SHRIMATI INDIRA GANDHI COLLEGE

Affiliated to Bharathidasan University| Nationally Accredited at 'A' Grade(3rd Cycle) by NAAC

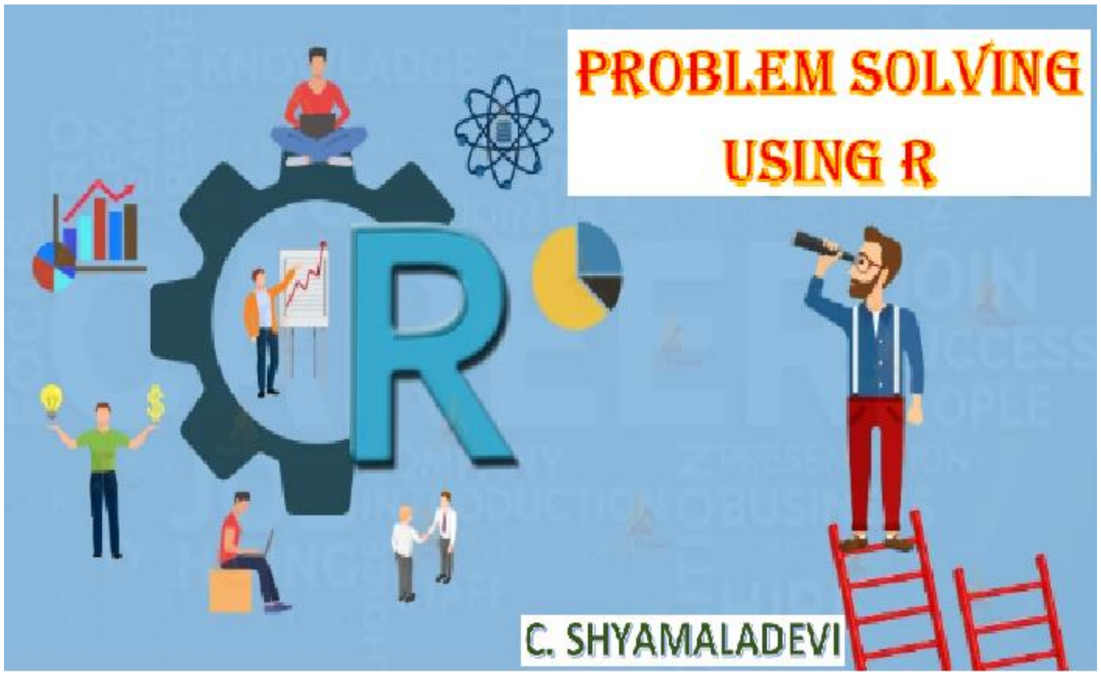
An ISO 9001:2015 Certified Institution

Thiruchirrappalli

STUDY MATERIAL

PROBLEM SOLVING USING R

DEPARTMENT OF COMPUTER SCIENCE, INFORMATION TECHNOLOGY AND COMPUTER APPLICATIONS



Prepared by,

MS. C. SHYAMALADEVI, M.C.A., M.Phil., B.Ed.,

ASST. PROF. IN COMPUTER SCIENCE,

SHRIMATI INDIRA GANDHI COLLEGE,

TIRUCHIRAPPALLI - 2

PROBLEM SOLVING USING R

UNIT - I OVERVIEW OF R:

History and Overview of R- Basic Features of R-Design of the R SystemInstallation of R-Console and Editor Panes- Comments- Installing and Loading R Packages- Help Files and Function Documentation- Saving Work and Exiting RConventions- R for Basic Math-Arithmetic- Logarithms and Exponentials- ENotation- Assigning Objects- Vectors- Creating a Vector- Sequences, Repetition, Sorting, and Lengths- Subsetting and Element Extraction-Vector-Oriented Behaviour Practical

UNIT - II MATRICES AND ARRAYS:

Defining a Matrix – Defining a Matrix- Filling Direction- Row and Column Bindings- Matrix Dimensions- Subsetting- Row, Column, and Diagonal Extractions- Omitting and Overwriting-Matrix Operations and Algebra- Matrix Transpose- Identity Matrix- Matrix Addition and Subtraction- Matrix Multiplication- Matrix Inversion-Multidimensional Arrays- Subsets, Extractions, and Replacements

UNIT - III NON-NUMERIC VALUES:

Logical Values- Relational Operators- Characters- Creating a StringConcatenation- Escape Sequences- Substrings and Matching- FactorsIdentifying Categories- Defining and Ordering Levels- Combining and Cutting

UNIT - IV LISTS AND DATA FRAMES:

Lists of Objects-Component Access-Naming-Nesting-Data Frames-Adding Data Columns and Combining Data Frames-Logical Record Subsets-Some Special Values-Infinity-NaN-NA-NULL Attributes-Object-Class-Is-Dot Object-Checking Functions-As-Dot Coercion Functions

UNIT – V BASIC PLOTTING:

Using plot with Coordinate Vectors-Graphical Parameters-Automatic Plot TypesTitle and Axis Labels Color-Line and Point Appearances-Plotting Region LimitsAdding Points, Lines, and Text to an Existing Plot-ggplot2 Package-Quick Plot with qplot-Setting Appearance Constants with Geoms-- READING AND WRITING FILES- R-Ready Data Sets- Contributed Data Sets-

Reading in External Data Files- Writing Out Data Files and Plots- Ad Hoc Object Read/Write Operations

UNIT – VI

CURRENT CONTOURS (For continuous internal assessment only): Contemporary Developments Related to the Course during the Semester Concerned.

REFERENCES:

1. Tilman M. Davies, "The Book of R - A First Programming and Statistics" Library of Congress Cataloging-in-Publication Data,2016.

2. Roger D. Peng,"R Programming for Data Science"Lean Publishing, 2016.

3. Hadley Wickham, Garrett Grolemund," R for Data Science",OREILLY Publication,2017

4. Steven Keller, "R Programming for Beginners", CreateSpace Independent Publishing Platform 2016.

5. Kun Ren ,"Learning R Programming", Packt Publishing,2016

6. https://www.geeksforgeeks.org/r-programming-exercises-practice-questionsand-solutions/

7. https://www.w3schools.com/r/r_graph_plot.asp

8. https://www.geeksforgeeks.org/list-of-dataframes-in-r/

*****

UNIT I - OVERVIEW OF R

History and Overview of R

R is a programming language and software environment designed for statistical computing and graphics.

Developed by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand in the 1990s.

R is an implementation of the S programming language and is open-source, which encourages collaboration and continuous development.

Widely used in academia, research, and industry for data analysis, visualization, and machine learning.

Basic Features of R

Interactive programming environment: R provides an interactive shell where commands can be executed and results are immediately displayed.

Object-oriented nature: R treats data as objects that can be manipulated and analyzed using functions and methods.

Extensive statistical and graphical capabilities: R offers a wide range of statistical and graphical techniques for data analysis and visualization.

Open-source and community-driven development: R benefits from a vibrant community of developers who contribute packages, tools, and resources to enhance its functionality.

Cross-platform compatibility: R is available for various operating systems, including Windows, macOS, and Linux.

Design of the R System

R consists of several components that work together to provide a complete programming environment.

The core component is the R interpreter, which executes R code and performs computations.

R also includes a compiler that translates R code into machine-readable instructions for improved performance.

The R runtime environment manages memory, handles function calls, and coordinates the execution of R code.

Users interact with R through the R console, where commands are entered and results are displayed. The R console supports a command-line interface for executing code interactively.

Additionally, R provides an editor pane for writing and editing R scripts, allowing users to create and save a sequence of R commands for later execution.

Installation of R

Installing R is a straightforward process. Here are the general steps:

Visit the official R website at [insert website URL].

Choose the appropriate version of R for your operating system (Windows, macOS, or Linux).

Download the R installer package.

Run the installer and follow the on-screen instructions.

Configure any installation settings, such as the installation directory and additional components.

Complete the installation process.

After successful installation, you can launch R from the Start menu (Windows) or Applications folder (macOS).

Console and Editor Panes

R provides two primary panes for interacting with the programming environment: the console pane and the editor pane.

The console pane is the primary interface for executing R commands interactively. It displays the R prompt (>) where you can enter commands and see the output.

The console pane supports both immediate execution and line-by-line execution of code.

The editor pane is used for writing and editing R scripts. It allows you to create and save a sequence of R commands for later execution.

The editor pane offers features like syntax highlighting, code indentation, and line numbering to enhance code readability.

Using the R Console

The R console allows you to execute R commands interactively and obtain immediate results.

To execute a command, simply type it at the prompt (>), and press Enter.

The console displays the output of each command immediately below the command line.

You can also recall and edit previously entered commands using the up and down arrow keys.

To clear the console and start fresh, use the Ctrl + L shortcut or the clearConsole() function.

Using the Editor Pane

The editor pane provides a convenient way to write and edit longer sequences of R commands.

To open the editor pane, go to the File menu and choose "New Script" or use the shortcut Ctrl + Shift + N (Windows) or Cmd + Shift + N (macOS).

Once the editor pane is open, you can write R code, save it to a file, and execute the entire script or selected portions of it.

To execute code from the editor pane, either select the code you want to run and click the "Run" button or use the Ctrl + Enter shortcut.

Saving and Loading Scripts

To save your R script, go to the File menu and choose "Save" or "Save As." Provide a name and choose a location to save the file.

You can load an existing R script into the editor pane by going to the File menu and selecting "Open Script." Locate the file on your computer and click "Open."

Saving your R scripts allows you to reuse and share your code, making it easier to reproduce analyses and collaborate with others.

It's good practice to organize your scripts into logical folders and use meaningful file names to easily locate and identify them.

Comments

Comments in R are used to add explanatory notes within your code for better understanding and documentation.

They are ignored by the R interpreter and have no impact on the execution of the code.

To add a single-line comment, use the # symbol followed by your comment.

For multi-line comments, you can enclose the text between /* and */.

Comments play a crucial role in making your code more readable and maintainable, especially for yourself and others who may read your code.

Installing and Loading R Packages

R packages are collections of functions, data, and documentation that extend the functionality of the base R system.

To install an R package, you can use the install.packages() function. For example: install.packages("packageName").

The function will download and install the specified package from the Comprehensive R Archive Network (CRAN) repository.

Once installed, you can load a package into your R session using the library() function. For example: library(packageName).

Loading a package makes its functions and data available for use in your R code.

Help Files and Function Documentation

R provides extensive documentation for its functions and packages, making it easier to understand their purpose and usage.

To access the help file for a specific function, you can use the ? operator followed by the function name. For example: ?functionName.

Another option is to use the help() function. For example: help(functionName).

The help file provides detailed information about the function's parameters, return values, usage examples, and related functions.

Additionally, many packages have their own vignettes, tutorials, or online documentation to guide users in utilizing their functionalities effectively.

Navigating Function Documentation

Function documentation in R typically follows a standardized structure.

The documentation includes a description of the function's purpose and a list of its parameters, along with their types and default values.

Usage examples are provided to illustrate how to use the function correctly.

Additional sections may cover details such as the function's source code, related functions, and references to relevant literature.

Reading and understanding function documentation is crucial for using functions effectively and ensuring proper input and output handling.

Using Package Vignettes

Many R packages include vignettes, which are long-form documentation and tutorials.

Vignettes provide comprehensive explanations of package functionalities and demonstrate their application with examples.

To access a package's vignettes, use the browseVignettes() function, specifying the package name. For example:

browseVignettes("packageName").

This function opens a web browser window where you can navigate through the available vignettes and explore the package's capabilities in detail.

Community Resources and Forums

Apart from official documentation, there are various community-driven resources and forums available to seek help and interact with other R users.

Websites like Stack Overflow, R-bloggers, and the RStudio Community provide platforms for asking questions, sharing insights, and discussing R-related topics.

Participating in these communities can help you find solutions to specific problems, gain insights into best practices, and connect with experts in the R ecosystem.

When seeking assistance, it's helpful to provide a minimal reproducible example (reprex) to clearly describe the problem and facilitate a quicker response from the community.

Saving Work and Exiting R

Saving your work in R is important to preserve your objects, data, and analysis for future use.

To save your entire R workspace, including all objects and their values, you can use the save.image() function. For example: save.image("path/to/file.RData").

The saved workspace file has the extension .RData and can be loaded later to restore your environment.

You can also save specific objects using the save() function. For example: save(object1, object2, file = "path/to/file.RData").

To load a saved workspace or object file, use the load() function. For example: load("path/to/file.RData").

To exit the R session, you can use the q() or quit() functions. R will prompt you to save your workspace before exiting.

Conventions

Following coding conventions is essential for writing clean, readable, and maintainable R code.

Consistent and clear code makes it easier for yourself and others to understand and work with your code.

Some common conventions in R programming include:

Using lowercase letters for variable and function names (e.g., my_variable, calculate_mean).

Using meaningful names that describe the purpose of variables and functions.

Avoiding abbreviations and single-letter variable names, except in special cases.

Indenting code using consistent spaces or tabs for better readability.

Adding spaces around operators for clarity (e.g., x <- 10 + 2).

Using comments to provide explanations, clarify intentions, or disable code temporarily.

Code Formatting and Readability

Proper code formatting and indentation improve code readability and comprehension.

Use consistent indentation (e.g., 2 or 4 spaces) to indicate code blocks and hierarchy.

Group related code together and separate different sections with blank lines for clarity.

Break long lines of code into multiple lines to avoid excessive horizontal scrolling.

Align similar elements vertically to enhance code readability (e.g., aligning equal signs or commas).

Documentation and Self-Explanatory Code

Documenting your code is crucial for understanding its purpose, inputs, and outputs.

Use comments to explain complex logic, assumptions, or important considerations within your code.

Document functions using roxygen-style comments before their definitions to provide clear explanations of their usage and behavior.

Write self-explanatory code by using meaningful variable names, following established conventions, and avoiding excessive complexity.

Clear and concise code reduces the need for excessive commenting and facilitates code understanding.

Version Control and Collaboration

Version control systems like Git provide efficient ways to manage code versions, collaborate with others, and track changes.

By using version control, you can easily revert to previous versions, collaborate with team members, and merge code changes seamlessly.

Platforms like GitHub and GitLab host repositories, facilitate collaboration, and provide additional features like issue tracking and project management.

R for Basic Math - Arithmetic

R provides a wide range of arithmetic operators for performing basic mathematical calculations.

Addition: The + operator adds two numbers together. For example, 3 + 5 evaluates to 8.

Subtraction: The - operator subtracts one number from another. For example, 10 - 4 evaluates to 6.

