comparisons**.
- Highlighting **frequencies** or counts for categories.
- **Summarizing survey responses** or grouped data.

13.6 DOT PLOTS:

Dot plots are simple graphs where each data point is represented by a dot along an axis. They are particularly useful for displaying the distribution of small datasets or comparing values across categories. Dot plots provide a clear view of individual observations.

13.6.1 Syntax

```
stripchart(x, method, main, xlab, ylab, col, ...)
```

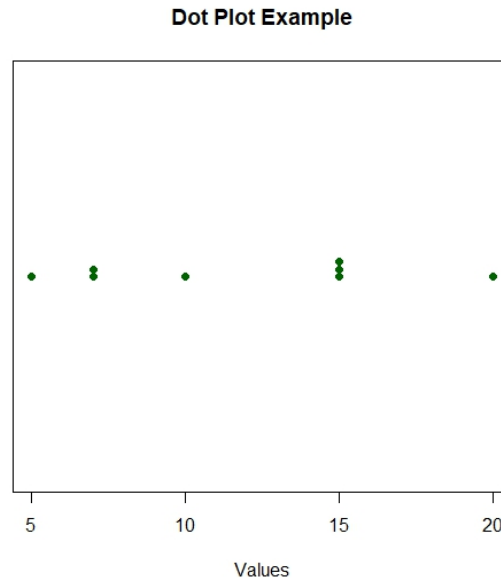
Key Parameters:

- **x**: Numeric vector of data.
- **method**: Method of plotting (e.g., "stack", "jitter", "overplot").
- **main**: Title of the plot.
- **xlab/ylab**: Axis labels.
- **col**: Color of dots.

13.6.2 R Code Example

```
# Generate data
values <-c(5,7,7,10,15,15,15,20)
```

```
# Create a dot plot
stripchart(values,
           method = "stack", main = "Dot Plot Example",
           xlab = "Values", col = "darkgreen",
           pch = 19)
```



13.6.3 Applications

- **Small Data Analysis:** Display individual observations.
- **Highlighting Clusters:** Spot repeated values.
- **Comparison:** Compare data across categories.

13.7 LINE CHARTS IN R: NUMERIC AND CATEGORICAL DATA:

Line charts are one of the most common and effective tools for visualizing data trends over time or across categories. They are particularly useful for highlighting patterns, changes, and comparisons in datasets. In R, creating line charts is straightforward and flexible, allowing for extensive customization of styles, labels, and additional elements like legends and grids. Whether you are working with purely numeric data or categorical variables, R provides functions that can handle both types seamlessly.

This guide explains the process of creating line charts for numeric and categorical data, including practical examples and syntax breakdowns. By the end, you will be able to create customized line charts suitable for your specific analysis and presentation needs.

13.8 LINE CHART WITH NUMERIC DATA:

A line chart with numeric data connects points where both x and y values are numeric.

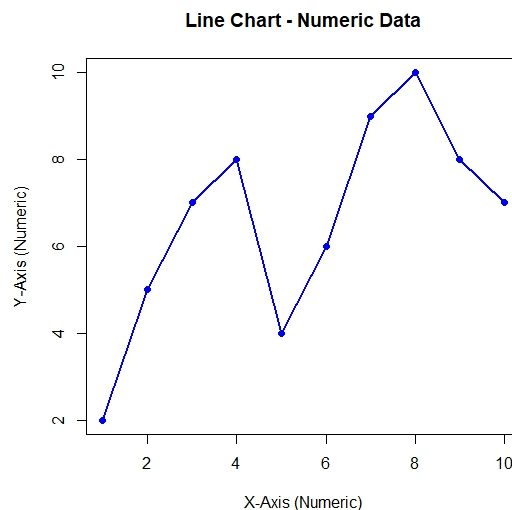
13.8.1 Syntax

```
plot(x, y, type = "o", col = "color", lwd = line_width, pch =
     point_type,
     xlab = "X-axis Label", ylab = "Y-axis Label", main =
     "Title")
```

- **x**: Numeric vector specifying x-coordinates.
- **y**: Numeric vector specifying y-coordinates.
- **type**: Defines the plot type:
 - "l": Lines only.
 - "p": Points only.
 - "o": Both points and lines.
- **col**: Line and point color.
- **lwd**: Line width.
- **pch**: Point type (e.g., 16 for filled circles).
- **xlab, ylab, main**: Labels for the axes and title.

13.8.2 Example

```
# Numeric Data Example
x <- 1:10                                # Numeric x-coordinates
y <- c(2, 5, 7, 8, 4, 6, 9, 10, 8, 7)    # Numeric y-coordinates
# Create Line Chart
plot(x, y, type = "o", col = "blue", lwd = 2, pch = 16,
     xlab = "X-Axis (Numeric)", ylab = "Y-Axis (Numeric)",
     main = "Line Chart - Numeric Data")
```



13.9 LINE CHART WITH CATEGORICAL DATA:

For categorical data, the x-axis represents categories, and the y-axis shows numeric values. Categories must be handled as factors or displayed using custom axis labels.

13.9.1 Syntax

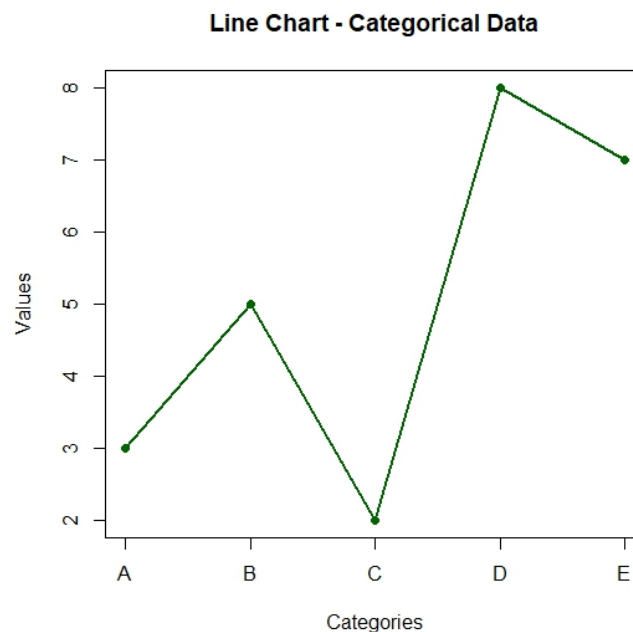
```
plot(values, type = "o", col = "color", lwd = line_width, pch
     = point_type,
     xaxt = "n", xlab = "X-axis Label", ylab = "Y-axis Label",
     main = "Title")
axis(1, at = 1:length(categories), labels = categories)
```

- **values**: Numeric y-values corresponding to categories.
- **xaxt = "n"**: Suppresses default x-axis.
- **axis()**: Customizes the x-axis:

- 1: Bottom axis.
- at: Positions for axis labels.
- labels: Text for axis labels.

13.9.2 Example

```
# Categorical Data Example
Categories<-c("A","B","C","D","E") # Categorical x-coordinates
values <- c(3, 5, 2, 8, 7) # Numeric y-coordinates
# Create Line Chart
plot(values, type = "o", col = "darkgreen", lwd = 2, pch = 16,
      xaxt = "n", xlab = "Categories", ylab = "Values", main =
"Line Chart - Categorical Data")
# Add Category Labels
axis(1, at = 1:length(categories), labels = categories)
```



13.10 COMBINED LINE CHART:

You can combine multiple data series in a single line chart to compare trends.

