Unit - 1 and 2

---------------

1.How do you download and install R on your computer? Briefly explain the steps.

To download and install R on your computer, follow these steps:

1. **Visit the R Website**: Go to the official R website at CRAN.
2. **Choose Your Operating System**: Select your OS (Windows, macOS, or Linux).
3. **Download R**: Click on the appropriate link to download the latest version of R for your OS.
4. **Run the Installer**: Open the downloaded file and start the installation process.
5. **Follow Installation Instructions**: Accept the license agreement and choose the default settings unless specific changes are needed.
6. **Select Installation Path**: Choose the folder where R will be installed (default is recommended).
7. **Complete the Installation**: Click "Install" and wait for the process to finish.
8. **Verify Installation**: Open R by searching for it in your system and running it.
9. **Install RStudio (Optional)**: Download and install RStudio from [posit.co](https://posit.co) for a better coding experience.
10. **Test R**: Run a simple command like print("Hello, R!") to check if it works properly.

2.Explain how help command in R is used in different Operating System.

The **help command** in R is used to get information about functions, packages, and features. It works the same way across different operating systems (Windows, macOS, and Linux), but how the help documentation is displayed may vary.

**Ways to Use the Help Command in R:**

1. **Basic Help for a Function**
   * Use help(function\_name) or ?function\_name
   * Example: help(mean) or ?mean
2. **Search for Topics**
   * Use help.search("keyword") to find related functions.
   * Example: help.search("regression")
3. **Help for a Package**
   * Use help(package="package\_name")
   * Example: help(package="ggplot2")
4. **Opening Documentation in a Browser**
   * Some OS may open help files in a web browser.
   * Example: help.start() opens the R documentation homepage.
5. **Using example() to See Function Usage**
   * Example: example(mean) shows how mean() is used.
6. **Using ??keyword for Fuzzy Search**
   * Example: ??plot searches for all related functions.

In **Windows**, help often opens in a separate window. In **macOS** and **Linux**, it may appear in the terminal or a browser, depending on the settings.

**How Help is Displayed in Different Operating Systems**

* **Windows**: Help pages usually open in a separate window in RGui or RStudio.
* **macOS**: Help opens in a separate viewer in RStudio or as an HTML page in a browser when using R from the terminal.
* **Linux**: Help is displayed in the terminal by default, but help.start() opens an HTML version in the web browser.

3.Explain with example different data operators in R.

In R, **operators** are symbols used to perform operations on values or variables. There are different types of operators in R:

**1. Arithmetic Operators (Used for basic math calculations)**

| **Operator** | **Meaning** | **Example** | **Output** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 10 - 4 | 6 |
| \* | Multiplication | 6 \* 2 | 12 |
| / | Division | 8 / 2 | 4 |
| %% | Modulus (Remainder) | 10 %% 3 | 1 |
| %/% | Integer Division | 10 %/% 3 | 3 |
| ^ or \*\* | Exponentiation | 2^3 or 2\*\*3 | 8 |

📌 **Example in R**

a <- 10

b <- 3

sum <- a + b # Addition

rem <- a %% b # Remainder

sum # Output: 13

rem # Output: 1

**2. Relational (Comparison) Operators (Used to compare values)**

| **Operator** | **Meaning** | **Example** | **Output** |
| --- | --- | --- | --- |
| == | Equal to | 5 == 5 | TRUE |
| != | Not equal to | 5 != 3 | TRUE |
| > | Greater than | 10 > 5 | TRUE |
| < | Less than | 4 < 6 | TRUE |
| >= | Greater than or equal to | 7 >= 7 | TRUE |
| <= | Less than or equal to | 2 <= 5 | TRUE |

📌 **Example**

x <- 10

y <- 5

result <- x > y # Checks if x is greater than y

result # Output: TRUE

**3. Logical Operators (Used for logical operations)**

| **Operator** | **Meaning** | **Example** | **Output** |
| --- | --- | --- | --- |
| & | AND (Both conditions must be TRUE) | (5 > 2) & (4 > 1) | TRUE |
| ` | ` | OR (At least one condition must be TRUE) | `(5 > 2) |
| ! | NOT (Reverses TRUE/FALSE) | !(5 > 3) | FALSE |