Multiplication: The * operator multiplies two numbers. For example, 2 * 6 evaluates to 12.

Division: The / operator divides one number by another. For example, 15 / 3 evaluates to 5.

Modulo: The %% operator calculates the remainder after division. For example, 17 %% 5 evaluates to 2.

Exponentiation: The ^ operator raises a number to a specified power. For example, 2^3 evaluates to 8.

R for Basic Math - Logarithms and Exponentials

R provides functions for computing logarithms and exponentials.

Logarithms: The log() function calculates the natural logarithm (base e) of a number. For example, log(10) evaluates to approximately 2.3026.

If you want to calculate logarithms with a different base, you can use the log(x, base) syntax. For example, log(100, base = 10) evaluates to 2.

Exponentials: The exp() function calculates the exponential of a number. For example, exp(2) evaluates to approximately 7.3891.

Additionally, you can use the ^ operator to raise a number to a specified power. For example, 2^3 evaluates to 8, as mentioned earlier.

R for Basic Math - Numeric Functions

R provides a variety of numeric functions that can be used for basic mathematical calculations.

Absolute value: The abs() function returns the absolute value of a number. For example, abs(-5) evaluates to 5.

Square root: The sqrt() function calculates the square root of a number. For example, sqrt(25) evaluates to 5.

Ceiling and floor: The ceiling() function returns the smallest integer greater than or equal to a number, while the floor() function returns the largest integer less than or equal to a number.

Trigonometric functions: R provides functions like sin(), cos(), and tan() for trigonometric calculations.

Check the R documentation or use the help functions (?functionName) for more information on additional numeric functions available in R.

R for Basic Math - Working with Vectors

R is particularly powerful when working with vectors, which are one-dimensional arrays of values.

Arithmetic operations can be performed on entire vectors, making calculations more efficient.

For example, if you have two vectors x and y with the same length, you can add them together using the + operator: x + y.

Similarly, you can multiply two vectors element-wise using the * operator: x * y.

R's vectorized operations eliminate the need for explicit loops and enable concise and efficient code.

ENOTATION:

E-notation Overview:

E-notation, also known as scientific notation, is a compact way to represent very large or very small numbers.
It uses the format "aEb" where "a" is a number between 1 and 10 (mantissa), and "b" is an exponent of 10.

E-notation in R:

R programming language supports E-notation for representing numeric values.
It is commonly used in scientific and statistical calculations and data analysis.
E-notation provides a concise representation of large or small numbers.

Syntax and Examples:

The basic syntax for using E-notation in R is: numberEexponent
Examples:
3.14E5 represents 3.14 multiplied by 10^5 (314,000)
2.5E-3 represents 2.5 multiplied by 10^-3 (0.0025)

Converting E-notation to Numeric:

In R, E-notation is automatically converted to numeric values.
Example:

x <- 3.14E5 assigns the value 314000 to variable x.

Using E-notation in Calculations:

E-notation is particularly useful in scientific calculations involving large or small numbers.

It allows for more manageable and precise computations.

Example:

x <- 3.14E5

y <- 2.5E-3

z <- x * y

z will be equal to 785

Summary:

E-notation is a compact and efficient way to represent large or small numbers in R programming.

It is widely used in scientific and statistical calculations.

R automatically converts E-notation to numeric values for computations.

Assigning Objects in R Programming Introduction:

Objects in R:

In R, objects are used to store data or values.

They can be variables, vectors, matrices, data frames, etc.

Objects allow us to manipulate and analyze data in R.

Assigning Objects:

Assigning values to objects is done using the assignment operator "<-" or "=".

The general syntax is: object_name <- value or object_name = value.

Examples of Assigning Objects:

x <- 5 assigns the value 5 to the object "x".

y <- "Hello" assigns the string "Hello" to the object "y".

z <- c(1, 2, 3) assigns a vector of values 1, 2, and 3 to the object "z".

Rules for Object Naming:

Object names must start with a letter or a dot.

They can include letters, numbers, and dots.

Avoid using reserved words as object names (e.g., if, else, for).

Best Practices:

Use descriptive and meaningful names for objects.

Avoid single-letter variable names unless they have a clear purpose.

Comment your code to provide context and improve readability.

Assigning Objects in Functions:

Objects can be assigned within functions.

They are typically local to the function and not accessible outside.

The assignment operator works the same way within functions.

Multiple Assignments:

R allows assigning multiple objects at once using the c() function.

Example: x <- y <- z <- 10 assigns the value 10 to all three objects.

Summary:

Objects in R are used to store data or values.

Assignment of objects is done using the "<-" or "=" operator.

Follow best practices for naming objects and improving code readability.

Vectors in R Programming Introduction:

Vectors Overview:

Vectors are one-dimensional arrays that can hold multiple elements of the same data type.

They are fundamental data structures in R.

Vectors allow efficient manipulation and analysis of data.

Creating Vectors:

Vectors can be created using the **c()** function.

Example: **x <- c(1, 2, 3, 4, 5)** creates a numeric vector with values 1, 2, 3, 4, and 5.

Creating Sequences:

The **seq()** function generates a sequence of numbers.

Example: **x <- seq(1, 10, by = 2)** creates a sequence from 1 to 10, with a step of 2.

Repeating Elements:

The **rep()** function replicates elements of a vector.

Example: **x <- rep(1:3, times = 2)** creates a vector repeating the values 1, 2, 3, 1, 2, and 3.

Sorting Vectors:

The **sort()** function sorts the elements of a vector in ascending order.

Example: **x <- c(3, 1, 5, 2, 4)** sorts the vector in ascending order: 1, 2, 3, 4, 5.

Lengths of Vectors:

The **length()** function returns the number of elements in a vector.

Example: **x <- c(1, 2, 3)** returns the length of the vector, which is 3.

Subsetting Vectors:

Subsetting allows selecting specific elements from a vector.

Example: **x <- c(1, 2, 3, 4, 5)** subsetting **x[2:4]** returns the elements 2, 3, and 4.

Element Extraction:

Extracting elements from a vector is done using indexing.

Example: **x <- c(1, 2, 3, 4, 5)** extracting **x[3]** returns the third element, which is 3.

Vector-Oriented Behavior:

R exhibits vector-oriented behavior, meaning many operations are automatically applied element-wise to vectors.

Example: **x <- c(1, 2, 3)** multiplied by **y <- c(2, 4, 6)** results in **z <- c(2, 8, 18)**.

Practical Examples:

Demonstrate practical use cases where vector-oriented behavior in R simplifies data manipulation and analysis.

Summary:

Vectors are one-dimensional arrays used to store and manipulate data in R.

They can be created using **c()**, **seq()**, or **rep()** functions.

Vectors support operations such as sorting, subsetting, and element extraction.

R's vector-oriented behavior allows efficient data manipulation.

UNIT II - MATRICES AND ARRAYS

Defining a Matrix in R Programming Introduction:

Matrix Overview:

A matrix is a two-dimensional data structure in R.

It consists of rows and columns.

Matrices are useful for organizing and manipulating tabular data.

Defining a Matrix:

Matrices can be defined using the **matrix()** function.

Syntax: **matrix(data, nrow, ncol, byrow)**

Parameters:

data: The data to be filled in the matrix.

nrow : Number of rows in the matrix.

ncol: Number of columns in the matrix.

byrow: Optional parameter to specify the filling direction (default is by column).

Defining a Matrix - Example 1:

Example: **matrix(1:9, nrow = 3, ncol = 3)**

Creates a 3x3 matrix filled with values 1 to 9.

Elements are filled by column (default).

Defining a Matrix - Example 2:

Example: **matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)**

Creates a 2x3 matrix filled with values 1 to 6.

Elements are filled by row.

Matrix Dimensions:

The dimensions of a matrix can be obtained using the **dim()** function.

Syntax: **dim(matrix)**

Returns a vector with the number of rows and columns.

Filling a Matrix with Values:

Matrices can be filled with values directly during creation.

Example: **matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)**

Creating a Matrix from Existing Data:

Matrices can be created from existing data structures like vectors or lists.

Example: **matrix(vector, nrow, ncol)**

Example: **matrix(list, nrow, ncol)**

Summary:

A matrix is a two-dimensional data structure in R.

Matrices can be defined using the **matrix()** function.

The **nrow** and **ncol** parameters specify the dimensions.

The **byrow** parameter controls the filling direction.

Matrices can be filled with values directly or created from existing data.

Filling Direction:

Filling Direction Overview:

Matrices are filled with values row by row or column by column.

Filling direction affects how elements are arranged in a matrix.

Understanding filling direction is crucial for data organization and analysis.

Default Filling Direction:

By default, matrices are filled column by column in R.

Values are assigned to the first column, then the second column, and so on.

Changing Filling Direction:

The **byrow** argument controls the filling direction in the **matrix()** function.

Syntax: **matrix(data, nrow, ncol, byrow = FALSE)**

Set **byrow** to **TRUE** for row-wise filling.

Filling Direction - Example 1:

Example: **matrix(1:9, nrow = 3, ncol = 3)**

Creates a 3x3 matrix with values 1 to 9.

Values are filled column by column (default).

Filling Direction - Example 2:

Example: **matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)**

Creates a 3x3 matrix with values 1 to 9.

Values are filled row by row.

Importance of Filling Direction:

Filling direction affects how data is organized and interpreted.

It can impact calculations, analyses, and visualizations.

Consider the natural flow of data when choosing the appropriate filling direction.

Practical Applications:

Filling direction is essential when working with time series data.

It affects the order of operations, such as smoothing or differencing.

It impacts plotting and visualization, especially for line plots and heatmaps.

Summary:

Filling direction determines how elements are arranged in a matrix.

By default, matrices are filled column by column in R.

The **byrow** argument in the **matrix()** function controls the filling direction.

Consider the nature of your data and analysis when choosing the appropriate filling direction.

Row and Column Bindings:

Row and Column Bindings in R Programming Introduction:

Row and column bindings.

Combining matrices and vectors.

Concatenating rows and columns.

Examples and practical applications.

Row and Column Bindings Overview:

Row binding: Combining matrices or vectors vertically.

Column binding: Combining matrices or vectors horizontally.

Bindings allow merging data structures of compatible dimensions.

Row Binding:

Row binding combines matrices or vectors vertically.

The **rbind()** function is used for row binding.

Syntax: **rbind(object1, object2, ...)**

Row Binding - Example:

Example: **matrix1 <- matrix(1:6, nrow = 2)**

Example: **matrix2 <- matrix(7:12, nrow = 2)**

Example: **rbind(matrix1, matrix2)**

Result: A 4x3 matrix combining matrix1 and matrix2 by rows.

Column Binding:

Column binding combines matrices or vectors horizontally.

The **cbind()** function is used for column binding.

Syntax: **cbind(object1, object2, ...)**

Column Binding - Example:

Example: **matrix1 <- matrix(1:6, nrow = 2)**

Example: **matrix2 <- matrix(7:12, nrow = 2)**

Example: **cbind(matrix1, matrix2)**

Result: A 2x6 matrix combining matrix1 and matrix2 by columns.

Combining Matrices and Vectors:

Matrices and vectors can be combined using row or column bindings.

Dimensions must be compatible for successful binding.

Vectors are automatically transformed into a matrix before binding.

Combining Matrices and Vectors - Example:

Example: **matrix1 <- matrix(1:6, nrow = 2)**

Example: **vector1 <- c(7, 8)**

Example: **cbind(matrix1, vector1)**

Result: A 2x4 matrix combining matrix1 and vector1 by columns.

Practical Applications:

Combining data from different sources or experiments.

Merging datasets horizontally or vertically.

Creating larger matrices for analysis or visualization.

Summary:

Row binding combines matrices or vectors vertically using **rbind()**.

Column binding combines matrices or vectors horizontally using **cbind()**.

Matrices and vectors must have compatible dimensions for successful binding.

Row and column bindings are useful for merging data structures in R.

Matrix Dimensions

Matrix Dimensions Overview:

Matrix dimensions define the size and shape of a matrix.

Dimensions include the number of rows and columns.

Manipulating dimensions helps organize and analyze data efficiently.

Retrieving Matrix Dimensions:

The **dim()** function is used to retrieve matrix dimensions.

Syntax: **dim(matrix)**

Returns a vector with the number of rows and columns.

Retrieving Matrix Dimensions - Example:

Example: **matrix1 <- matrix(1:6, nrow = 2)**

Example: **dim(matrix1)**

Result: A vector **[2, 3]** representing 2 rows and 3 columns.

Manipulating Matrix Dimensions:

The **dim()** function can be used to set or change matrix dimensions.

Syntax: **dim(matrix) <- c(nrow, ncol)**

Manipulating Matrix Dimensions - Example:

Example: **matrix1 <- matrix(1:6, nrow = 2)**

Example: **dim(matrix1) <- c(3, 2)**

Result: The matrix dimensions are changed to 3 rows and 2 columns.

Matrix Dimension Attributes:

Matrix dimensions are stored as attributes.

The **attributes()** function can be used to access and modify attributes.

Matrix Dimension Attributes - Example:

Example: **matrix1 <- matrix(1:6, nrow = 2)**

Example: **attributes(matrix1)$dim <- c(3, 2)**

Result: The matrix dimensions are changed to 3 rows and 2 columns.

Practical Applications:

Reshaping matrices for analysis or visualization.

Manipulating dimensions for compatibility with other functions.

Extracting and rearranging data based on desired dimensions.

Summary:

Matrix dimensions define the size and shape of a matrix.

The **dim()** function retrieves matrix dimensions.

Dimensions can be manipulated using **dim()** or **attributes()**.

Understanding and manipulating matrix dimensions is crucial for data analysis.

Subsetting

Subsetting Overview:

Subsetting refers to extracting a portion of a data structure.

It helps focus on specific data elements of interest.

Subsetting is essential for data analysis and manipulation.

Types of Subsetting:

Subsetting Vectors:

Extracting specific elements from a vector.

Subsetting based on indices or logical conditions.

Subsetting Matrices and Data Frames:

Extracting rows, columns, or specific elements.

Subsetting based on indices or logical conditions.

Subsetting with Brackets:

Square brackets **[ ]** are used for subsetting in R.

Syntax: **object[subset]**

Subset can be indices, logical conditions, or a combination.

Subsetting Vectors - Example 1:

Example: **my_vector<- c(10, 20, 30, 40, 50)**

Example: **my_vector[3]**

Result: Extracts the third element, 30.

Subsetting Vectors - Example 2:

Example: **my_vector<- c(10, 20, 30, 40, 50)**

Example: **my_vector[c(2, 4)]**

Result: Extracts the second and fourth elements, 20 and 40.