13.10.1 Syntax

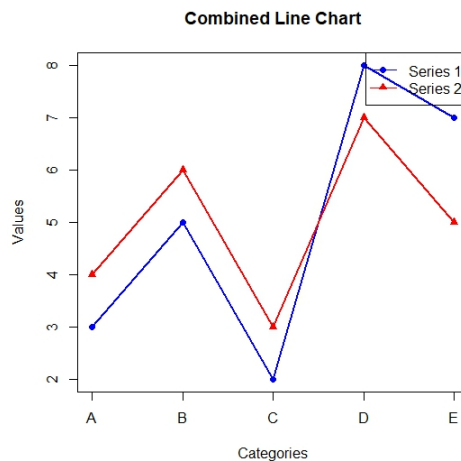
```
plot(x, y1, type = "o", col = "color1", lwd = line_width1, pch
= point_type1,xaxt = "n", xlab = "X-axis Label", ylab =
"Y-axis Label", main = "Title")
lines(x, y2, type = "o", col = "color2", lwd = line_width2,
pch = point_type2)
axis(1, at = 1:length(categories), labels = categories)
legend("position", legend = c("Label1", "Label2"), col =
c("color1", "color2"),
      pch = c(point_type1, point_type2), lty = 1)
```

13.10.2 Example

```
# Data for Combined Line Chart
categories <- c("A", "B", "C", "D", "E")
numeric_x <- 1:5
values1 <- c(3, 5, 2, 8, 7) # Numeric y-values for series 1
values2 <- c(4, 6, 3, 7, 5) # Numeric y-values for series 2
# Create Line Chart
plot(numeric_x, values1, type = "o", col = "blue", lwd = 2,
     pch = 16, xaxt = "n", xlab = "Categories", ylab = "Values",
     main = "Combined Line Chart")
lines(numeric_x, values2, type = "o", col = "red", lwd = 2,
     pch = 17)
```

```
# Add Category Labels
axis(1, at = numeric_x, labels = categories)
```

```
# Add Legend
legend("topright", legend = c("Series 1", "Series 2"), col =
c("blue", "red"),
      pch = c(16, 17), lty = 1)
```



13.11 CHARTS IN R: PIE CHARTS, BAR CHARTS, Q-Q PLOTS, AND CURVES:

This section provides a detailed guide on creating Pie Charts, Bar Charts, Q-Q Plots, and Curves in R, complete with syntax, examples, and additional tips for effective data visualization. By mastering these charts, you can visually represent data insights in a variety of contexts.

13.12 PIE CHARTS:

Pie charts are used to represent proportions or percentages of a whole. Each slice of the pie corresponds to a category's proportion.

Pie charts are a popular way to visualize proportions or percentages of a whole. They are especially useful for showing how individual parts contribute to the total. Each slice of the pie represents a specific category, and the size of the slice is proportional to its value.

13.12.1 Syntax

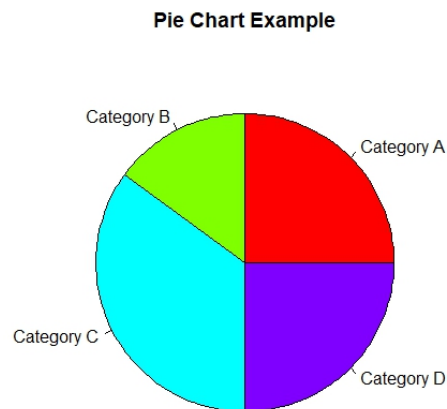
```
pie(x, labels, col, main)
```

- **x**: Numeric vector of values to be represented.
- **labels**: Character vector of category names.
- **col**: Colors for the slices.
- **main**: Title for the chart.

13.12.2 Example

```
# Example Data
values <- c(25, 15, 35, 25)
categories <- c("Category A", "Category B", "Category C",
"Category D")

# Create Pie Chart
pie(values, labels = categories, col =rainbow(length(values)),
    main = "Pie Chart Example")
```



13.13 BAR CHARTS:

Bar charts are commonly used for comparing quantities across different categories. Each bar's height represents the magnitude of the corresponding category. This chart type is ideal for showing trends or differences between groups.

13.13.1 Syntax

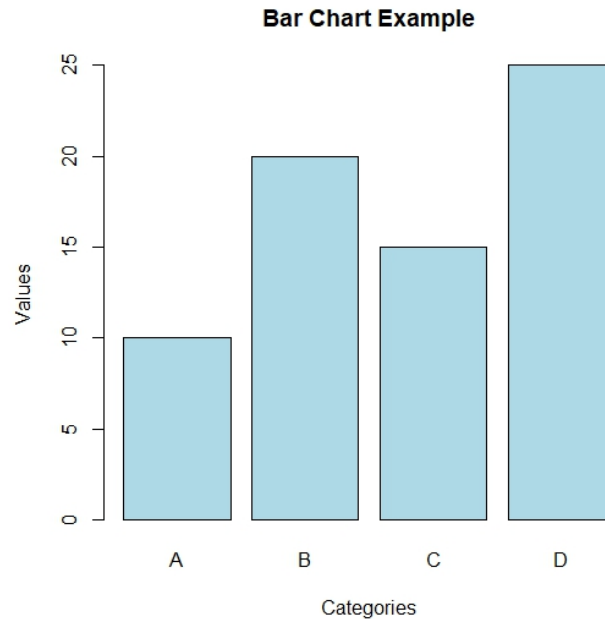
```
barplot(height, names.arg, col, xlab, ylab, main, beside)
```

- **height**: Numeric vector of bar heights.
- **names.arg**: Character vector of category labels.
- **col**: Colors for the bars.
- **xlab, ylab**: Labels for the x-axis and y-axis.
- **main**: Title for the chart.
- **beside**: Logical value; if `TRUE`, bars are side-by-side.

13.13.2 Example

```
# Example Data
values <- c(10, 20, 15, 25)
categories <- c("A", "B", "C", "D")

# Create Bar Chart
barplot(values, names.arg = categories, col = "lightblue",
        xlab = "Categories", ylab = "Values", main = "Bar
Chart Example")
```



13.14 Q-Q PLOTS:

Quantile-Quantile (Q-Q) plots are used to assess whether a dataset follows a specified theoretical distribution, such as the normal distribution. If the data points align closely with the reference line, it indicates a good fit to the distribution.

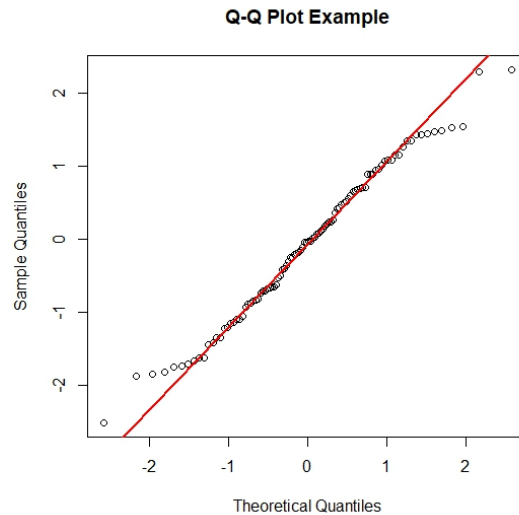
13.14.1 Syntax

```
qqnorm(y, main)
qqline(y, col, lwd)
```

- **y**: Numeric vector of data.
- **main**: Title for the plot.
- **col**: Color of the reference line.
- **lwd**: Width of the reference line.

13.14.2 Example

```
# Example Data
data <- rnorm(100) # Generate random normal data
# Create Q-Q Plot
qqnorm(data, main = "Q-Q Plot Example")
qqline(data, col = "red", lwd = 2)
```



13.15 CURVES:

Curves are used to plot mathematical functions or models over a continuous range of values. They are especially useful for visualizing relationships, trends, or behaviors of functions.

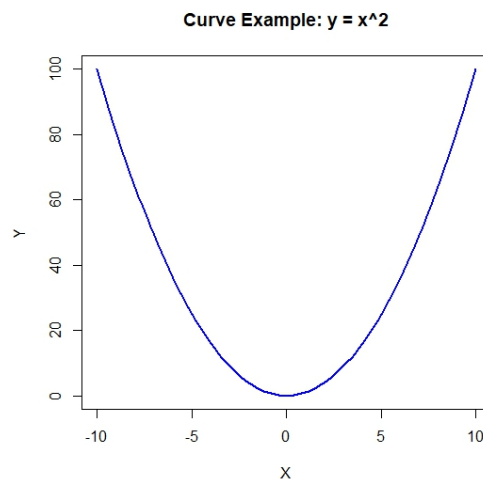
13.15.1 Syntax

```
curve(expr, from, to, col, lwd, xlab, ylab, main)
```

- **expr**: Expression defining the function.
- **from, to**: Range for the x-axis.
- **col**: Color of the curve.
- **lwd**: Line width of the curve.
- **xlab, ylab, main**: Labels and title for the plot.