Subsetting Matrices - Example 1:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **my_matrix[2, 3]**

Result: Extracts the element in the second row, third column, which is 6.

Subsetting Matrices - Example 2:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **my_matrix[2,]**

Result: Extracts the entire second row of the matrix.

Logical Subsetting - Example 1:

Example: **my_vector<- c(10, 20, 30, 40, 50)**

Example: **my_vector> 30**

Result: Returns a logical vector indicating elements greater than 30.

Logical Subsetting - Example 2:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **my_matrix[my_matrix> 5]**

Result: Extracts elements in the matrix greater than 5.

Practical Applications:

Filtering data based on specific criteria.

Selecting relevant subsets for analysis or visualization.

Modifying or replacing specific elements within a data structure.

Summary:

Subsetting involves extracting a portion of a data structure.

Square brackets **[ ]** are used for subsetting in R.

Subsetting can be done with indices, logical conditions, or a combination.

Understanding subsetting is crucial for data manipulation and analysis.

Row, Column, and Diagonal Extractions:

Row Extraction:

Extracting specific rows from a matrix.

The **[]** brackets with row indices are used for row extraction.

Row Extraction - Example:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **my_matrix[2,]**

Result: Extracts the second row of the matrix.

Column Extraction:

Extracting specific columns from a matrix.

The **[]** brackets with column indices are used for column extraction.

Column Extraction - Example:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **my_matrix[, 3]**

Result: Extracts the third column of the matrix.

Diagonal Extraction:

Extracting the main diagonal elements from a matrix.

The **diag()** function is used for diagonal extraction.

Diagonal Extraction - Example:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **diag(my_matrix)**

Result: Extracts the main diagonal elements of the matrix.

Extracting Multiple Rows, Columns, and Diagonals:

Multiple rows, columns, or diagonals can be extracted simultaneously.

Combine row, column, or diagonal indices within the **[]** brackets.

Extracting Multiple Rows, Columns, and Diagonals - Example:

Example: **my_matrix<- matrix(1:9, nrow = 3)**

Example: **my_matrix[c(1, 3),]**

Result: Extracts the first and third rows of the matrix.

Practical Applications:

Selecting specific subsets of data for analysis or visualization.

Extracting features or attributes of interest from a matrix.

Working with triangular matrices or symmetric matrices.

Summary:

Row extraction is done using the **[]** brackets with row indices.

Column extraction is done using the **[]** brackets with column indices.

Diagonal extraction is done using the **diag()** function.

Understanding these extraction techniques is essential for data manipulation and analysis.

Omitting and Overwriting:

Omitting Observations:

Omitting observations with missing values.

The **na.omit()** function removes rows with missing values.

Syntax: **na.omit(data)**

Omitting Observations - Example:

Example: **my_data<- data.frame(x = c(1, NA, 3), y = c(4, 5, 6))**

Example: **na.omit(my_data)**

Result: Removes the row with the missing value.

Omitting Variables:

Omitting variables from a dataset.

The **$** operator or **subset()** function can be used.

Syntax: **data$variable** or **subset(data, select = -variable)**

Omitting Variables - Example:

Example: **my_data<- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))**

Example: **my_data$y<- NULL**

Result: Omits the "y" variable from the dataset.

Overwriting Data Values:

Modifying or overwriting specific data values.

Direct assignment using the **[]** brackets.

Syntax: **data[rows, columns] <- value**

Overwriting Data Values - Example:

Example: **my_data<- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))**

Example: **my_data[2, 1] <- 99**

Result: Overwrites the value in the second row, first column.

Overwriting Multiple Data Values:

Overwriting multiple data values simultaneously.

Use logical conditions within the **[]** brackets.

Overwriting Multiple Data Values - Example:

Example: **my_data<- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))**

Example: **my_data[my_data$x> 2, "y"] <- 99**

Result: Overwrites the "y" values where "x" is greater than 2.

Practical Applications:

Handling missing or incomplete data in datasets.

Customizing datasets by omitting irrelevant observations or variables.

Modifying specific data values for data cleaning or transformation.

Summary:

Omitting observations with missing values is done using **na.omit()**.

Omitting variables is achieved through the **$** operator or **subset()** function.

Overwriting data values is done using the **[]** brackets.

Understanding these techniques is essential for data cleaning and manipulation.

Matrix Operations and Algebra:

Arithmetic Operations on Matrices:

Matrices can be added, subtracted, and multiplied element-wise.

The **+**, **-**, and ***** operators are used for arithmetic operations.

Matrix Addition:

Adding two matrices together.

Matrices must have the same dimensions.

Syntax: **matrix1 + matrix2**

Matrix Addition - Example:

Example: **matrix1 <- matrix(1:4, nrow = 2)**

Example: **matrix2 <- matrix(5:8, nrow = 2)**

Example: **matrix1 + matrix2**

Result: Element-wise addition of the two matrices.

Matrix Subtraction:

Subtracting one matrix from another.

Matrices must have the same dimensions.

Syntax: **matrix1 - matrix2**

Matrix Subtraction - Example:

Example: **matrix1 <- matrix(1:4, nrow = 2)**

Example: **matrix2 <- matrix(5:8, nrow = 2)**

Example: **matrix1 - matrix2**

Result: Element-wise subtraction of the two matrices.

Matrix Multiplication:

Multiplying matrices.

The number of columns in the first matrix must equal the number of rows in the second matrix.

Syntax: **matrix1 %*% matrix2**

Matrix Multiplication - Example:

Example: **`matrix1 <- matrix(1:4, nrow = 2)`**

Example: **`matrix2 <- matrix(5:8, nrow = 2)`**

Example: **`matrix1 %*% matrix2`**

Result: Matrix multiplication of the two matrices.

Matrix Algebra:

Additional matrix algebra operations:

Matrix transpose: **`t(matrix)`**

Matrix inversion: **`solve(matrix)`**

Identity matrix: **`diag(n)`**

Matrix Transpose:

Flipping the matrix along its diagonal.

Rows become columns and columns become rows.

Syntax: **`t(matrix)`**

Matrix Inversion:

Finding the inverse of a square matrix.

The matrix must be invertible.

Syntax: **`solve(matrix)`**

Identity Matrix:

A square matrix with ones on the main diagonal and zeros elsewhere.

Useful for matrix algebra and calculations.

Syntax: **`diag(n)`**

Practical Applications:

Linear transformations and systems of equations.

Data analysis and manipulation.

Statistical modeling and regression.

Summary:

Matrices can be added, subtracted, and multiplied element-wise.

Matrix algebra operations include transpose, inversion, and identity matrix.

Understanding matrix operations is crucial for various data analysis tasks.

Matrix Transpose:

Matrix Transpose Overview:

Transposing a matrix involves interchanging its rows and columns.

The transpose of an m x n matrix is an n x m matrix.

Importance of matrix transpose in various mathematical and computational operations.

Transposing a Matrix in R:

R provides a built-in function for matrix transpose.

The **t()** function is used to transpose a matrix.

Syntax: **t(matrix)**

Matrix Transpose - Example:

Example: **my_matrix<- matrix(1:6, nrow = 2)**

Example: **t(my_matrix)**

Result: Transposes the matrix, interchanging rows and columns.

Properties of Matrix Transpose:

Transposing a transposed matrix returns the original matrix: **(t(t(matrix)) == matrix)**

Transpose of a sum of matrices is equal to the sum of transposed matrices: **t(matrix1 + matrix2) == t(matrix1) + t(matrix2)**

Transposing a Data Frame:

Data frames can also be transposed using the **t()** function.

Columns become rows and rows become columns.

Useful for reshaping and reorganizing data.

Transposing a Data Frame - Example:

Example: **my_data<- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))**

Example: **t(my_data)**

Result: Transposes the data frame, interchanging columns and rows.

Practical Applications:

Matrix operations and calculations.

Reshaping and reorganizing data.

Machine learning and data analysis.

Summary:

Matrix transpose involves interchanging rows and columns of a matrix.

The **t()** function in R is used for matrix transpose.

Transposing a data frame can be helpful for data manipulation.

Understanding matrix transpose is important for various computational tasks.

Identity Matrix

Identity Matrix Overview:

An identity matrix is a square matrix where the main diagonal contains ones and all other elements are zeros.

Denoted as I or I<sub>n</sub>, where n represents the dimensions of the matrix.

Important properties and uses in linear algebra and computations.

Identity Matrix Properties:

The product of any matrix and its corresponding identity matrix is the matrix itself.

Symbolic representation: A * I = A, where A is any matrix.

I<sub>n</sub> * A = A * I<sub>n</sub> = A, where A is an n x m matrix.

Creating Identity Matrices in R:

R provides functions to create identity matrices.

The **diag()** function is used to create an identity matrix.

Syntax: **diag(n)** creates an n x n identity matrix.

Identity Matrix - Example:

Example: **identity_matrix<- diag(3)**

Result: Creates a 3 x 3 identity matrix.

Properties of Identity Matrices:

Identity matrices retain their properties in various mathematical operations.

Addition or subtraction of identity matrices does not change the result.

Multiplication of a matrix by the identity matrix preserves the matrix.

Identity Matrix Application - Matrix Multiplication:

Multiplying a matrix by an identity matrix preserves the original matrix.

Example: A * I = A, where A is any matrix.

Useful in computations and transformations without altering the matrix.

Identity Matrix Application - Linear Algebra:

Identity matrices play a crucial role in linear algebra operations.

Used in solving systems of linear equations, matrix inverses, and determinant calculations.

Identity Matrix Application - Data Transformations:

Identity matrices can be used to transform and manipulate data.

Scaling, rotation, and translation operations in data analysis and computer graphics.

Practical Applications:

Linear algebra and matrix calculations.

Geometric transformations.

Statistical modeling and data analysis.

Summary:

An identity matrix is a square matrix with ones on the main diagonal and zeros elsewhere.

Created using the **diag()** function in R.

Retains its properties in mathematical operations.

Has various applications in linear algebra and data transformations.

Matrix Addition:

Introduction

Briefly introduce the topic and its relevance in data analysis and computations

Highlight the importance of matrix addition in various mathematical and statistical operations

Overview of R Programming

Provide a brief introduction to R programming language and its features

Mention that R has built-in support for matrix operations

Creating Matrices in R

Explain different ways to create matrices in R, such as using the matrix() function or by converting vectors into matrices

Matrix Addition Concept

Explain the concept of matrix addition

Discuss how matrix addition works by adding corresponding elements of matrices together

Matrix Addition Example

Present an example to demonstrate matrix addition in R

Show the step-by-step process of adding two matrices together

Broadcasting in Matrix Addition

Discuss the concept of broadcasting in matrix addition

Explain how R automatically aligns matrices of different dimensions during addition

Element-wise Matrix Addition

Discuss the concept of element-wise matrix addition

Explain how R allows for adding matrices element by element using the "+" operator

R Functions for Matrix Addition

Discuss built-in R functions for matrix addition, such as "+", "add", or "mat.plus"

Explain their usage and any additional parameters they might accept

Error Handling in Matrix Addition

Discuss common errors that may occur during matrix addition, such as incompatible dimensions
Explain how R handles such errors and provides informative error messages

Performance Considerations

Discuss performance considerations when working with large matrices
Mention the use of optimized R packages for matrix operations, such as "matrixStats" or "gmatrix"

Conclusion

Recap the main points covered in the presentation
Emphasize the importance of matrix addition in R programming and data analysis

Additional Resources
Provide a list of recommended resources, such as books, online tutorials, or documentation, for further learning about matrix operations in R

UNIT III NON – NUMERIC VALUES

Handling Non-Numeric Values

Introduction:

- In R programming, non-numeric values are used to represent text orcategorical data.

- Understanding how to work with non-numeric values is essential for dataanalysis and manipulation.

- Let's explore some techniques for handling non-numeric values in R.

Character Values:

- Character values represent text in R.

- They are enclosed in quotes (either single or double).

- Example: `name <- "John Doe"`

Creating Character Vectors:

- Character vectors are sequences of character values.

- They are created using the `c()` function.

- Example: `fruits <- c("apple", "banana", "orange")`

Concatenating Character Values:

- The `paste()` function is used to combine character values.

- Example: `greeting <- "Hello"`

- Example: `subject <- "world"`

- Example: `message <- paste(greeting, subject)`

Accessing Individual Characters:

- The `substr()` function extracts specific characters from a string.

- Example: `text <- "Hello"`

- Example: `first_letter <- substr(text, 1, 1)`

Comparing Strings:

- The `==` operator is used to compare strings for equality.

- Example: `color1 <- "blue"`

- Example: `color2 <- "red"`

- Example: `is_equal <- color1 == color2`

String Functions:

- R provides numerous string manipulation functions.

- Example: `text <- "OpenAI"`

- Example: `uppercase_text <- toupper(text)`

Conclusion:

- Non-numeric values, such as character or string values, are widely used in R programming.

- Understanding how to handle and manipulate non-numeric values is crucial for data analysis and processing.

- Armed with these techniques, you can confidently work with non-numericdata in R.

Handling Logical Values

Introduction:

- Logical values in R represent the truth or falsehood of a statement.

- Understanding how to work with logical values is essential for decisionmaking and data filtering.

- Let's explore some techniques for handling logical values in R.

Logical Operators:

- R provides several logical operators for evaluating conditions.

- The most common operators are:
  - `==` for equality
  - `!=` for inequality
  - `>` for greater than
  - `<` for less than
  - `>=` for greater than or equal to
  - `<=` for less than or equal to

Creating Logical Vectors:

- Logical vectors are sequences of logical values.
- They can be created using logical operators.
- Example: `grades <- c(85, 92, 77, 95)`
- Example: `passing <- grades >= 80`

Logical Functions:

- R provides functions to perform logical operations.
- The most commonly used functions are:
  - `all()` to check if all values are TRUE
  - `any()` to check if any value is TRUE
  - `which()` to find the indices of TRUE values
  - `isTRUE()` to check if a value is TRUE

Combining Logical Values:

- The `&` operator performs element-wise logical AND.
- The `|` operator performs element-wise logical OR.
- Example: `A <- c(TRUE, TRUE, FALSE, FALSE)`
- Example: `B <- c(TRUE, FALSE, TRUE, FALSE)`
- Example: `result <- A & B`

Negating Logical Values:

- The `!` operator negates a logical value.

- Example: `A <- c(TRUE, FALSE, TRUE)`

- Example: `negated <- !A`

Conditional Statements:

- Conditional statements are used to perform different actions based on logicalconditions.

- The `if`, `else if`, and `else` statements are commonly used.