13.15.2 Example

```
# Create Curve
curve(x^2, from = -10, to = 10, col = "blue", lwd = 2,
      xlab = "X", ylab = "Y", main = "Curve Example: y = x^2")
```



13.16 SUMMARY:

This lesson provides an overview of R's graphical functions for data visualization, emphasizing their role in statistical analysis. It explores high-level plotting techniques such as histograms, scatter plots, box plots, bar charts, and line graphs, which enable comprehensive data representation. Additionally, low-level functions for customization enhance the interpretability of these visualizations. The lesson ensures a thorough understanding of R's plotting capabilities, outlining their implementation and significance in data analysis. The conceptual framework of these graphical methods is examined for better comprehension, demonstrating their effectiveness in representing various data distributions and relationships.

13.17 SELF ASSESSMENT QUESTIONS:

1. What is the primary purpose of high-level plotting functions in R?
2. What parameters are used in the `hist()` function to customize a histogram in R?
3. What do the `x` and `y` vectors represent in the `plot()` function for creating a scatter plot?
4. What does a box-whisker plot represent, and how can it be useful for understanding data?
5. Explain how you would modify the `barplot()` function to display bar heights based on categorical data.
6. How are dot plots different from histograms and scatter plots?
7. Describe the difference in creating line charts with numeric data versus categorical data in R.
8. What is the role of the `labels` parameter in the `pie()` function, and how does it affect the chart?
9. How do you use the `qqnorm()` and `qqline()` functions to create a Q-Q plot in R?
10. How do you create a curve in R using the `curve()` function, and what is its purpose?

13.18 SUGGESTED READINGS

1. R Graphics Cookbook by Winston Chang
2. Data Visualization with ggplot2 by Hadley Wickham
3. R for Data Science by Hadley Wickham and Garrett Grolemund
4. Interactive Data Visualization with R by Carson Sievert
5. Data Visualization with ggplot2 by Hadley Wickham
6. The Art of Data Science by Roger D. Peng and Elizabeth Matsui

Dr. S. BHANU PRAKASH

LESSON -14

R-GRAPHICS

OBJECTIVES:

- To identify how visual representation, enhance data interpretation.
- To explore and apply different graphical techniques like histograms, scatter plots, and box plots in R.
- To modify colors, labels, titles, and themes for better readability and presentation.
- To evaluate different plotting techniques based on data structure and distribution.

STRUCTURE:

14.1 Introduction

14.2 Overview of Low-Level Plotting Functions

14.3 Adding Lines

14.4 Adding Segments

14.5 Adding Points to Plots

14.6 Adding Polygons to Plots

14.7 Adding Grids to the Plotting Region

14.8 Adding Text Using text()

14.9 Adding Legends Using legend()

14.10 Adding Marginal Text Using mtext()

14.11 Modifying and Adding Axes

14.12 Putting Multiple Plots on a Single Page

14.13 Summary

14.14 Self-Assessment Questions

14.15 Suggested Readings

14.1 INTRODUCTION:

This lesson provides an in-depth look at controlling plot options using low-level plotting functions in R. These tools allow for precise customization of visualizations, enabling users to add lines, segments, points, polygons, grids, text, legends, and axes to existing plots. Additionally, techniques for creating multiple plots on a single page are discussed.

14.2 OVERVIEW OF LOW-LEVEL PLOTTING FUNCTIONS:

Low-level plotting functions are used to add or modify elements in an existing plot. Unlike high-level functions that create new plots, low-level functions enhance the current plotting region.

Common Low-Level Functions

- **lines()**: Adds connected line segments.
- **segments()**: Draws line segments between pairs of points.
- **points()**: Adds individual points.
- **polygon()**: Adds a filled polygon.
- **grid()**: Adds a customizable grid.
- **text()**: Adds text annotations.
- **legend()**: Adds a legend to the plot.
- **mtext()**: Adds text in the margins of the plot.
- **axis()**: Customizes or adds axes to the plot.

14.3 ADDING LINES:

The `lines()` function in R is a versatile low-level plotting tool that allows you to add connected line segments to an existing plot. This is particularly useful for enhancing visualizations, such as overlaying trends, fitting curves, or marking thresholds.

14.3.1 Syntax

`lines(x, y, col,lwd,lty, type)`

Arguments

- **x, y**: Numeric vectors specifying the coordinates for the line segments.
- **col**: The color of the line. This can be specified using names (e.g., "red"), hexadecimal codes (e.g., "#FF0000"), or functions like `rainbow()`.
- **lwd**: Line width, with the default being 1. Larger values produce thicker lines.
- **lty**: Line type, specified as an integer or string. Common types include:
 - "solid" (default)
 - "dashed"
 - "dotted"
- **type**: The type of line to be drawn. Use "l" for lines or "b" for both lines and points.

14.3.2 :R Code Example(Adding a Simple Line)

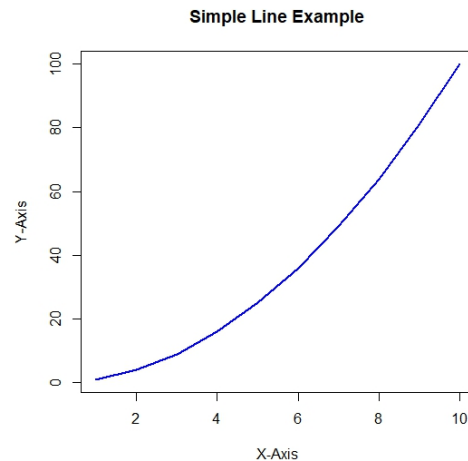
```
x <- 1:10
```

```
y <- x^2
```

```
plot(x, y, type="n", main="Simple Line Example",xlab="X-Axis",ylab="Y-Axis")
```

```
lines(x, y, col="blue",lwd=2,lty="solid")
```

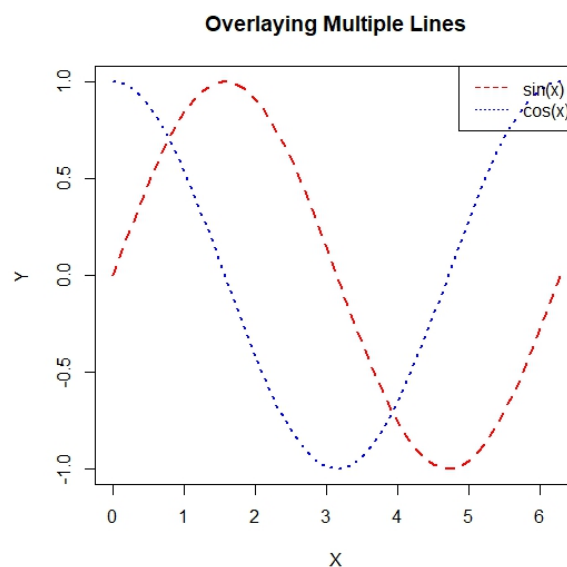
This example generates a plot with a simple blue line representing the quadratic relationship.