- Example:

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

Conclusion:

- Logical values play a vital role in decision making and data filtering in R programming.

- By understanding logical operators, functions, and conditional statements, you can effectively handle logical values.

- These techniques empower you to make informed decisions and filter data based on conditions.

Relational Operators in R Programming

Introduction:

- Relational operators in R are used to compare values and evaluate conditions.

- They are crucial for decision making, filtering data, and controlling programflow.
- Let's explore the relational operators available in R and how they can beused.

Relational Operators:

- R provides several relational operators:
  - `==` for equality
  - `!=` for inequality
  - `>` for greater than
  - `<` for less than
  - `>=` for greater than or equal to
  - `<=` for less than or equal to

Example: Equality Operator

- The `==` operator checks if two values are equal.
- Returns `TRUE` if the values are equal, `FALSE` otherwise.
- Example: `5 == 5` evaluates to `TRUE`.
- Example: `7 == 3` evaluates to `FALSE`.

Example: Inequality Operator

- The `!=` operator checks if two values are not equal.
- Returns `TRUE` if the values are not equal, `FALSE` otherwise.
- Example: `5 != 5` evaluates to `FALSE`.
- Example: `7 != 3` evaluates to `TRUE`.

Example: Greater Than and Less Than Operators

- The `>` operator checks if the left value is greater than the right value.
- The `<` operator checks if the left value is less than the right value.
- Returns `TRUE` if the condition is satisfied, `FALSE` otherwise.

- Example: `5 > 3` evaluates to `TRUE`.

- Example: `7 < 3` evaluates to `FALSE`.

Example: Greater Than or Equal To and Less Than or Equal To Operators

- The `>=` operator checks if the left value is greater than or equal to the rightvalue.

- The `<=` operator checks if the left value is less than or equal to the rightvalue.

- Returns `TRUE` if the condition is satisfied, `FALSE` otherwise.

- Example: `5 >= 5` evaluates to `TRUE`.

- Example: `7 <= 3` evaluates to `FALSE`.

Combining Relational Operators:

- Relational operators can be combined using logical operators (`&` and `|`) to form complex conditions.

- Example: `(x > 5) & (x < 10)` checks if `x` is greater than 5 and less than 10.

Using Relational Operators for Data Filtering:

- Relational operators are often used to filter data based on specific conditions.

- Example: `filtered_data <- data[data$age > 30, ]` selects rows from a dataframe where the age column is greater than 30.

Conclusion:

- Relational operators are powerful tools for comparing values and evaluatingconditions in R programming.

- By understanding and utilizing these operators effectively, you can make informed decisions, filter data, and control program flow.

- Mastering relational operators expands your capabilities for data analysis and manipulation in R.

Working with Characters in R Programming

Introduction:

- Characters play a crucial role in representing and manipulating text data in R programming.

- Understanding how to work with characters is essential for tasks like datacleaning, text processing, and string manipulation.

- Let's explore the fundamentals of working with characters in R.

Character Data Type:

- In R, characters are used to represent text and are enclosed in quotes (either single or double).

- Example: `name <- "John Doe"`

Creating Character Vectors:

- Character vectors are sequences of character values.

- They are created using the `c()` function.

- Example: `fruits <- c("apple", "banana", "orange")`

Concatenating Characters:

- The `paste()` function combines character values into a single string.

- Example: `greeting <- "Hello"`

- Example: `subject <- "world"`

- Example: `message <- paste(greeting, subject)`

Accessing Characters:

- Individual characters within a string can be accessed using indexing.

- Example: `text <- "Hello"`

- Example: `first_letter <- substr(text, 1, 1)`

String Manipulation:

- R provides various functions to manipulate character strings.

- Common functions include `tolower()`, `toupper()`, `substr()`, `gsub()`, and `strsplit()`.

- Example: `text <- "OpenAI"`

- Example: `lowercase_text <- tolower(text)`

Pattern Matching:

- Regular expressions (regex) enable pattern matching within character strings.

- Functions like `grep()`, `grepl()`, and `sub()` facilitate regex pattern matching.

- Example: `text <- c("apple", "banana", "orange")`

- Example: `matching_fruits <- grep("an", text, value = TRUE)`

Comparing Characters:

- The `==` and `!=` operators compare character values for equality orinequality.

- Example: `color1 <- "blue"`

- Example: `color2 <- "red"`

- Example: `is_equal <- color1 == color2`

Conclusion:

- Characters are fundamental for working with text data in R programming.

- By leveraging character vectors, string manipulation functions, and pattern matching techniques, you can effectively process and analyze textual information.

- Understanding characters expands your capabilities for data cleaning, text processing, and string manipulation tasks in R.

Creating Strings

Introduction:

- Strings, or character values, are widely used in R programming to represent and manipulate text data.

- Understanding how to create and work with strings is essential for varioustasks, such as data cleaning, text processing, and generating output.

- Let's explore the different ways to create strings in R.

Using Double Quotes:

- Strings can be created by enclosing text within double quotes (" ").

- Example: `message <- "Hello, World!"`

Using Single Quotes:

- Strings can also be created by enclosing text within single quotes (' ').

- Example: `greeting <- 'Welcome to R programming!'`

Combining Strings:

- Strings can be combined using the `paste()` function or the concatenation operator (`c()`).

- Example using `paste()`: `full_name <- paste("John", "Doe")`

- Example using `c()`: `fruit <- c("apple", "banana", "orange")`

Using Escaping Characters:

- Special characters can be included in strings using escaping characters like backslash (\).

- Example: `escaped_string <- "I\'m learning R programming."`

Using Line Breaks and Tabs:

- Line breaks and tabs can be added to strings for formatting using escapesequences.

- Example: `multiline_string <- "First line.\nSecond line."`

- Example: `tabbed_string <- "Name:\tJohn\nAge:\t25"`

Using Variables in Strings:

- Variables can be embedded within strings using the `paste()` function or the `sprintf()` function.

- Example using `paste()`: `name <- "John"; message <- paste("Hello,", name)`

- Example using `sprintf()`: `count <- 10; formatted_string <- sprintf("The count is %d", count)`

Using String Interpolation:

- R supports string interpolation using the `glue` or `stringr` packages.

- Example using `glue`: `name <- "John"; message <- glue("Hello, {name}!")`

- Example using `stringr`: `library(stringr);  name <- "John"; message <- str_interp("Hello, {name}!")`

Conclusion:

- Creating strings is an essential skill in R programming for working with textdata.

- By utilizing quotes, escaping characters, concatenation, and interpolation techniques, you can generate and manipulate strings effectively.

- Understanding string creation expands your capabilities for data cleaning, text processing, and generating dynamic output in R.

Concatenation

Introduction:

- Concatenation is the process of combining or joining multiple elementstogether.

- In R programming, concatenation is commonly used for combining vectors, strings, or data frames.

- Let's explore the various ways to perform concatenation in R.

Concatenating Numeric Vectors:

- Numeric vectors can be concatenated using the `c()` function.

- Example: `numbers <- c(1, 2, 3)`

- Example: `more_numbers <- c(numbers, 4, 5)`

Concatenating Character Vectors:

- Character vectors can also be concatenated using the `c()` function.

- Example: `fruits <- c("apple", "banana")`

- Example: `more_fruits <- c(fruits, "orange")`

Concatenating Using the `paste()` Function:

- The `paste()` function concatenates multiple strings or elements into a single character string.

- Example: `greeting <- "Hello"`

- Example: `name <- "John"`

- Example: `message <- paste(greeting, name)`

Concatenating Using the `paste0()` Function:

- The `paste0()` function is similar to `paste()`, but it concatenates without anyseparator.

- Example: `greeting <- "Hello"`
- Example: `name <- "John"`

- Example: `message <- paste0(greeting, name)`

Concatenating Data Frames:

- Data frames can be concatenated row-wise or column-wise using functionslike `rbind()` and `cbind()`.

- Example: `df1 <- data.frame(x = 1:3, y = 4:6)`

- Example: `df2 <- data.frame(x = 7:9, y = 10:12)`

- Example: `combined <- rbind(df1, df2)`

Concatenating Using the `append()` Function:

- The `append()` function is used to append elements to a vector or list.

- Example: `numbers <- c(1, 2, 3)`

- Example: `more_numbers <- append(numbers, c(4, 5))`

Concatenating Using the `union()` Function:

- The `union()` function combines elements of two vectors, removingduplicates.

- Example: `x <- c(1, 2, 3)`

- Example: `y <- c(3, 4, 5)`

- Example: `combined <- union(x, y)`

Conclusion:

- Concatenation is a fundamental operation in R programming for combiningelements.

- By utilizing functions like `c()`, `paste()`, `rbind()`, and `append()`, you caneffectively concatenate vectors, strings, and data frames.

- Understanding different concatenation techniques expands your capabilities for data manipulation and analysis in R.

Escape Sequences

Introduction:

- Escape sequences are special characters used in strings to represent non-printable or special characters.

- R programming provides escape sequences to include characters that cannot be directly represented.

- Let's explore the common escape sequences used in R.

Escape Sequence:

- An escape sequence starts with a backslash (\) followed by a specificcharacter.

- The backslash tells R to treat the following character differently.

- Example: `"\n"` represents a line break.

Common Escape Sequences:

1. `\n` - Newline character
2. `\t` - Tab character
3. `\"` - Double quote character
4. `\'` - Single quote character
5. `\\` - Backslash character
6. `\b` - Backspace character
7. `\r` - Carriage return character

Example: Newline Character

- The `\n` escape sequence represents a newline character.
- Example: `message <- "Hello\nWorld"`

Example: Tab Character

- The `\t` escape sequence represents a tab character.
- Example: `message <- "Name:\tJohn"`

Example: Double Quote Character

- The `\"` escape sequence represents a double quote character.
- Example: `message <- "He said, \"Hello!\""`

Example: Single Quote Character

- The `\'` escape sequence represents a single quote character.
- Example: `message <- 'It\'s raining outside.'`

Example: Backslash Character

- The `\\` escape sequence represents a backslash character.

- Example: `message <- "C:\\Documents\\File.txt"`

Example: Backspace Character

- The `\b` escape sequence represents a backspace character.

- Example: `message <- "Hello\bWorld"`

Example: Carriage Return Character

- The `\r` escape sequence represents a carriage return character.

- Example: `message <- "123\rXYZ"`

Conclusion:

- Escape sequences are used in R programming to represent non-printable or special characters within strings.

- By utilizing escape sequences like `\n`, `\t`, `\"`, `\'`, `\\`, `\b`, and `\r`, you can include special characters in your strings.

- Understanding and using escape sequences expands your capabilities for textmanipulation and formatting in R.

Substring and Pattern Matching

Introduction:

- Substring extraction and pattern matching are essential operations in text processing and data manipulation.

- R programming provides powerful functions to extract substrings and perform pattern matching on character strings.

- Let's explore substring extraction and pattern matching techniques in R.

Substring Extraction:

- Substring extraction involves retrieving a portion of a string based on itsposition or length.

- The `substr()` function is commonly used for extracting substrings in R.

- Example: `text <- "Hello, World!"`

- Example: `substring <- substr(text, start = 8, stop = 13)`

Pattern Matching using `grep()`:

- The `grep()` function is used for pattern matching within character strings.

- It returns the indices of the matching elements.

- Example: `text <- c("apple", "banana", "orange")`

- Example: `matching_indices <- grep("an", text)`

Pattern Matching using `grepl()`:

- The `grepl()` function is similar to `grep()`, but it returns a logical vectorindicating whether a match was found or not.

- Example: `text <- c("apple", "banana", "orange")`

- Example: `matching_elements <- grepl("an", text)`

Pattern Matching using `sub()`:

- The `sub()` function is used for substitution or replacement based on apattern.

- It replaces the first occurrence of the pattern with a specified replacement.

- Example: `text <- "Hello, World!"`

- Example: `new_text <- sub("World", "Universe", text)`

Pattern Matching using Regular Expressions:

- Regular expressions (regex) provide advanced pattern matching capabilities.

- Functions like `grep()`, `grepl()`, and `sub()` support regex pattern matching.

- Example: `text <- c("apple", "banana", "orange")`

- Example: `matching_fruits <- grep("^a", text, value = TRUE)`

Pattern Matching using the `stringr` Package:

- The `stringr` package provides additional functions for pattern matching and string manipulation.

- Functions like `str_detect()`, `str_extract()`, and `str_replace()` offerconvenient regex-based operations.

- Example: `library(stringr)`

- Example: `text <- c("apple", "banana", "orange")`

- Example: `matching_fruits <- str_detect(text, "^a")`

Conclusion:

- Substring extraction and pattern matching are crucial operations in text processing and data manipulation in R.

- By utilizing functions like `substr()`, `grep()`, `grepl()`, and regular expressions, you can extract substrings and perform pattern matching effectively.

- Understanding substring extraction and pattern matching expands your capabilities for data cleaning, text processing, and data analysis in R.

Factors

Introduction:

- Factors are a unique data type in R used to represent categorical or discretevariables.

- Factors provide a way to store and analyze data with a fixed set of distinctvalues.

- Let's explore factors and their importance in R programming.

Creating Factors:

- Factors are created using the `factor()` function.

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

Levels:

- Factors have levels that define the distinct categories or values.

- Levels can be specified manually or inferred from the data.

- Example: `gender <- factor(c("Male", "Female", "Female", "Male"), levels =c("Male", "Female"))`

Accessing Levels:

- The `levels()` function is used to access the levels of a factor.

- Example: `gender_levels <- levels(gender)`

Modifying Levels:

- Levels can be modified using the `levels()` function.

- Example: `levels(gender) <- c("M", "F")`

Summary of Levels:

- The `table()` function provides a summary of the levels and their frequency.

- Example: `level_summary <- table(gender)`

Ordering Levels:

- The order of levels in a factor can be specified using the `ordered()` function.

- Example: `temperature <- ordered(c("Low", "Medium", "High"), levels = c("Low", "Medium", "High"))`

Working with Factors:

- Factors play a crucial role in statistical analysis, data visualization, andmodeling in R.

- They enable efficient handling of categorical data and support operations like grouping, cross-tabulation, and plotting.

- Example: `summary(gender)`

- Example: `table(gender)`

- Example: `plot(gender)`

Converting Factors to Character or Numeric:

- Factors can be converted to character or numeric vectors using the `as.character()` and `as.numeric()` functions.

- Example: `gender_character <- as.character(gender)`

- Example: `gender_numeric <- as.numeric(gender)`

Conclusion:

- Factors are a special data type in R used to represent categorical or discretevariables.

- They provide a structured way to store and analyze categorical data, enablingefficient data manipulation and analysis.

- Understanding factors expands your capabilities for data cleaning, analysis, and visualization in R.

Identifying Categories

Introduction:

- Identifying categories or distinct values in data is a common task in dataanalysis and manipulation.

- R programming offers several functions and techniques to identify categoriesefficiently.

- Let's explore the methods to identify categories in R.

Unique Values using `unique()`:

- The `unique()` function returns the unique values from a vector or column.

- Example: `data <- c(1, 2, 3, 2, 1)`

- Example: `unique_values <- unique(data)`

Frequency Table using `table()`:

- The `table()` function generates a frequency table, showing the counts ofeach unique value.

- Example: `data <- c("apple", "banana", "apple", "orange", "banana")`

- Example: `frequency_table <- table(data)`

Frequency Counts using `count()` (dplyr package):

- The `count()` function from the `dplyr` package provides a concise way to count the occurrences of each value.

- Example: `library(dplyr)`

- Example: `data <- c("apple", "banana", "apple", "orange", "banana")`

- Example: `frequency_counts <- count(data)`

Factor Levels:

- Factors can be used to represent categorical variables with predefined levels.

- The `levels()` function extracts the levels from a factor.

- Example: `data <- factor(c("A", "B", "C", "B", "A"))`

- Example: `factor_levels <- levels(data)`

Using the `levels()` Function (dplyr package):

- The `levels()` function from the `dplyr` package can be used to retrieve unique values from a column or factor.

- Example: `library(dplyr)`

- Example: `data <- c("apple", "banana", "apple", "orange", "banana")`

- Example: `unique_values <- data %>% levels()`

Using the `distinct()` Function (dplyr package):

- The `distinct()` function from the `dplyr` package returns distinct rows based on selected columns.

- Example: `library(dplyr)`

- Example: `data <- data.frame(category = c("A", "B", "A", "C", "B"))`

- Example: `distinct_values <- data %>% distinct(category)`

Using the `summary()` Function:

- The `summary()` function provides a summary of a variable, including thecount and unique values.

- Example: `data <- c("apple", "banana", "apple", "orange", "banana")`

- Example: `summary_data <- summary(data)`

Conclusion:

- Identifying categories in R is crucial for understanding the unique values and their frequencies in data.

- By utilizing functions like `unique()`, `table()`, `count()`, `levels()`, and `summary()`, you can efficiently identify categories in your data.

- Understanding category identification techniques expands your capabilities for data analysis, visualization, and modeling in R.

Defining and Ordering Levels

Introduction:

- Defining and ordering levels in R programming is essential when working withcategorical variables.

- R provides functions and techniques to define the levels of factors and specifytheir desired order.

- Let's explore how to define and order levels in R.

Defining Levels:

- Levels can be defined manually using the `factor()` function.

- Example: `data <- factor(c("High", "Low", "Medium"), levels = c("Low", "Medium", "High"))`

Accessing Levels:

- The `levels()` function is used to access and modify the levels of a factor.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `factor_levels <- levels(data)`

Modifying Levels:

- Levels can be modified by assigning new values to the `levels()` function.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `levels(data) <- c("Low", "Medium", "High")`

Ordering Levels:

- The order of levels in a factor can be specified using the `factor()` function with the `levels` argument.

- Example: `data <- factor(c("High", "Low", "Medium"), levels = c("Low", "Medium", "High"))`

Reordering Levels using `relevel()` (dplyr package):

- The `relevel()` function from the `dplyr` package is used to reorder levels.

- Example: `library(dplyr)`

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `reordered_data <- data %>% relevel("Medium")`

Ordering Levels using `forcats` Package:

- The `forcats` package provides additional functions to reorder and modifyfactor levels.

- Functions like `fct_reorder()`, `fct_relevel()`, and `fct_inorder()` are commonlyused.

- Example: `library(forcats)`

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `reordered_data <- fct_reorder(data, data, .fun = function(x)length(x))`

Ordering Levels using Numeric Codes:

- Levels can be ordered by assigning numeric codes to the levels using the `levels()` function.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `levels(data) <- c("Low", "Medium", "High")`

Summary of Levels:

- The `table()` function provides a summary of the levels and their frequency.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `level_summary <- table(data)`

Conclusion:

- Defining and ordering levels in R programming is crucial for categoricalvariables.

- By utilizing functions like `factor()`, `levels()`, `relevel()`, and packages like `dplyr` and `forcats`, you can define and order levels effectively.

- Understanding how to define and order levels expands your capabilities fordata analysis, visualization, and modeling in R.

Defining and Ordering Levels

Introduction:

- Defining and ordering levels in R programming is essential when working withcategorical variables.

- R provides functions and techniques to define the levels of factors and specifytheir desired order.

- Let's explore how to define and order levels in R.

Defining Levels:

- Levels can be defined manually using the `factor()` function.

- Example: `data <- factor(c("High", "Low", "Medium"), levels = c("Low", "Medium", "High"))`

Accessing Levels:

- The `levels()` function is used to access and modify the levels of a factor.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `factor_levels <- levels(data)`

Modifying Levels:

- Levels can be modified by assigning new values to the `levels()` function.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `levels(data) <- c("Low", "Medium", "High")`

Ordering Levels:

- The order of levels in a factor can be specified using the `factor()` function with the `levels` argument.

- Example: `data <- factor(c("High", "Low", "Medium"), levels = c("Low", "Medium", "High"))`

Reordering Levels using `relevel()` (dplyr package):

- The `relevel()` function from the `dplyr` package is used to reorder levels.

- Example: `library(dplyr)`

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `reordered_data <- data %>% relevel("Medium")`

Ordering Levels using `forcats` Package:

- The `forcats` package provides additional functions to reorder and modifyfactor levels.

- Functions like `fct_reorder()`, `fct_relevel()`, and `fct_inorder()` are commonlyused.

- Example: `library(forcats)`

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `reordered_data <- fct_reorder(data, data, .fun = function(x)length(x))`

Ordering Levels using Numeric Codes:

- Levels can be ordered by assigning numeric codes to the levels using the `levels()` function.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `levels(data) <- c("Low", "Medium", "High")`

Summary of Levels:

- The `table()` function provides a summary of the levels and their frequency.

- Example: `data <- factor(c("High", "Low", "Medium"))`

- Example: `level_summary <- table(data)`

Conclusion:

- Defining and ordering levels in R programming is crucial for categoricalvariables.

- By utilizing functions like `factor()`, `levels()`, `relevel()`, and packages like `dplyr` and `forcats`, you can define and order levels effectively.

- Understanding how to define and order levels expands your capabilities fordata analysis, visualization, and modeling in R.

Combining and Cutting Data

Introduction:

We will explore two essential operations in data manipulation: combining and cutting data using the R programming language. These operations are crucial for preprocessing and cleaning data before analysis. So, let's dive into the world of combining and cutting data in R!

Outline:

1. Combining Data
   a. Concatenating Data Frames
   b. Merging Data Frames
   c. Binding Rows and Columns
   d. Handling Missing Values

2. Cutting Data
   a. Subsetting Data Frames
   b. Filtering Data
   c. Removing Outliers
   d. Creating New Variables

I. Combining Data:

a. Concatenating Data Frames:

When working with multiple data frames, we often need to combine themvertically or horizontally. R provides various functions such as `rbind()` and `cbind()` to concatenate data frames accordingly.

b. Merging Data Frames:

Merging is used when we have common columns in different data frames and want to combine them based on those common columns. We will explore functions like `merge()` and `join()` to perform different types of merges, suchas inner, outer, left, and right joins.

c. Binding Rows and Columns:

Sometimes we have data in separate vectors or lists that we want to bindtogether into a data frame. We will cover functions like `bind_rows()` and `bind_cols()` to combine rows and columns, respectively.

d. Handling Missing Values:

Missing values are common in datasets. We will learn how to handle themduring data combination using functions like `na.omit()` and `complete.cases()`.

II. Cutting Data:

a. Subsetting Data Frames:

Subsetting allows us to extract specific rows and columns from a data frame based on certain conditions. We will explore various techniques such as using logical operators, numeric indexing, and column names for subsetting.

b. Filtering Data:

Filtering helps us extract rows from a data frame that satisfy specific conditions. We will use functions like `subset()` and the powerful `dplyr`package's `filter()` function to achieve this.

c. Removing Outliers:

Outliers can adversely affect our analysis. We will discuss techniques like z- score and IQR (interquartile range) to identify and remove outliers from ourdata.

d. Creating New Variables:

Sometimes we need to create new variables based on existing variables. Wewill learn how to use the `mutate()` function from the `dplyr` package to generate new variables with desired transformations.

Conclusion:

Combining and cutting data are fundamental operations in R programming. Bymastering these techniques, you can efficiently manipulate, preprocess, and analyze your data. We have covered concatenation, merging, binding, subsetting, filtering, outlier removal, and variable creation. With these skills, you are well-equipped to handle diverse datasets and derive meaningful insights.

UNIT - IV LISTS AND DATA FRAMES

In R programming, lists and data frames are two fundamentaldata structures used for organizing and manipulating data.

LIST:

A list is a collection of objects, which can be of differenttypes (e.g., vectors, matrices, data frames, or even other lists). Lists are created using the **list()** function.

Here's an example:

```
# Create a list
my_list <- list(name = "John", age = 30, scores = c(85, 90,95))


# Access elements in a list

print(my_list$name)                    # "John"
print(my_list$age)                     # 30
print(my_list$scores)                 # 85 90 95
```

Data Frames:

A data frame is a two-dimensional tabular datastructure with rows and columns. It is similar to a table or spreadsheet. Data frames can store different types of data (numeric, character, logical, etc.) and each column can have adifferent data type. Data frames can be created using the **data.frame()** function.

Here's an example:

```
# Create a data frame
student_data <- data.frame(
   name = c("John", "Alice", "Bob"),age = c(20, 22, 21),
   scores = c(85, 90, 95))
# Access elements in a data frame
```

```
print(student_data$name)                              # "John" "Alice" "Bob"
print(student_data$age)                               # 20 22 21
print(student_data$scores)                            # 85 90 95 #
```

Access specific elements in a data frame

```
print(student_data[1, 2])                             # "John"
print(student_data[2, "age"])                         # 22
```

These are just basic examples to illustrate the concepts of lists and data frames in R. Lists and data frames offer morefunctionality and can be manipulated using various functionsand operators in R.

LIST OF OBJECT

In R, lists can contain objects of different types, allowing you to store heterogeneous data within a single structure. Here's an example of creating a list of objects:

```
# Create a list of objects
my_list <- list(
  name = "John",age = 30,
  scores = c(85, 90, 95),is_student =
  TRUE, address = list(street = "123
  Main St", city = "New York",
  country = "USA")
)
```

# Access elements in the list

```
print(my_list$name)                                   # "John"
print(my_list$age)                                    # 30
print(my_list$scores)                                 # 85 90 95
print(my_list$is_student)                             # TRUE
print(my_list$address$street)                         # "123 Main St"
print(my_list$address$city)                           # "New York"
print(my_list$address$country)                        # "USA"
```

In the example above, the list **my_list** contains differentobjects such as character strings (**name**), numeric values (**age**), a numeric vector (**scores**), a logical value (**is_student**), and even another nested list (**address**).

Each element in the list can be accessed using the **$** operator followed by the name of the element. You can also access list elements using indexing with brackets **[]**. For example:

```
print(my_list[1]) # Access the first element of the list
print(my_list[3]) # Access the third element of the list
print(my_list[[3]]) # Access the content of the third element (scores vector)
print(my_list[[5]][2]) # Access the second element of thenested list (address$city)
```

Lists in R are highly flexible and allow you to store and manipulate different types of data structures together, making them a powerful tool for organizing and handling heterogeneous data.

COMPONENT ACCESS

In R programming, there are multiple ways to access components or elements within different data structures.Here are some commonly used methods for accessing components in R:

LIST:

Using the $ operator:

You can access components within a list by using the **$**operator followed by the name of the component.

For example:

my_list$name.

Using double brackets [[]]:

You can access components within a list by using double brackets and specifying the index or name of the component.For example: **my_list[[1]]** or **my_list[["name"]]**.

<u>Using single brackets []:</u>

You can access a subset of a list by using single bracketsand specifying the index or name of the component. For example: **my_list[1]** or **my_list["name"]**.

<u>Data Frames:</u>

Using the **$** operator: You can access columns within a data frame by using the **$** operator followed by the name of the column. For example: **my_df$column_name**. Using single brackets **[]**: You can access subsets of a data frame by using single brackets and specifying the row and column indices or names. For example: **my_df[1, 2]** or my_df["row_name", "column_name"]**.**

Matrices and Vectors:

<u>Using single brackets **[]**:</u>

You can access elements of a matrix or vector by usingsingle brackets and specifying the indices. For example: **my_matrix[1, 2]** or **my_vector[3]**.

<u>Factors:</u>

Using the **$** operator or single brackets **[]**: You can access levels of a factor by using the **$** operator or single brackets and specifying the indices or names of the levels. For example: **my_factor$level_name** or **my_factor[2]**. These are some of the common methods for accessing components in R programming.

NAMING

In R programming, naming refers to the process of assigningnames or labels to objects such as variables, functions, or data structures.

Naming is an important aspect of programming as it allowsyou to create meaningful and descriptive names that make your code easier to understand and maintain.

Here are some guidelines and best practices for naming in Rprogramming:

Use descriptive names:

Choose names that accurately describe the purpose or content of the object. This makes your code more self- explanatory and helps other programmers (including yourfuture self) understand the code.

Avoid using reserved keywords:

R has a set of reserved keywords that have special meanings in the language. Avoid using these reserved words as names for your objects to prevent conflicts and confusion.
Examples of reserved keywords include **if**, **for**, **function**,**TRUE**, **FALSE**, and **NULL**.

Follow a consistent naming style:

Consistency in naming improves the readability and maintainability of your code. You can choose from various naming conventions, such as camelCase (e.g., myVariable),snake_case (e.g., my_variable), or Period.Separated (e.g., my.variable). The key is to pick a style and stick to it throughout your codebase.

Use lowercase letters:

It is a common convention in R programming to use lowercase letters for variable and function names. This helpsdifferentiate them from reserved keywords, which are typically written in uppercase.

Be mindful of case sensitivity:

R is a case-sensitive language, so "myVariable" and "myvariable" would be considered as two different objects.Make sure to be consistent with the capitalization of your names to avoid confusion.