14.3.3: R Code Example(Overlaying Multiple Lines)

```
x <- seq(0, 2*pi, length.out=100)
y1 <- sin(x)
y2 <- cos(x)
plot(x, y1, type="n", main="Overlaying Multiple Lines", xlab="X", ylab="Y")
lines(x, y1, col="red", lwd=2, lty="dashed")
lines(x, y2, col="blue", lwd=2, lty="dotted")
legend("topright", legend=c("sin(x)", "cos(x)"), col
=c("red", "blue"), lty=c("dashed", "dotted"))
```

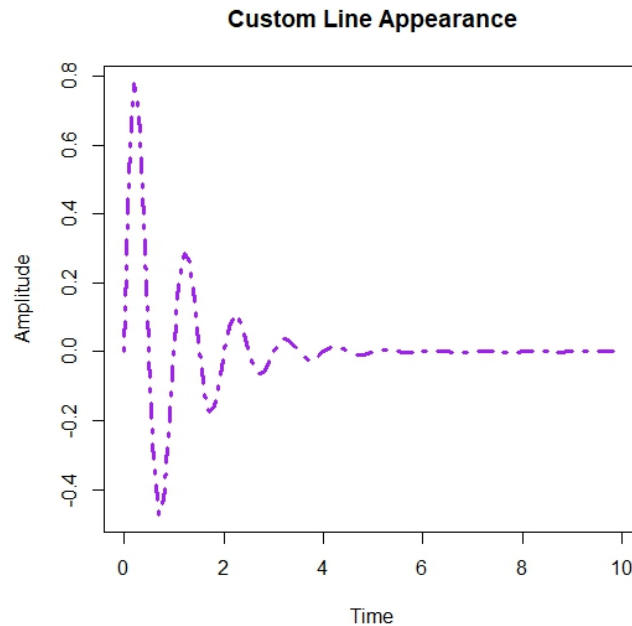
This demonstrates how to overlay two functions, $\sin(x)$ and $\cos(x)$, with distinct colors and line types for clear differentiation.



14.3.4: R Code Example(Customizing Line Appearance)

```
x <- seq(0, 10, by=0.1)
y <- exp(-x)*sin(2*pi*x)
plot(x, y, type="n", main="Custom Line Appearance", xlab="Time", ylab="Amplitude")
lines(x, y, col="purple", lwd=3, lty="dotdash")
```

This example highlights advanced customization using `lwd` for thicker lines and `lty` for a mixed line pattern.



14.4 ADDING SEGMENTS:

In data visualization, it is often necessary to emphasize specific parts of a plot or connect points to highlight relationships. The `segments()` function in R is a powerful tool that allows you to add straight-line segments to an existing plot. These segments can represent intervals, thresholds, or transitions, making your visualizations more informative and tailored to your analytical goals.

The flexibility of `segments()` allows users to precisely control the appearance and placement of each segment, making it ideal for annotating plots or illustrating trends. Whether you're highlighting data ranges, connecting observations, or adding contextual markers, the `segments()` function ensures clarity and precision in your graphical output.

14.4.1 Syntax

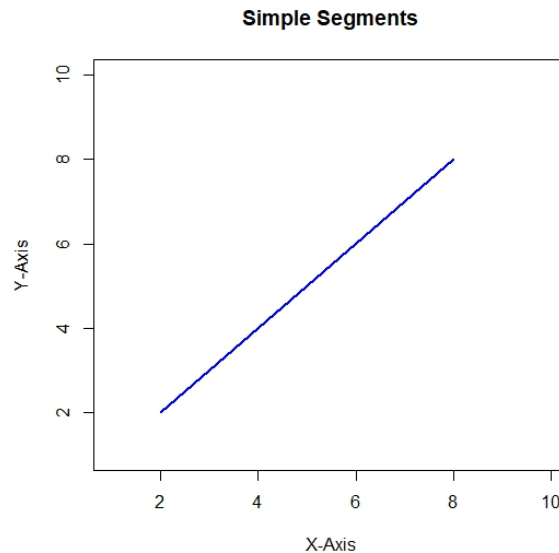
`segments(x0, y0, x1, y1, col, lwd, lty)`

Arguments

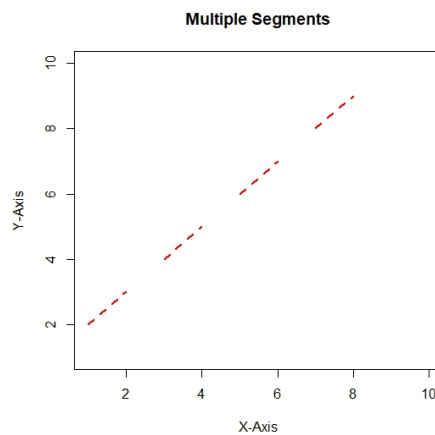
- **x0, y0:** Numeric vectors specifying the starting coordinates of the line segments.
- **x1, y1:** Numeric vectors specifying the ending coordinates of the line segments.
- **col:** The color of the segments. Colors can be defined using names (e.g., "red"), hexadecimal codes (e.g., "#FF0000"), or functions like `rainbow()`.
- **lwd:** Line width. The default value is 1, with larger values creating thicker segments.
- **lty:** Line type, such as:
 - "solid" (default)
 - "dashed"
 - "dotted"
 - "dotdash"

14.4.2: R Code Example(Simple Segments)

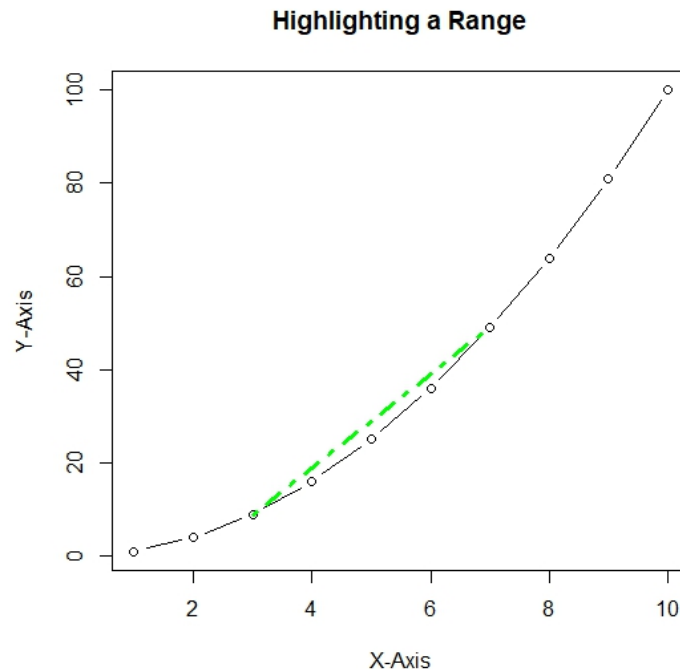
```
plot(1:10,1:10, type ="n", main ="Simple Segments",xlab="X-Axis",ylab="Y-Axis")
segments(x0 =2, y0 =2, x1 =8, y1 =8, col ="blue",lwd=2,lty="solid")
```

**14.4.3: R Code Example(Multiple Segments)**

```
x_start<-c(1,3,5,7)
y_start<-c(2,4,6,8)
x_end<-c(2,4,6,8)
y_end<-c(3,5,7,9)
plot(1:10,1:10, type ="n", main ="Multiple Segments",xlab="X-Axis",ylab="Y-Axis")
segments(x0 =x_start, y0 =y_start, x1 =x_end, y1 =y_end, col ="red",lwd=2,lty="dashed")
```

**14.4.4: R Code Example(Highlighting a Range)**

```
x <-seq(1,10, by =1)
y <- x^2
plot(x, y, type ="b", main ="Highlighting a Range",xlab="X-Axis",ylab="Y-Axis")
segments(x0 =3, y0 =9, x1 =7, y1 =49, col ="green",lwd=3,lty="dotdash")
```



14.5 ADDING POINTS TO PLOTS:

Points play a fundamental role in data visualization, serving as the building blocks of scatterplots, line graphs, and other graphical representations. The `points()` function in R is a low-level plotting function that allows you to add individual or multiple points to an existing plot. This makes it a versatile tool for customizing visualizations, such as overlaying additional data, marking specific observations, or enhancing plot clarity.

By leveraging the extensive customization options available in the `points()` function, such as controlling the size, color, and shape of points, you can create precise and informative plots tailored to your needs.