Avoid using abbreviations:

While it may be tempting to use abbreviations to makeyour code shorter, it can reduce code clarity. Use descriptive names instead of abbreviations whenever possible. However,keep the names concise and meaningful.

Use meaningful prefixes:

You can use prefixes to provide additional information about the type or purpose of an object.

For example, "num_"for numeric variables, "df_" for data frames, or "fn_" for function names.

Avoid starting names with a dot:

In R, names starting with a dot have a special meaning,typically used for internal functions or hidden objects. Avoidstarting your object names with a dot unless you have a specific reason to do so. By following these naming conventions and best practices,you can write clean, readable, and maintainable code in R programming

NESTING

Nesting in R programming refers to the practice of placingone data structure or function within another. It allows you to organize and manipulate data in a hierarchical manner, providing a way to handle complex structures and performadvanced operations.

There are different ways to nest data structures and functions in R, depending on your specific requirements. Hereare some examples:

Nesting vectors:

You can nest vectors within other vectors to create multi- dimensional structures. For example, you can create a matrixby nesting multiple vectors of the same length using the **matrix()** function.

EXAMPLE

```
vec1 <- c(1, 2, 3)

vec2 <- c(4, 5, 6)

mat <- matrix(c(vec1, vec2), nrow = 2)
```

Nesting lists:

Lists in R allow for arbitrary nesting of different data types andstructures. You can create a nested list by including other lists or data structures as elements. Here's an example

```
list1 <- list(a = 1, b = 2)
```

```
list2 <- list(c = 3, d = 4)
nested_list <- list(list1, list2)
```

Nesting functions:

R allows you to nest functions inside other functions. This technique is commonly used for defining helper functions or creatingmore complex behavior.

Here's a simple example:

```
calculate_average <- function(values) {
  sum_values <- sum(values)
  count_values <- length(values)
  calculate_mean <- function() {
     sum_values / count_values
  }
  calculate_mean()
}
result <- calculate_average(c(1, 2, 3, 4, 5))
```

In the above example, the **calculate_average** function nests the **calculate_mean** function within it to calculate the average of a givenset of values.

Nesting can be a powerful tool in R programming, enabling youto handle and manipulate data in more flexible and efficient ways. However, it's important to ensure that your code remains readable and maintainable, especially when dealing with deep or complexnesting structures

Data Frames

In R programming, a data frame is a two-dimensional data structurethat stores data in a tabular format, similar to a spreadsheet or a database table.

It is one of the most commonly used data structures in R andprovides a convenient way to manipulate and analyze data. Here's how you can create a data frame in R:

```
# Creating a data frame using vectors
```

```
name <- c("John", "Emily", "Michael")
age <- c(25, 30, 35)
city <- c("New York", "London", "Paris")
df <- data.frame(Name = name, Age = age, City = city)
```

In the example above, we created a data frame **df** by combining three vectors (**name**, **age**, and **city**) using the **data.frame()** function.

Each vector corresponds to a column in the data frame, and the **Name**, **Age**, and **City** are the column names.

You can also read data from external files, such as CSV or Excelfiles, into a data frame using functions like **read.csv()** or read_excel()**.**

Once you have a data frame, you can perform various operationson it. Here are some commonly used operations:

Accessing data:

You can access individual columns or subsets of data in a data frame using indexing. For example, **df$Name** gives you the values inthe "Name" column, and **df[2,]** gives you the second row of the data frame.

Manipulating data:

You can add or remove columns, rename columns, and modify values within a data frame. For example, you can add a new columnusing **df$new_column <- c(1, 2, 3)**.

Summarizing data:

You can calculate summary statistics, such as mean, median, orcount, for specific columns or subsets of data using functions like **mean()**, **median()**, or **table()**.

Filtering data:

You can filter rows based on specific conditions using logical operators. For example, **subset(df, Age > 30)** gives you the rowswhere the "Age" column is greater than 30. Merging data frames: You can combine multiple data frames basedon common variables using functions like **merge()** or **dplyr** packagefunctions such as **inner_join()**, **left_join()**, etc. These are just a few examples of what you can do with data frames in R. Data frames provide a versatile and efficient way to handle, analyze, and manipulate tabular data in R

Adding Data Columns and Combining Data Frames

In R programming, you can add data columns to an existing data frame and combine multiple data frames using various functions andtechniques. Here are some common methods:

Adding data columns:

To add a new column to an existing data frame, you can simplyassign a vector of values to a new column name within the data frame.

Here’s an example:

```
# Creating a data frame

df <- data.frame(Name = c("John", "Emily", "Michael"),Age = c(25, 30, 35))


# Adding a new column

df$City <- c("New York", "London", "Paris")
```

In the above example, we added a new column named "City" to theexisting data frame **df** by assigning a vector of city names to it. Combining data frames: There are multiple ways to combine data frames in R, depending on the desired outcome:

rbind():

Use the rbind() function to combine data frames vertically, stacking them on top of each other. The data frames should have thesame column names and order.

Here's an example:

```
# Creating two data frames


df1 <- data.frame(Name = c(“John”, “Emily”),Age = c(25, 30))
df2 <- data.frame(Name = c(“Michael”, “Jessica”),Age = c(35, 28))
```

```
# Combining data frames using rbind
Combined_df <- rbind(df1, df2)
```

cbind():

Use the **cbind()** function to combine data frames horizontally, merging them based on the rows. The data frames should have the same number of rows. Here's an example:

```
# Creating two data frames
df1 <- data.frame(Name = c("John", "Emily"), Age = c(25, 30))
df2 <- data.frame(City = c("New York", "London"), Country = c("USA", "UK"))

# Combining data frames using cbind

combined_df <-cbind(df1, df2)
```

merge():

Use the **merge()** function to combine data frames based on common variables. This function performs a database-style merge based on specified columns. Here's an example:

```
# Creating two data frames
df1 <- data.frame(ID = c(1, 2, 3), Name = c("John", "Emily", "Michael"))
df2 <- data.frame(ID = c(2, 3, 4), Age = c(25, 30, 35))
# Combining data frames using merge

combined_df <- merge(df1, df2, by = "ID")
```

In the above example, the merge() function merges the data frames df1 and df2 based on the common “ID” column.

These methods allow you to add new columns to existing data frames and combine multiple data frames based on different requirements in R programming.

Logical Record Subsets

In problem-solving using R, logical record subsets refer to the process of extracting or filtering specific records from a dataset basedon logical conditions. R provides various functions and operators thatallow you to create logical conditions and apply them to subsets of your data.

Here's an example of how you can use logical record subsets in R:

Suppose you have a dataset called “data” with multiple columns, including “age” and “income.” You want to extract a subset of records where the age is greater than 30 and the income is above acertain threshold.

```
# Create a logical condition
condition <- data$age > 30 & data$income > 50000

# Apply the condition to subset the data
subset_data <- data[condition, ]

# View the subsetted data
print(subset_data)
```

In this example, the logical condition data$age > 30 & data$income >50000 checks if the age is greater than 30 and the income is above 50,000. The subset() function is used to apply this condition to the dataset, resulting in a new subsetted dataset called subset_data.

Finally, the print() function is used to display the subsetted data. You can modify the logical condition based on your specific problemand data attributes. R provides a range of logical operators such as <, >, ==, !=, <=, and >=, along with logical functions like & (AND), | (OR), and ! (NOT), to create complex logical conditions.

By using logical record subsets, you can focus on specific subsets of your data that are relevant to your problem and performfurther analysis or computations as needed.

Some Special values

In R programming, there are several special values that have specific meanings. These values serve different purposes in data analysis, computations, and programming. Here are some commonly used special values in R:

NA:

NA represents missing or undefined values. It is used to indicate the absence of a value or when a value cannot be computed or determined. NA is often used in data cleaning, handling missing data,and computations that involve missing values.

NULL:

NULL is a special value that represents an object with no value or an empty object. It is often used to initialize variables or to removean object from memory.

Inf and -Inf:

Inf represents positive infinity, and -Inf represents negative infinity. These values are used to represent extremely large or smallnumbers in calculations, such as division by zero or overflow/underflow situations.

NaN:

NaN stands for “Not a Number” and is used to represent undefined or nonsensical results in mathematical operations. It typically occurs when performing operations that involve invalid or undefined values, such as taking the square root of a negative number.

TRUE and FALSE:

TRUE and FALSE are logical values in R representing boolean values. TRUE indicates a true condition or logical “1”, while FALSEindicates a false condition or logical “0”. These values are commonlyused in logical expressions, conditional statements, and boolean operations. Inf, -Inf, NaN, TRUE, and FALSE are all considered atomic vectorsin R.

These special values are important in data analysis and programmingas they help handle missing data, perform computations, handle exceptional cases, and work with logical conditions and boolean operations. Understanding their meanings and behaviors is crucial foreffective programming in R.

Infinity-NaN-NA-NULLAttributes

In R programming, there are certain attributes associated with special values such as Infinity, NaN, NA, and NULL. These attributes provide additional information or characteristics about these values. Here’s an explanation of the attributes associated witheach of these special values:

Infinity Attributes:

Inf:

It has the attribute “numeric” since it represents a numeric value.

-Inf: Similar to Inf, it also has the attribute “numeric.”

NaN Attributes:

<u>NaN:</u>

It has the attribute “numeric” like Inf and -Inf since it represents anumeric value. Additionally, it has the attribute “NA” to indicate thatit is not available or undefined.

NA Attributes:

<u>NA:</u>

It has the attribute “logical” since it represents a missing or undefined value. It can also have attributes such as “integer” or “numeric” when used in a specific context or data type.

NULLAttributes:

<u>NULL:</u>

It has the attribute “NULL” itself, indicating an empty or non- existent object. NULL does not have any other attributes associatedwith it.

These attributes provide information about the data type or nature ofthe special values. They can be helpful when performing operations,handling missing data, or dealing with special cases in R programming. For example, checking the attributes of a value can help determine its type or handle missing values appropriately.

You can access the attributes of a value using functions like typeof(),is.numeric(), is.logical(), and is.null(). Additionally, you can manipulate or modify the attributes using functions like attributes(), attr(), and setattr().

It's Important to understand these attributes and their implications toeffectively work with special values in R programming and handle data analysis tasks.

Attributes-Object-Class-Is-Dot

In R programming, attributes, object classes, and the “is.” Functionsare related concepts that are used to define and manipulate objects.

Let’s explore each of these concepts in more detail:

<u>Attributes:</u>

Attributes provide additional information or metadata associated with an object in R. They can include properties such asnames, dimensions, labels, or other user-defined information.

Attributes can be accessed and modified using functions like attributes(), attr(), and setattr(). Some common attributes include names, dimensions (for arrays and matrices), class, and levels (forfactors).

Object Classes:

Object classes define the type or structure of an object in R. Each object belongs to a specific class, which determines its behaviorand the available methods or functions that can be applied to it. You can check the class of an object using the class() function. R hasa wide range of predefined classes, such as numeric, character, logical, factor, data frame, and more.

Classes can also be user-defined through the use of object-orientedprogramming principles in R.

Functions:

R provides a set of functions starting with "is." That are used tocheck the class or type of an object. These functions return a logical value (TRUE or FALSE) indicating whether the object belongs to a specific class.

For example, is.numeric() checks if an object is of the numeric class, is.character() checks if it is a character, and so on. These functions are useful for conditional statements or for determining thetype of object before performing specific operations or computations.

Dot ("."):

The dot, or ".", is commonly used in R as a placeholder or shorthand for referring to an object or argument within a function or method. It is often used when you want to pass or refer to an object without explicitly mentioning its name. For example, the dot can be used in function arguments like … to indicate variable arguments, where multiple objects can be passed. It can also be used within function calls to specify the current object or the object being workedon.

Understanding attributes, object classes, and the "is." Functions isimportant for working with and manipulating objects in R.

These concepts allow you to define, identify, and handleobjects based on their class or type, and also provide additional information or metadata about the objects.

Object-Checking Functions-As-Dot Coercion Functions

In R programming, there are various object-checking functions andas-dot coercion functions that are used for data type checking and type conversion. Let's discuss these functions:

Object-Checking Functions:

is.numeric(x): Checks if the object "x" is of numeric class.
is.character(x): Checks if the object "x" is of character class.
is.logical(x): Checks if the object "x" is of logical class.
is.integer(x): Checks if the object "x" is of integer class.
is.factor(x): Checks if the object "x" is of factor class.
is.data.frame(x): Checks if the object "x" is a data frame.
is.list(x): Checks if the object "x" is a list.
is.null(x): Checks if the object "x" is NULL.
is.na(x): Checks if the object "x" contains missing values (NA).

As-Dot Coercion Functions:

as.numeric(x): Converts the object "x" to a numeric class.
as.character(x): Converts the object "x" to a character class.
as.logical(x): Converts the object "x" to a logical class.
as.integer(x): Converts the object "x" to an integer class.
as.factor(x): Converts the object "x" to a factor class.
as.data.frame(x): Converts the object "x" to a data frame.
as.list(x): Converts the object “x” to a list.

These functions allow you to check the type or class of an objectand perform type conversion or coercion as needed. They are useful for ensuring that objects have the desired data type for specific operations or for transforming data into a suitableformat for analysis. It’s important to note that coercion functions may introducechanges or loss of information if the conversion is not possible or if the original data cannot be accurately represented in the new data type.

Therefore, it’s advisable to carefully consider the implications of type conversion and handle data type mismatches appropriately inyour programming tasks.

UNIT V - BASIC PLOTTING

Using plot with Coordinate Vectors

In R programming, you can use the `plot()` function to create plots with coordinate vectors. The `plot()` function is a versatile function that can be used to create various types of plots, including scatter plots, line plots, bar plots, and more. Here's an example of how you can use the `plot()` function with coordinate vectors:

```
# Create x and y coordinate vectors

x <- c(1, 2, 3, 4, 5)

y <- c(2, 4, 6, 8, 10)


# Create a scatter plot

plot(x, y, type = "p", pch = 16, col = "blue", xlab = "X", ylab = "Y", main = "Scatter Plot")


# Add a line plot

lines(x, y, type = "l", col = "red")


# Add points with different color and shape

points(x, y + 2, pch = 17, col = "green")


# Add a legend

legend("topleft", legend = c("Scatter", "Line", "Points"), pch = c(16, NA, 17), col = c("blue",
"red", "green"), bty = "n")
```

In this example, we first create two coordinate vectors `x` and `y`. Then, we use the `plot()` function to create a scatter plot by specifying `type = "p"` (for points). The `pch` parameter sets the shape of the points, and `col` sets the color. We also provide labels for the x-axis and y-axis using `xlab` and `ylab`, and a title using `main`.