14.5.1 Syntax

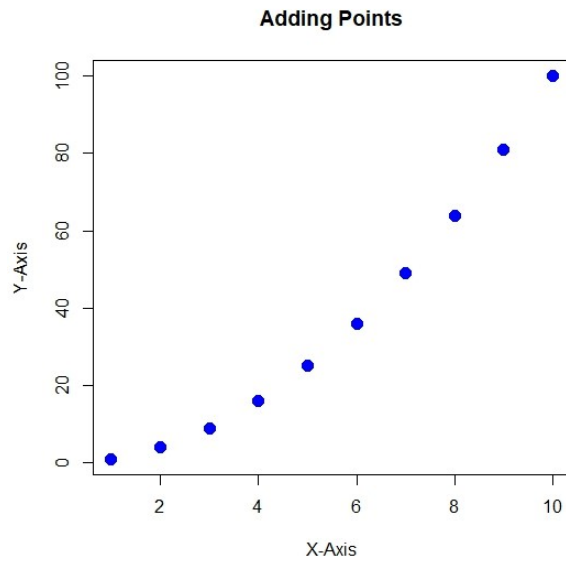
`points(x, y, col, pch, cex)`

Arguments

- **x, y:** Coordinates of the points to be added.
- **col:** Point color. Accepts color names (e.g., "red") or codes (e.g., "#FF0000").
- **pch:** Point character or symbol type. Common options include:
 - 16: Solid circle (default).
 - 1: Hollow circle.
 - 2: Triangle.
 - 3: Plus symbol.
- **cex:** Scaling factor for the size of points. The default value is 1.

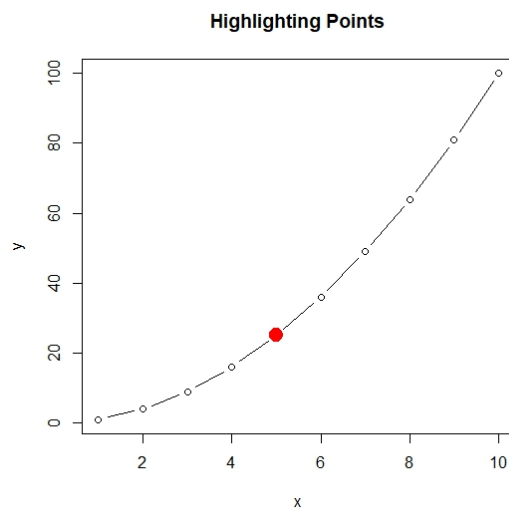
14.5.2: R Code Example(Adding Simple Points)

```
plot(1:10,(1:10)^2, type="n", main="Adding Points",xlab="X-Axis",ylab="Y-Axis")
points(x=1:10, y=(1:10)^2, col="blue",pch=16,cex=1.5)
```



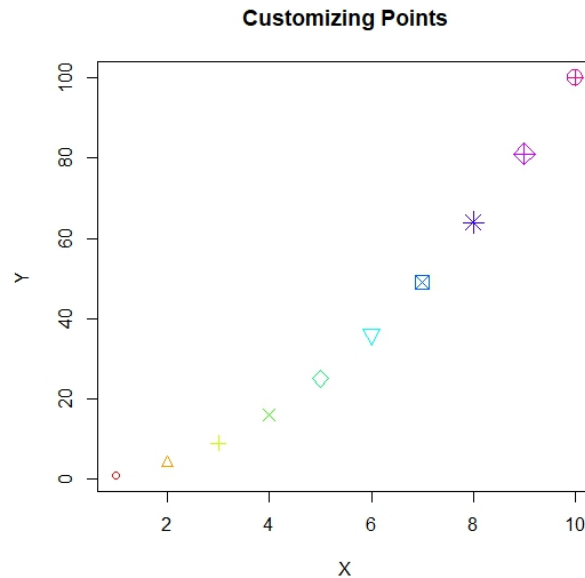
14.5.3: R Code Example(Highlighting Specific Points)

```
x <- 1:10
y <- x^2
plot(x, y, type = "b", main = "Highlighting Points")
points(5, 25, col = "red", pch = 19, cex = 2) # Highlight (5, 25)
```



14.5.4: R Code Example(Customizing Multiple Points)

```
x <- 1:10
y <- x^2
plot(x, y, type = "n", main = "Customizing Points", xlab = "X", ylab = "Y")
points(x, y, col = rainbow(10), pch = 1:10, cex = seq(1, 2, length.out = 10))
```

14.6 ADDING POLYGONS TO PLOTS:

Polygons are versatile graphical elements used to represent areas, boundaries, or regions on a plot. In R, the `polygon()` function allows you to draw filled shapes defined by a series of vertices. This makes it an essential tool for highlighting specific regions, creating shaded areas, or representing geometric shapes in your visualizations.

The `polygon()` function is particularly useful in creating custom visualizations, such as:

- Shading areas under curves or between lines.
- Highlighting specific data ranges or regions.
- Adding geometric shapes for visual emphasis.

By combining polygons with other low-level functions, you can enhance the interpretability and aesthetic appeal of your plots.

14.6.1 Syntax

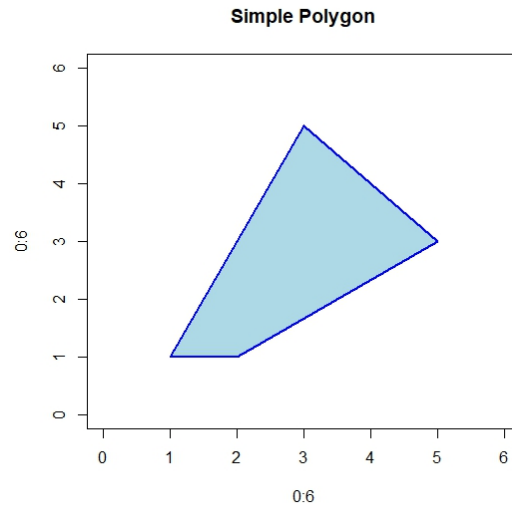
`polygon(x, y, col, border, lty, lwd, density, angle)`

Arguments

- **x, y**: Coordinates of the vertices defining the polygon.
- **col**: Fill color of the polygon.
- **border**: Color of the polygon's border (use NA for no border).
- **lty**: Line type for the border (e.g., solid, dashed).
- **lwd**: Line width for the border.
- **density**: Line density for shading (in number of lines per inch).
- **angle**: Angle of the shading lines.

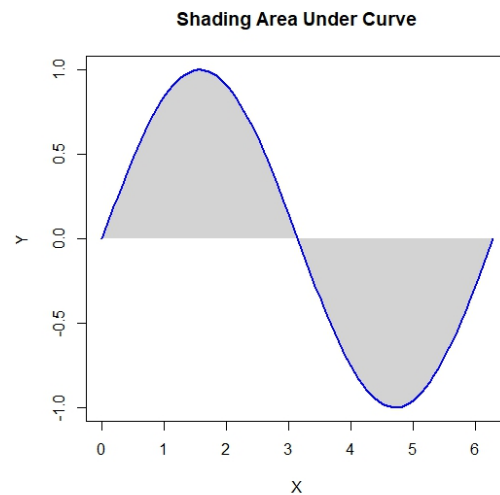
14.6.2: R Code Example(Simple Polygon)

```
x <-c(1,3,5,2)
y <-c(1,5,3,1)
plot(0:6,0:6, type ="n", main ="Simple Polygon")
polygon(x, y, col ="lightblue", border ="blue",lwd=2)
```



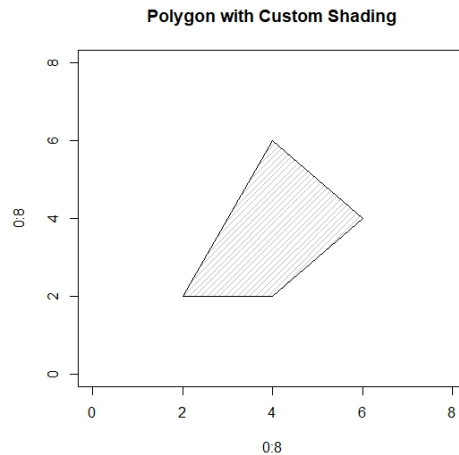
14.6.3: R Code Example(Shading an Area Under a Curve)

```
x <- seq(0, 2*pi, length.out=100)
y <- sin(x)
plot(x, y, type="l", main="Shading Area Under Curve", xlab="X", ylab="Y")
polygon(c(x, rev(x)), c(y, rep(0, length(y))), col="lightgray", border=NA)
lines(x, y, col="blue", lwd=2)
```



14.6.4: R Code Example(Custom Shading)

```
x <- c(2, 4, 6, 4)
y <- c(2, 6, 4, 2)
plot(0:8, 0:8, type="n", main="Polygon with Custom Shading")
polygon(x, y, col="gray", border="black", density=20, angle=45)
```



14.7. ADDING GRIDS TO THE PLOTTING REGION:

Grids are a helpful visual element for organizing and aligning plot elements. The `grid()` function in R allows you to add a customizable grid to the plotting region, enhancing the interpretability of data points or patterns. Grids can be used in scatterplots, line graphs, bar charts, and other visualizations where alignment aids in data comparison.