Next, we add a line plot using the `lines()` function, specifying `type = "l"` (for lines) and `col = "red"`.

We further add points with a different color and shape using the `points()` function, specifying `pch = 17` (for a different shape) and `col = "green"`.

Finally, we add a legend using the `legend()` function to explain the different elements of the plot. We position the legend in the top-left corner using `"topleft"`, and specify the legend labels, point shapes, and colors using the `legend`, `pch`, and `col` parameters, respectively.

You can customize the plot further by adjusting various parameters of the `plot()`, `lines()`, `points()`, and `legend()` functions according to your requirements.

Graphical Parameters

In R programming, graphical parameters are used to control the appearance of plots. These parameters allow you to customize various aspects of the plot, such as colors, line types, point shapes, axis labels, titles, legends, and more. Here are some commonly used graphical parameters in R:

1. `pch`: Specifies the symbol or shape of points in a plot. For example, `pch = 16` represents solid circles, `pch = 17` represents solid triangles, and so on.

2. `col`: Sets the color of points, lines, and text in a plot. You can specify colors using color names (e.g., `"red"`, `"blue"`), hexadecimal color codes (e.g., `"#FF0000"` for red), or numerical values (e.g., `2` for red, `4` for blue).

3. `lty`: Specifies the line type. Values include `"solid"` (default), `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, and more.

4. `lwd`: Sets the line width. A larger value represents a thicker line.

5. `xlim` and `ylim`: Sets the limits for the x-axis and y-axis, respectively. For example, `xlim = c(0, 10)` sets the x-axis limits from 0 to 10.

6. `xlab` and `ylab`: Specifies the labels for the x-axis and y-axis, respectively.

7. `main`: Sets the title of the plot.

8. `legend`: Allows you to add a legend to the plot. You can specify the position (`"topleft"`, `"topright"`, `"bottomleft"`, `"bottomright"`, etc.) and customize the legend labels, point shapes, and colors.

These parameters can be used in conjunction with various plotting functions in R, such as `plot()`, `lines()`, `points()`, and `legend()`, to customize the appearance of your plots.

For example, to create a scatter plot with red points and a blue line, you can use the following code:

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

y <- c(2, 4, 6, 8, 10)


plot(x, y, type = "p", pch = 16, col = "red", xlab = "X", ylab = "Y", main = "Scatter Plot")

lines(x, y, type = "l", col = "blue")
```

In this example, we set `pch = 16` to represent solid circles for points, `col = "red"` to set the color of points to red, and `col = "blue"` to set the color of the line to blue.

These are just a few examples of the graphical parameters available in R. You can explore more graphical parameters and their usage in the documentation of specific plotting functions or the overall R documentation.

Automatic Plot Types

In R programming, graphical parameters are used to control the appearance of plots. These parameters allow you to customize various aspects of the plot, such as colors, line types, point shapes, axis labels, titles, legends, and more. Here are some commonly used graphical parameters in R:

1. `pch`: Specifies the symbol or shape of points in a plot. For example, `pch = 16` represents solid circles, `pch = 17` represents solid triangles, and so on.

2. `col`: Sets the color of points, lines, and text in a plot. You can specify colors using color names (e.g., `"red"`, `"blue"`), hexadecimal color codes (e.g., `"#FF0000"` for red), or numerical values (e.g., `2` for red, `4` for blue).

3. `lty`: Specifies the line type. Values include `"solid"` (default), `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, and more.

4. `lwd`: Sets the line width. A larger value represents a thicker line.

5. `xlim` and `ylim`: Sets the limits for the x-axis and y-axis, respectively. For example, `xlim = c(0, 10)` sets the x-axis limits from 0 to 10.

6. `xlab` and `ylab`: Specifies the labels for the x-axis and y-axis, respectively.

7. `main`: Sets the title of the plot.

8. `legend`: Allows you to add a legend to the plot. You can specify the position (`"topleft"`, `"topright"`, `"bottomleft"`, `"bottomright"`, etc.) and customize the legend labels, point shapes, and colors.

These parameters can be used in conjunction with various plotting functions in R, such as `plot()`, `lines()`, `points()`, and `legend()`, to customize the appearance of your plots.

For example, to create a scatter plot with red points and a blue line, you can use the following code:

```
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 6, 8, 10)

plot(x, y, type = "p", pch = 16, col = "red", xlab = "X", ylab = "Y", main = "Scatter Plot")
lines(x, y, type = "l", col = "blue")
```

In this example, we set `pch = 16` to represent solid circles for points, `col = "red"` to set the color of points to red, and `col = "blue"` to set the color of the line to blue.

These are just a few examples of the graphical parameters available in R. You can explore more graphical parameters and their usage in the documentation of specific plotting functions or the overall R documentation.

Title and Axis Labels Color

In R programming, you can customize the color of the title and axis labels in plots by using graphical parameters. Here's how you can set the color for the title and axis labels:

```
# Create x and y coordinate vectors
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 6, 8, 10)

# Create a scatter plot with customized title and axis labels
plot(x, y, type = "p", pch = 16, col = "blue", xlab = "X", ylab = "Y", main = "Scatter Plot")
```

```
# Customize the color of the title and axis labels

title(main = "Scatter Plot", col.main = "red") # Set title color to red

title(xlab = "X", col.lab = "green") # Set x-axis label color to green

title(ylab = "Y", col.lab = "blue") # Set y-axis label color to blue
```

In this example, the `plot()` function is used to create a scatter plot with a title and axis labels. The `main` parameter sets the title of the plot, `xlab` sets the label for the x-axis, and `ylab` sets the label for the y-axis.

To customize the color of the title and axis labels, you can use the `title()` function. The `col.main` parameter is used to set the color of the title, and `col.lab` is used to set the color of the axis labels. In the example above, the title color is set to red (`col.main = "red"`), the x-axis label color is set to green (`col.lab = "green"`), and the y-axis label color is set to blue (`col.lab = "blue"`).

You can modify the color values to any valid color specification in R, such as color names (e.g., `"red"`, `"blue"`), hexadecimal color codes (e.g., `"#FF0000"` for red), or numerical values (e.g., `2` for red, `4` for blue).

By adjusting the `col.main` and `col.lab` parameters in the `title()` function, you can customize the color of the title and axis labels to suit your needs.

Line and Point Appearances

In R programming, you can customize the appearance of lines and points in plots using graphical parameters. Here are some common graphical parameters that you can use to modify the appearance of lines and points:

1. Line Appearance:

- `lty`: Specifies the line type. Options include `"solid"` (default), `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, etc.

- `lwd`: Sets the line width. A larger value represents a thicker line.

2. Point Appearance:

- `pch`: Specifies the symbol or shape of points. Common values include numbers from 0 to 25, where each number represents a different shape. For example, `pch = 16` represents solid circles, `pch = 17` represents solid triangles, and so on.

- `col`: Sets the color of the points. You can use color names (e.g., `"red"`, `"blue"`), hexadecimal color codes (e.g., `"#FF0000"` for red), or numerical values (e.g., `2` for red, `4` for blue).

Here's an example that demonstrates how to customize line and point appearances in a plot:

```
# Create x and y coordinate vectors

x <- c(1, 2, 3, 4, 5)

y <- c(2, 4, 6, 8, 10)
```

```
# Create a plot with customized line and point appearances
plot(x, y, type = "b", pch = 16, lty = 2, col = "blue", lwd = 2,
xlab = "X", ylab = "Y", main = "Customized Plot")
```

In this example, the `plot()` function is used to create a plot with customized line and point appearances. The `type = "b"` parameter specifies that both lines and points should be plotted.

To customize the line appearance, we set `lty = 2` to create a dashed line and `lwd = 2` to make the line thicker.

To customize the point appearance, we set `pch = 16` to represent solid circles for points and `col = "blue"` to set the color of the points to blue.

You can experiment with different values for `lty`, `lwd`, `pch`, and `col` to achieve the desired line and point appearances in your plots.

Additionally, you can use these graphical parameters with other plotting functions, such as `lines()` and `points()`, to customize the appearance of lines and points in more complex plots.

Plotting Region Limits

In R programming, you can set the limits for the plotting region by using the `xlim` and `ylim` parameters. These parameters allow you to define the range of values that will be displayed on the x-axis and y-axis of the plot, respectively. Here's an example of how to set the limits for the plotting region:

```
# Create x and y coordinate vectors

x <- c(1, 2, 3, 4, 5)

y <- c(2, 4, 6, 8, 10)


# Create a scatter plot with customized limits for the plotting region

plot(x, y, type = "p", pch = 16, col = "blue", xlab = "X", ylab = "Y", main = "Scatter Plot", xlim = c(0, 6), ylim = c(0, 12))
```

In this example, the `plot()` function is used to create a scatter plot with customized limits for the plotting region. The `xlim` parameter is set to `c(0, 6)`, which means that the x-axis will be displayed from 0 to 6. Similarly, the `ylim` parameter is set to `c(0, 12)`, which means that the y-axis will be displayed from 0 to 12.

By adjusting the values provided to `xlim` and `ylim`, you can define the desired range for the x-axis and y-axis in your plot. This allows you to focus on specific portions of the data or ensure that all relevant data points are displayed within the plotting region.

Note that if you don't specify the `xlim` and `ylim` parameters, R will automatically determine the appropriate limits based on the data provided.

Adding Points, Lines, and Text to an Existing Plot

In R programming, you can add additional points, lines, and text to an existing plot using various functions. Here are three commonly used functions for adding elements to a plot:

1. `points()`: The `points()` function is used to add points to an existing plot. You can specify the coordinates of the points, the shape of the points, the color, and other graphical parameters. For example:

```
# Create x and y coordinate vectors

x <- c(1, 2, 3, 4, 5)

y <- c(2, 4, 6, 8, 10)


# Create a scatter plot

plot(x, y, type = "p", pch = 16, col = "blue", xlab = "X", ylab = "Y", main = "Scatter Plot")


# Add additional points

points(x, y + 2, pch = 17, col = "red")
```

In this example, the `points()` function is used to add additional points to the existing scatter plot. The `x` and `y` coordinates of the additional points are provided, and the `pch` parameter specifies the shape of the points (solid triangles), while the `col` parameter sets their color to red.

2. `lines()`: The `lines()` function is used to add lines to an existing plot. You can specify the coordinates of the points that define the lines, the line type, color, and other graphical parameters. For example:

```
# Create x and y coordinate vectors

x <- c(1, 2, 3, 4, 5)

y <- c(2, 4, 6, 8, 10)
```

```
# Create a scatter plot

plot(x, y, type = "p", pch = 16, col = "blue", xlab = "X", ylab = "Y", main = "Scatter Plot")


# Add a line

lines(x, y, type = "l", col = "red", lwd = 2)
```

In this example, the `lines()` function is used to add a line to the existing scatter plot. The `x` and `y` coordinates are provided to define the line, and the `col` parameter sets the line color to red. The `lwd` parameter sets the line width to 2.

3. `text()`: The `text()` function is used to add text to an existing plot. You can specify the coordinates where the text should be placed, the text itself, the font size, color, and other graphical parameters. For example:

```
# Create x and y coordinate vectors

x <- c(1, 2, 3, 4, 5)

y <- c(2, 4, 6, 8, 10)


# Create a scatter plot

plot(x, y, type = "p", pch = 16, col = "blue", xlab = "X", ylab = "Y", main = "Scatter Plot")


# Add text

text(3, 7, "Text Example", col = "green", font = 2)
```

In this example, the `text()` function is used to add text to the existing scatter plot. The `x` and `y` coordinates specify the location of the text, and the text itself is specified as `"Text Example"`. The `col` parameter sets the color of the text to green, and the `font` parameter specifies the font style (2 represents bold).

You can use these functions, along with various graphical parameters, to add points, lines, and text to an existing plot, allowing you to enhance and annotate your visualizations in R.

ggplot2 Package

The ggplot2 package is a widely used and powerful data visualization package in R programming. It provides a flexible and grammar-based approach to creating plots, allowing you to easily build complex and customized visualizations. The package is based on the principles of the Grammar of Graphics, which provides a systematic way of describing the components of a plot.

To use the ggplot2 package, you first need to install it by running the following command in R:

```
install.packages("ggplot2")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```
library(ggplot2)
```

Here's a simple example that demonstrates the basic usage of ggplot2:

```
# Create a data frame

data<- data.frame(x = c(1, 2, 3, 4, 5), y = c(2, 4, 6, 8, 10))


# Create a scatter plot using ggplot2

ggplot(data, aes(x = x, y = y)) +

geom_point() +

labs(x = "X", y = "Y", title = "Scatter Plot")
```

In this example, we create a data frame `data` with x and y values. We then use the `ggplot()` function to initiate the plot, specifying the data frame and mapping the x and y variables using the `aes()` function. The `geom_point()` function adds the points to the plot. We also use the `labs()` function to set the labels for the x-axis, y-axis, and the plot title.

ggplot2 offers a wide range of geometric objects (`geom_`) and statistical transformations (`stat_`) that you can combine to create different types of plots. You can customize various aspects of the plot, such as colors, scales, facets, and themes, to create visually appealing and informative visualizations.

To learn more about ggplot2 and its capabilities, you can refer to the official ggplot2 documentation, which provides detailed explanations, examples, and tutorials: https://ggplot2.tidyverse.org/

Quick Plot with qplot

In the ggplot2 package, the `qplot()` function is a quick way to create basic plots by specifying the data and aesthetic mappings in a concise manner. It is a simplified version of the `ggplot()`

function that provides a fast and convenient way to generate common plot types. Here's an example of how to use `qplot()`:

```
# Create a data frame

data<- data.frame(x = c(1, 2, 3, 4, 5), y = c(2, 4, 6, 8, 10))


# Create a scatter plot using qplot

qplot(x, y, data = data, xlab = "X", ylab = "Y", main = "Scatter Plot")
```

In this example, we use `qplot()` to create a scatter plot. The `x` and `y` arguments specify the variables from the data frame to be plotted on the x-axis and y-axis, respectively. The `data` parameter specifies the data frame from which the variables are taken. Additional parameters such as `xlab`, `ylab`, and `main` are used to set the x-axis label, y-axis label, and plot title, respectively.

`qplot()` automatically selects appropriate default settings for the plot based on the input. However, keep in mind that `qplot()` is designed for simple and quick visualizations. If you need more flexibility and customization options, you may want to consider using the full `ggplot()` syntax.

It's worth noting that `qplot()` is considered a deprecated function in recent versions of ggplot2. The preferred approach is to use the `ggplot()` function along with the various `geom_` and `labs()` functions to create more complex and customized plots.

Setting Appearance Constants with Geoms

In R programming, when using the `ggplot2` package, you can set appearance constants with `geoms` to customize the visual properties of your plots. Geoms are the geometric objects that represent the visual elements of the plot, such as points, lines, bars, and more. Here are some commonly used geoms and their appearance constants:

1. `geom_point()`: This geom is used to add points to the plot. You can customize the appearance with the following aesthetics:

   - `color`: Sets the color of the points.

   - `size`: Sets the size of the points.

   - `shape`: Sets the shape of the points.

2. `geom_line()`: This geom is used to add lines to the plot. You can customize the appearance with the following aesthetics:

   - `color`: Sets the color of the lines.

   - `size`: Sets the size of the lines.

   - `linetype`: Sets the line type, such as `"solid"`, `"dashed"`, etc.

3. `geom_bar()` and `geom_col()`: These geoms are used for bar plots. You can customize the appearance with the following aesthetics:

   - `fill`: Sets the fill color of the bars.

   - `color`: Sets the color of the bar outlines.

   - `width`: Sets the width of the bars.

4. `geom_text()`: This geom is used to add text labels to the plot. You can customize the appearance with the following aesthetics:

- `label`: Specifies the text to display.
- `color`: Sets the color of the text.
- `size`: Sets the size of the text.
- `fontface`: Sets the font face of the text.

Here's an example that demonstrates how to set appearance constants with geoms in `ggplot2`:

```
library(ggplot2)

# Create a data frame
data<- data.frame(x = c(1, 2, 3, 4, 5), y = c(2, 4, 6, 8, 10))

# Create a scatter plot with customized appearance
ggplot(data, aes(x, y)) +
geom_point(color = "blue", size = 3, shape = 16) +
geom_line(color = "red", size = 1.5, linetype = "dashed") +
geom_bar(fill = "green", color = "black", width = 0.5) +
geom_text(aes(label = y), color = "purple", size = 4, fontface = "bold") +
labs(x = "X", y = "Y", title = "Customized Plot")
```

In this example, we use various geoms (`geom_point()`, `geom_line()`, `geom_bar()`, `geom_text()`) to customize different elements of the plot. We set appearance constants such as color, size, shape, fill, linetype, label, and fontface to modify the visual properties of the points, lines, bars, and text labels.

By combining geoms with different aesthetics, you can create visually appealing and informative plots with `ggplot2`. Remember that the specific aesthetics available for each geom may vary, so refer to the `ggplot2` documentation for more details on individual geoms and their corresponding aesthetics.

READING AND WRITING FILES

In R programming, you can read and write files using various functions and packages. Here are some commonly used functions for reading and writing files in R:

1. Reading Files:

   - `read.csv()`: Reads a CSV (Comma-Separated Values) file and returns a data frame.

   - `read.table()`: Reads a delimited text file and returns a data frame.

   - `read_excel()` (from the `readxl` package): Reads an Excel file and returns a data frame.

   - `readRDS()`: Reads an RDS (R Data Store) file and returns the stored R object.

2. Writing Files:

   - `write.csv()`: Writes a data frame to a CSV file.

   - `write.table()`: Writes a data frame to a delimited text file.

   - `write_excel_csv()` (from the `writexl` package): Writes a data frame to an Excel file in CSV format.

   - `saveRDS()`: Saves an R object to an RDS file.

Here are a few examples to illustrate the usage of these functions:

```
# Reading Files
data<- read.csv("data.csv")  # Read a CSV file
data<- read.table("data.txt", sep = "\t")  # Read a tab-delimited text file
data<- read_excel("data.xlsx", sheet = 1)  # Read an Excel file (requires the `readxl` package)
object<- readRDS("data.rds") # Read an RDS file

# Writing Files
write.csv(data, "output.csv", row.names = FALSE)  # Write a data frame to a CSV file
write.table(data, "output.txt", sep = "\t", row.names = FALSE) # Write a data frame to a tab-delimited text file
write_excel_csv(data, "output.xlsx")   # Write a data frame to an Excel file in CSV format (requires the `writexl` package)
saveRDS(object, "output.rds")  # Save an R object to an RDS file
```

In these examples, we demonstrate how to read and write different types of files. When reading files, you provide the file name (and path if necessary) as the first argument to the respective reading function. When writing files, you provide the data frame or R object as the first argument and specify the desired file name (and path) as the second argument.

Make sure to replace the file names and paths in the examples with your own file names and paths to match your specific file locations.

Note that there are additional packages and functions available in R for reading and writing files, depending on the file format and requirements of your data. You can explore specific packages and functions for reading and writing files, such as `readr`, `openxlsx`, `haven`, and more, based on your specific needs.

R-Ready Data Sets

In R programming, there are several built-in datasets and packages that provide preloaded datasets, which are ready to use for various analysis and visualization tasks. These datasets cover a wide range of domains, including statistics, economics, social sciences, biology, and more. Here are a few commonly used packages and their associated datasets:

1. `datasets` package: This is a built-in package that comes with R and provides a collection of example datasets. Some commonly used datasets from this package include:

   - `mtcars`: Information about various car models, including their features and performance.

   - `iris`: Measurements of various characteristics of iris flowers.

   - `airquality`: Daily air quality measurements in New York from May to September 1973.

To load a dataset from the `datasets` package, use the `data()` function followed by the dataset name. For example, to load the `mtcars` dataset:

```
data(mtcars)
```

2. `ggplot2` package: In addition to its powerful visualization capabilities, the `ggplot2` package also includes several datasets for practice and demonstration purposes. Some commonly used datasets from this package include:

   - `diamonds`: Prices and attributes of a large number of diamonds.

   - `mpg`: Fuel economy data for various car models.

- `gapminder`: Data on various indicators for countries over time.

To load a dataset from the `ggplot2` package, you need to load the package first using the `library()` function, and then the dataset is available for use. For example, to load the `diamonds` dataset:

```
library(ggplot2)

data(diamonds)
```

3. Other packages: There are many other R packages that provide datasets for specific domains. Here are a few examples:

- `MASS` package: It provides datasets related to the book "Modern Applied Statistics with S" by Venables and Ripley. Examples include `Boston` (housing data), `survey` (population survey data), etc.

- `dplyr` package: It includes datasets like `starwars`, `storms`, etc., which are useful for learning and practicing data manipulation operations using the `dplyr` package.

To load datasets from these packages, you need to install and load the respective packages using the `install.packages()` and `library()` functions. After loading the package, you can access the datasets using the `data()` function.

These are just a few examples of R-ready datasets available in various packages. By exploring these packages and their associated datasets, you can find suitable data for your analysis and familiarize yourself with real-world data examples.

Contributed Data Sets

In addition to the built-in datasets in R, there are numerous contributed datasets available in various packages and online resources. These datasets cover a wide range of topics and can be useful for practice, learning, and conducting analyses. Here are a few popular sources of contributed datasets in R programming:

1. `UCI Machine Learning Repository`: The UCI Machine Learning Repository is a popular source of datasets for machine learning and data analysis. Many datasets are available in R-ready format and can be accessed through the `mlbench` package. For example, the `PimaIndiansDiabetes` dataset contains information about diabetes in Pima Indian women.

```
install.packages("mlbench")

library(mlbench)

data(PimaIndiansDiabetes)
```

2. `TidyTuesday`: TidyTuesday is a weekly social data project in the R community that provides a curated dataset for practice and exploration. Each dataset is accompanied by a detailed description and a set of questions to guide analysis. You can access TidyTuesday datasets using the `tidytuesdayR` package:

```
install.packages("tidytuesdayR")

library(tidytuesdayR)

tt<- tt_load("2022-01-18")  # Load the TidyTuesday dataset for a specific date

data<- tt$data$name_of_dataset  # Access the specific dataset from the loaded TidyTuesday data
```

3. `fivethirtyeight` package: The `fivethirtyeight` package provides datasets used by the FiveThirtyEight data journalism website. These datasets cover various topics, including politics, sports, economics, and more. For example, you can load the `college_all_ages` dataset:

```
install.packages("fivethirtyeight")

library(fivethirtyeight)

data(college_all_ages)
```

4. `gapminder` package: The `gapminder` package provides datasets from the Gapminder project, which contains global development data. This package includes the well-known `gapminder` dataset that provides indicators for countries over time:

```
install.packages("gapminder")

library(gapminder)

data(gapminder)
```

5. Online resources: There are several websites and online platforms that provide datasets for various domains, such as Kaggle, data.gov, and Data World. You can download datasets from these sources and import them into R using the appropriate functions based on the file format.

Remember to install and load the relevant packages to access the contributed datasets. Additionally, always ensure that you comply with any licensing requirements or terms of use associated with the datasets you use.

Reading in External Data Files

In R programming, you can read in external data files using various functions and packages, depending on the file format. Here are some commonly used functions for reading different types of data files in R:

1. Reading CSV Files:

- `read.csv()`: Reads a comma-separated values (CSV) file and returns a data frame.
- `read.csv2()`: Reads a semicolon-separated values file with decimal comma.
- `read.delim()`: Reads a tab-delimited file.

```
data<- read.csv("data.csv")  # Read a CSV file

data<- read.csv2("data.csv")  # Read a semicolon-separated file with decimal comma

data<- read.delim("data.txt", sep = "\t")  # Read a tab-delimited file
```

2. Reading Text Files:

- `read.table()`: Reads a delimited text file and returns a data frame.
- `scan()`: Reads data from a text file and returns a vector or list.

```
data<- read.table("data.txt", sep = "\t")  # Read a tab-delimited text file

data<- scan("data.txt", what = "character")  # Read a text file into a character vector
```

3. Reading Excel Files:

- `readxl` package: Provides functions to read Excel files.

```
install.packages("readxl")

library(readxl)

data<- read_excel("data.xlsx", sheet = 1)  # Read an Excel file
```

4. Reading SPSS Files:

   - `haven` package: Provides functions to read and write SPSS files.

```
install.packages("haven")

library(haven)

data<- read_sav("data.sav")  # Read an SPSS file
```

5. Reading SAS Files:

   - `haven` package: Also supports reading and writing SAS files.

```
install.packages("haven")

library(haven)

data<- read_sas("data.sas7bdat")  # Read a SAS file
```

These are just a few examples of functions and packages for reading in external data files in R. There are additional packages available for reading specific file formats, such as `readr` for reading delimited files, `foreign` for reading various file formats, and more. Depending on your specific data file format, you can choose the appropriate function or package to import the data into R.

Writing Out Data Files and Plots

In R programming, you can write out data files and plots using various functions and packages. Here are some commonly used functions for writing data files and saving plots in R:

1. Writing Data Files:

   - `write.csv()`: Writes a data frame to a CSV (Comma-Separated Values) file.

   - `write.table()`: Writes a data frame to a delimited text file.

   - `write.xlsx()` (from the `writexl` package): Writes a data frame to an Excel file.

   - `write_sav()` (from the `haven` package): Writes a data frame to an SPSS file.

   - `write_sas()` (from the `haven` package): Writes a data frame to a SAS file.

```
write.csv(data, "output.csv", row.names = FALSE)  # Write a data frame to a CSV file
write.table(data, "output.txt", sep = "\t", row.names = FALSE) # Write a data frame to a tab-delimited text file
write.xlsx(data, "output.xlsx", sheetName = "Sheet1")  # Write a data frame to an Excel file
write_sav(data, "output.sav") # Write a data frame to an SPSS file
write_sas(data, "output.sas7bdat")  # Write a data frame to a SAS file
```

2. Saving Plots:

   - `ggsave()` (from the `ggplot2` package): Saves a ggplot2 plot to a file.

   - `png()`, `jpeg()`, `pdf()` (base R functions): Opens a graphics device to save plots in different file formats.

```
# Saving a ggplot2 plot

library(ggplot2)

plot<- ggplot(data, aes(x, y)) + geom_point()

ggsave("plot.png", plot, width = 6, height = 4, dpi = 300)  # Save the plot as a PNG file


# Saving a base R plot

png("plot.png", width = 800, height = 600, res = 120)  # Open a PNG graphics device

plot(x, y) # Create a plot

dev.off()  # Close the graphics device and save the plot as a PNG file
```

In the examples above, we demonstrate how to write out data files and save plots in different formats. You specify the data or plot object as the first argument, followed by the desired file name (and path if necessary). Additional arguments can be used to customize the output, such as specifying the separator for text files, excluding row names, setting dimensions for images, etc.

Make sure to replace the file names and paths in the examples with your own file names and paths to match your specific requirements.

Note that there are additional packages and functions available in R for writing data files and saving plots, depending on the file format and requirements of your data. You can explore specific packages and functions for writing data files and saving plots, such as `writexl`, `xlsx`, `Hmisc`, `Cairo`, and more, based on your specific needs.

Ad Hoc Object Read/Write Operations

In R programming, you can perform ad hoc object read/write operations using the `save()` and `load()` functions to save and load R objects to/from binary files. These functions allow you to store any R object, such as variables, data frames, lists, or even custom objects, in a binary format for later use. Here's how you can perform ad hoc object read/write operations:

1. Saving Objects:

   - `save()`: Saves one or more R objects to a binary file.

```
# Save objects to a binary file
save(object1, object2, file = "objects.Rdata")
```

In the above example, you can specify one or more objects separated by commas in the `save()` function, and then provide the desired file name (and path if necessary) using the `file` parameter. The objects will be saved in a binary format in the specified file.

2. Loading Objects:

   - `load()`: Loads objects from a binary file into the R environment.

```
# Load objects from a binary file
load("objects.Rdata")
```

In this example, you provide the file name (and path if necessary) to the `load()` function, and it will load the objects saved in the binary file into the R environment.

By using these functions, you can perform ad hoc read/write operations to store and retrieve R objects as needed. This can be useful for saving intermediate results, sharing data across sessions, or archiving important objects.

Note that when using `save()` and `load()`, the objects are saved in a binary format specific to R, and the file extension is typically `.Rdata` or `.rda`. You can specify the desired file name and extension when saving and loading objects.

Additionally, you can use the `saveRDS()` and `readRDS()` functions for saving and loading single R objects to/from RDS (R Data Store) files. The RDS format is a more compact alternative to saving entire workspace objects using `save()`. It is commonly used for saving single objects that can be easily loaded back into R.

```
# Save a single object to an RDS file

saveRDS(object, file = "object.rds")


# Load a single object from an RDS file

object<- readRDS("object.rds")
```

Using `save()` and `load()` or `saveRDS()` and `readRDS()` provides flexibility for ad hoc object read/write operations in R, allowing you to easily store and retrieve specific objects as needed.