14.7.1 Syntax

`grid(nx,ny, col,lty,lwd)`

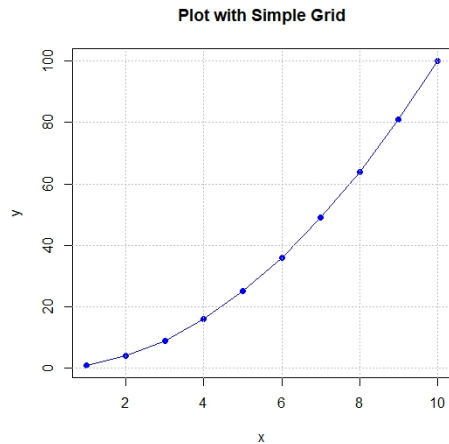
Arguments

- **nx**: Number of vertical grid lines. Defaults to NULL, which aligns with major x-axis ticks.
- **ny**: Number of horizontal grid lines. Defaults to NULL, which aligns with major y-axis ticks.
- **col**: Color of the grid lines (default is light gray).
- **lty**: Line type (e.g., "solid", "dashed", "dotted").
- **lwd**: Line width.

If `nx` or `ny` is NULL, the grid aligns with the existing axis tick marks.

14.7.2: R Code Example(Simple Grid Aligned with Axes)

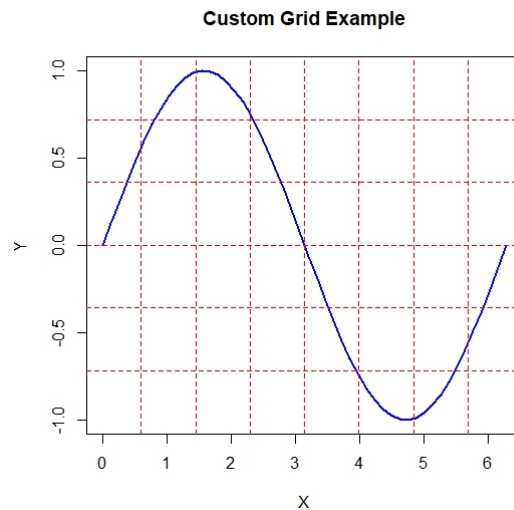
```
x <- 1:10
y <- x^2
plot(x, y, type = "o", main = "Plot with Simple Grid", col = "blue", pch = 16)
grid(col = "gray", lty = "dotted")
```



This adds a dotted gray grid to the plotting region, aligning with the tick marks on the axes.

14.7.3: R Code Example (Custom Grid with Specified Lines)

```
x <- seq(0, 2*pi, length.out=50)
y <- sin(x)
plot(x, y, type="l", col="blue", lwd=2, main="Custom Grid Example", xlab="X", ylab="Y")
grid(nx=8, ny=6, col="red", lty="dashed", lwd=0.5)
```



This example demonstrates a custom grid with 8 vertical and 6 horizontal lines, styled with red dashed lines.

14.8 ADDING TEXT USING text():

The `text()` function places text at specified coordinates within the plotting region. Text annotations can highlight key data points, provide additional context, or label specific regions of a plot. This is often used for labeling specific points or adding descriptions, ensuring the plot conveys a clear message.

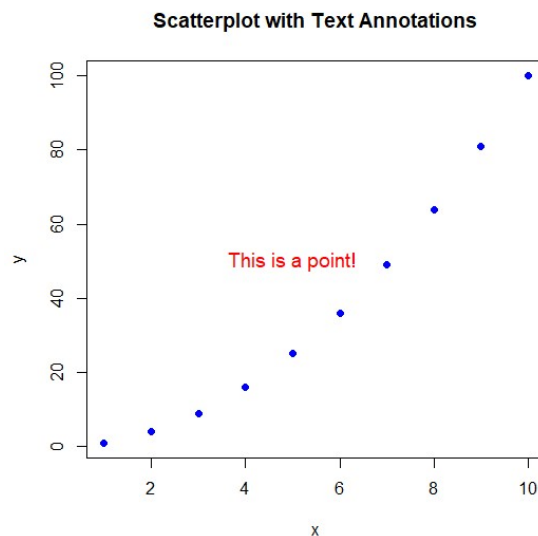
14.8.1 Syntax

`text(x, y, labels, col, cex)`

- **x, y**: Coordinates for placing the text.
- **labels**: Text to display.
- **col**: Text color.
- **cex**: Text size.

14.8.2: R Code Example(Annotating a Scatterplot)

```
x <- 1:10
y <- x^2
plot(x, y, main = "Scatterplot with Text Annotations", col = "blue", pch = 16)
text(5, 50, "This is a point!", col = "red", cex = 1.2)
```



14.9 ADDING LEGENDS USING legend():

Legends play a crucial role in explaining the meaning of symbols, colors, or line styles used in a plot. The `legend()` function creates a descriptive box that enhances the interpretability of the visualization. Properly labeled legends make complex plots accessible to the audience.

14.9.1 Syntax

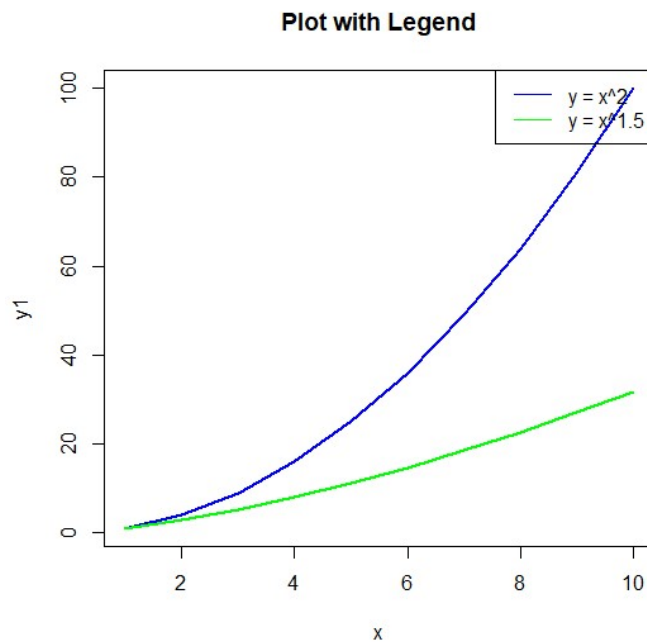
`legend(x, y, legend, col, pch, lty, bty)`

- **x, y**: Coordinates or position of the legend (e.g., "topright", "bottomleft").
- **legend**: Vector of labels for the legend.
- **col, pch, lty**: Colors, point types, and line styles matching the plot.
- **bty**: Box type (e.g., "o" for a box, "n" for none).

14.9.2: R Code Example(Adding a Legend to a Line Plot)

```
x <- 1:10
y1 <- x^2
y2 <- x^1.5
```

```
plot(x, y1, type = "l", col = "blue", lwd = 2, main = "Plot with Legend")
lines(x, y2, col = "green", lwd = 2)
legend("topright", legend = c("y = x^2", "y = x^1.5"), col = c("blue", "green"), lty = 1)
```



14.10 ADDING MARGINAL TEXT USING `mtext()`:

The `mtext()` function is used to place text in the margins of a plot. This feature is particularly useful for adding axis titles, supplementary information, or labels that do not fit within the main plotting region. Marginal text adds contextual information, enhancing the overall presentation.

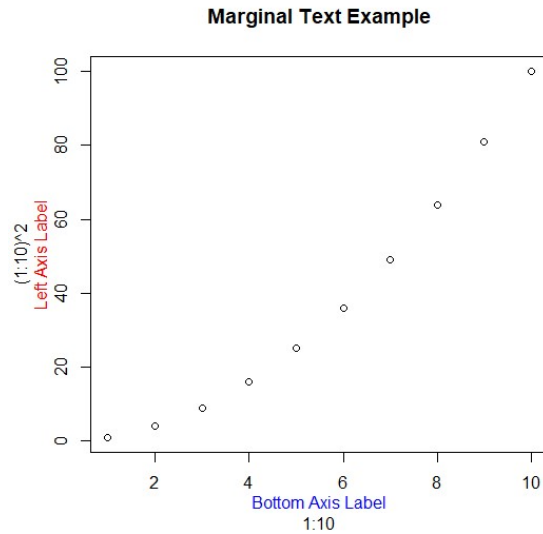
14.10.1 Syntax

```
mtext(text, side, line, col)
```

- **text**: Text to display.
- **side**: 1 = bottom, 2 = left, 3 = top, 4 = right.
- **line**: Line number in the margin.
- **col**: Text colour.

14.10.2: R Code Example (Adding Axis Labels with `mtext()`)

```
plot(1:10, (1:10)^2, main = "Marginal Text Example")
mtext("Bottom Axis Label", side = 1, line = 2, col = "blue")
mtext("Left Axis Label", side = 2, line = 2, col = "red")
```



14.11 MODIFYING AND ADDING AXES:

Axes are essential components of a plot as they provide the framework for interpreting the data. The `axis()` function allows users to customize or add new axes, improving the clarity and precision of the visualization. By modifying tick marks, labels, and axis styles, you can make your plot more informative.

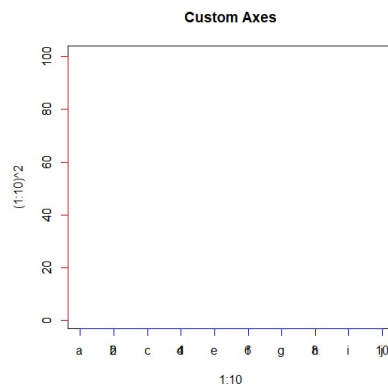
14.11.1 Syntax

`axis(side, at, labels, col, lwd)`

- **side:** 1 = bottom, 2 = left, 3 = top, 4 = right.
- **at:** Locations of tick marks.
- **labels:** Labels for the tick marks.
- **col, lwd:** Colors and widths of the axis lines.

14.11.1:R Code Example(Custom Axes)

```
plot(1:10, (1:10)^2, type = "n", main = "Custom Axes")
axis(1, at = 1:10, labels = letters[1:10], col = "blue")
axis(2, at = seq(0, 100, 20), col = "red")
```



14.12 PUTTING MULTIPLE PLOTS ON A SINGLE PAGE:

Arranging multiple plots on a single page is a powerful feature for comparative analysis and presentation. The `par()` function provides the capability to divide the plotting region into multiple sections, enabling the creation of grid-like layouts for multiple visualizations.

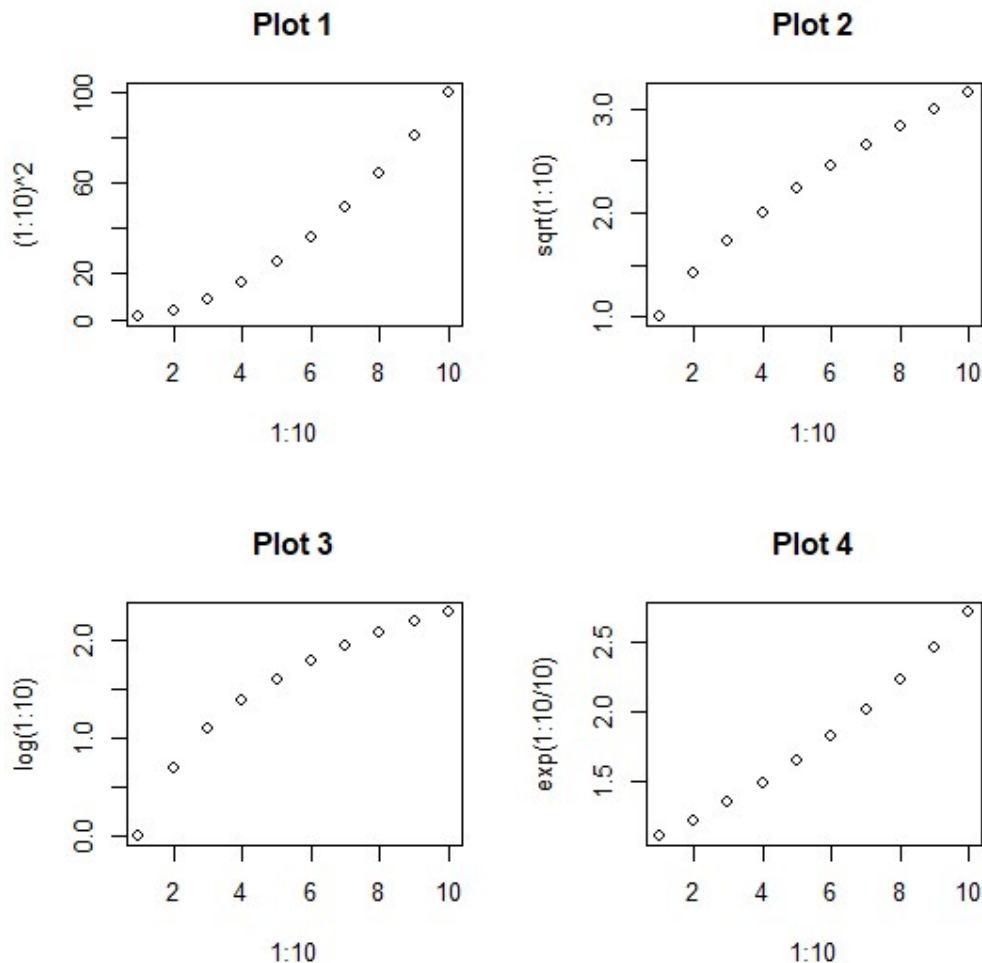
14.12.1 Syntax

```
par(mfrow = c(nrows, ncols))
```

- **nrows, ncols:** Number of rows and columns for the layout.

14.12.2: R Code Example(Four Plots on a Single Page)

```
par(mfrow = c(2, 2)) # Divide into 2 rows and 2 columns
plot(1:10, (1:10)^2, main = "Plot 1")
plot(1:10, sqrt(1:10), main = "Plot 2")
plot(1:10, log(1:10), main = "Plot 3")
plot(1:10, exp(1:10/10), main = "Plot 4")
par(mfrow = c(1, 1)) # Reset layout to single plot
```



14.13 SUMMARY:

Data visualization in R is essential for understanding and interpreting data effectively. It enables users to explore and apply various graphical techniques such as histograms, scatter plots, and box plots to represent data visually. Customization options like colors, labels, and titles enhance the clarity and aesthetics of these visualizations. Additionally, comparing different plotting methods helps in selecting the most suitable approach for various data types, ensuring accurate and meaningful insights. Mastering these techniques allows for better data-driven decision-making and improved communication of statistical findings.

14.14 SELF ASSESSMENT QUESTIONS:

1. What is the purpose of low-level plotting functions in R, and how do they differ from high-level plotting functions?
2. Describe the functionality of the `lines()` function in R. What arguments can be used to customize the appearance of the lines? Provide an example where you overlay multiple lines on the same plot.
3. How does the `segments()` function enhance a plot? What are the key arguments for drawing line segments in R, and how can you modify their appearance?
4. Explain the role of the `points()` function in R. How can you customize the appearance of individual points in a plot? Provide an example where you highlight specific points.
5. What is the purpose of the `polygon()` function, and how can it be used to shade areas or highlight regions in a plot? Provide an example of creating a polygon and adding shading under a curve.
6. How can you add a customizable grid to a plot using the `grid()` function? Describe the arguments involved in modifying the grid's appearance.
7. What are the use cases for adding text annotations in plots using the `text()` function? How do you specify the position and appearance of the text? Provide an example where text is added to annotate a plot.
8. How does the `par()` function in R allow users to display multiple plots on a single page? Write an example that arranges four plots in a 2x2 layout. What is the primary purpose of high-level plotting functions in R?

14.15 SUGGESTED READINGS:

1. R Graphics Cookbook by Winston Chang
2. The Art of R Programming by Norman Matloff
3. R for Data Science by Hadley Wickham and Garrett Grolemund
4. R Graphics by Paul Murrell
5. R in Action by Robert I. Kabacoff
6. The Art of Data Science by Roger D. Peng and Elizabeth Matsui

Dr. S. BHANU PRAKASH

LESSON -15

R-GRAPHICS

OBJECTIVES:

1. To understand the Concept of ANOVA.
2. To identify and Verify ANOVA Assumptions.
3. To perform One-Way ANOVA in R.
4. To perform Two-Way ANOVA in R.

STRUCTURE:

15.1 Introduction

15.2 Assumptions of ANOVA

15.3 One-Way ANOVA

15.3.1 Syntax

15.3.2 R Code Example

15.4 One-Way ANOVA

15.4.1 Syntax

15.4.2 R Code Example

15.5 Summary

15.6 Self-Assessment Questions

15.7 Suggested Readings

15.1 INTRODUCTION:

This lesson provides an in-depth overview of performing one-way and two-way ANOVA using R's built-in functions. ANOVA (Analysis of Variance) is a powerful statistical technique used to compare means across multiple groups and assess whether the observed differences are statistically significant. By analyzing the variability within and between groups, ANOVA allows researchers to draw meaningful conclusions about the factors influencing their data. R provides robust tools for conducting ANOVA, and this guide covers the necessary syntaxes, examples, and interpretations.

15.2 ASSUMPTIONS OF ANOVA:

- **Normality:** Residuals should be normally distributed.
 - **Homogeneity of Variance:** Variances should be equal across groups.
 - **Independence:** Observations should be independent.
- Ensure proper experimental design to satisfy this assumption.

15.3 ONE-WAY ANOVA:

One-way ANOVA is used to determine whether there are statistically significant differences between the means of three or more independent groups. It evaluates the impact of a single factor (categorical variable) on a continuous response variable. This method is widely used in experiments and studies where groups are formed based on different treatments or conditions.

15.3.1 Syntax

```
result <- aov(response_variable ~ factor_variable, data = dataset)
summary(result)
```

- **response_variable**: The numeric dependent variable.
- **factor_variable**: The independent categorical variable (factor).
- **dataset**: The data frame containing the variables.

15.3.2 R Code Example

```
# Sample data
data <- data.frame(
  treatment = rep(c("A", "B", "C"), each = 10),
  value = c(rnorm(10, mean = 20, sd = 3),
    rnorm(10, mean = 25, sd = 3),
    rnorm(10, mean = 30, sd = 3))
)
```

```
# Perform one-way ANOVA
result <- aov(value ~ treatment, data = data)
```

```
# Summary of ANOVA
summary(result)
```

Output:

```
      Df Sum Sq Mean Sq F value    Pr(>F)
treatment  2  660.0   330.0   35.7 2.62e-08 ***
Residuals 27  249.6     9.2
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Interpretation:

The summary output includes:

- **Degrees of Freedom (Df)**: Number of independent values in the calculation.
- **Sum of Squares (SS)**: Measures variability.
- **Mean Squares (MS)**: SS divided by Df.
- **F-value**: Ratio of MS between groups to MS within groups.
- **p-value**: Indicates if group means differ significantly.

A p-value less than 0.05 suggests significant differences between groups, implying that at least one group mean is different from the others.

15.4 TWO-WAY ANOVA

Two-way ANOVA is used when there are two independent categorical variables (factors) and one continuous dependent variable. It evaluates:

1. The individual effect of each factor (main effects).
2. The interaction effect between the two factors, showing whether the effect of one factor depends on the levels of the other factor.

This method is particularly useful in experiments where multiple factors are varied simultaneously, allowing researchers to explore combined and independent influences on the response variable.

15.4.1 Syntax

```
result <- aov(response_variable ~ factor1 * factor2, data = dataset)
summary(result)
```

- **factor1 and factor2:** The two independent categorical variables.
- **factor1 * factor2:** Includes main effects and interaction effects.
- **dataset:** The data frame containing the variables.

15.4.2 R Code Example

```
# Sample data
data <- data.frame(
  fertilizer = rep(c("Low", "Medium", "High"), each = 10),
  irrigation = rep(c("Low", "High"), times = 15),
  yield = c(rnorm(10, mean = 30, sd = 5),
            rnorm(10, mean = 35, sd = 5),
            rnorm(10, mean = 40, sd = 5),
            rnorm(10, mean = 50, sd = 5),
            rnorm(10, mean = 55, sd = 5),
            rnorm(10, mean = 60, sd = 5))
)
# Perform two-way ANOVA
result <- aov(yield ~ fertilizer * irrigation, data = data)

# Summary of ANOVA
summary(result)
```

Output:

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
fertilizer	2	1030	514.8	3.604	0.034 *
irrigation	1	0	0.1	0.000	0.984
fertilizer:irrigation	2	9	4.7	0.033	0.968
Residuals	54	7714	142.9		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Interpretation:

The output provides:

1. **Main Effects:** The effect of fertilizer and irrigation individually.
2. **Interaction Effect:** Combined effect of fertilizer and irrigation (fertilizer:irrigation).
3. **p-values:** Significance of main and interaction effects.

A significant interaction effect ($p < 0.05$) suggests that the effect of one factor depends on the levels of the other factor. If no interaction is present, the main effects can be interpreted independently.

15.5 SUMMARY:

This lesson provides a comprehensive guide to performing one-way and two-way ANOVA in R, covering their assumptions, syntax, implementation, and interpretation. Learners will understand the importance of ANOVA in comparing group means, ensuring assumptions like normality and homogeneity of variance are met, and analyzing variability within and between groups. Using R's `aov()` function, they will conduct ANOVA, interpret key outputs like F-values and p-values, and assess statistical significance. Additionally, they will explore main and interaction effects in two-way ANOVA, applying these techniques to real-world scenarios for data-driven decision-making.

15.6 SELF ASSESSMENT QUESTIONS:

1. What are the key assumptions of ANOVA, and why are they important?
2. How does one-way ANOVA differ from two-way ANOVA?
3. How do you interpret the output of `summary(aov(response_variable ~ factor_variable, data = dataset))` in R?
4. Write the R syntax for conducting a two-way ANOVA with two categorical independent variables and one continuous dependent variable.
5. A researcher performs an ANOVA test and finds a p-value of 0.03. What does this mean in terms of statistical significance?
6. How can ANOVA results be used for decision-making in real-world scenarios?
7. What post-hoc tests can be used after finding a significant result in ANOVA?

15.7 SUGGESTED READINGS:

1. Discovering Statistics Using R by Andy Field
2. Linear Models with R by Julian J. Faraway
3. Introductory Statistics with R by Peter Dalgaard
4. The R Book by Michael J. Crawley
5. Hands-On Programming with R by Garrett Grolmund
6. Designing Experiments and Analyzing Data: A Model Comparison Perspective by Maxwell & Delaney

Dr. S. BHANU PRAKASH