📌 **Example in R:**

a <- TRUE

b <- FALSE

result <- a & b # AND operation

result # Output: FALSE

**4. Assignment Operators (Used to assign values to variables)**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| <- | Assign value to variable | x <- 10 |
| -> | Assign value (right to left) | 10 -> x |
| <<- | Assign globally (inside functions) | x <<- 20 |

📌 **Example in R:**

x <- 100

y <<- 200 # Global assignment

x # Output: 100

y # Output: 200

**5. Special Operators**

| **Operator** | **Meaning** | **Example** | **Output** |
| --- | --- | --- | --- |
| %in% | Checks if an element is in a vector | 5 %in% c(1,2,3,5) | TRUE |
| : | Creates a sequence of numbers | 1:5 | 1 2 3 4 5 |

📌 **Example**

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

result <- 3 %in% nums # Check if 3 is in the list

result # Output: TRUE

**Conclusion**

These operators help in performing calculations, comparisons, logical operations, and data assignments in R. Understanding them is essential for writing efficient R scripts! 🚀

**4.Describe the multiple ways to read and write data into R.**

In R, we can **read** (import) and **write** (export) data from different file types. Below are some of the most common ways:

**1. Reading and Writing CSV Files**

CSV files (Comma-Separated Values) are commonly used for storing data in table format.

**Reading a CSV File**

data <- read.csv("file.csv") # Reads a CSV file

head(data) # Shows first few rows

**Writing a CSV File**

write.csv(data, "output.csv", row.names = FALSE) # Saves data as CSV

**2. Reading and Writing Excel Files**

To work with Excel files, install the readxl and writexl packages.

**Reading an Excel File**

library(readxl)

data <- read\_excel("file.xlsx", sheet = 1) # Reads first sheet

**Writing an Excel File**

library(writexl)

write\_xlsx(data, "output.xlsx") # Saves data as an Excel file

**3. Reading and Writing Text Files**

Text files contain data in simple text format.

**Reading a Text File**

data <- read.table("file.txt", header = TRUE, sep = "\t") # Reads a tab-separated file

**Writing a Text File**

write.table(data, "output.txt", sep = "\t", row.names = FALSE) # Saves as text file

**4. Reading and Writing R Data Files (.RData and .rds)**

R has its own data formats for storing and loading data quickly.

**Saving and Loading .RData File**

save(data, file = "data.RData") # Save multiple objects

load("data.RData") # Load the data

**Saving and Loading .rds File**

saveRDS(data, "data.rds") # Save a single object

data\_new <- readRDS("data.rds") # Load the object

**5. Reading and Writing Data from Databases**

To connect with databases like SQLite, use the DBI package.

**Reading Data from a Database**

library(DBI)

conn <- dbConnect(RSQLite::SQLite(), "database.sqlite")

data <- dbReadTable(conn, "table\_name") # Reads a table from database

dbDisconnect(conn) # Close connection

**Writing Data to a Database**

dbWriteTable(conn, "new\_table", data) # Saves data into database

**6. Reading and Writing JSON Files**

JSON files store data in a structured format, often used in web applications.

**Reading a JSON File**

library(jsonlite)

data <- fromJSON("file.json") # Reads a JSON file

**Writing a JSON File**

toJSON(data, "output.json", pretty = TRUE) # Saves as a JSON file

**7. Reading and Writing XML Files**

XML files are used for structured data storage.

**Reading an XML File**

library(XML)

data <- xmlParse("file.xml")

data <- xmlToList(data)

**Writing an XML File**

saveXML(data, file = "output.xml") # Saves XML data

**8. Reading and Writing HTML Tables**

Web pages sometimes contain tables that can be read in R.

**Reading an HTML Table from a Website**

library(rvest)

url <- "https://example.com"

data <- read\_html(url) %>% html\_table()

**Writing an HTML Table**

write.table(data, "output.html") # Saves as HTML file

**Conclusion**

There are **many ways** to read and write data in R, depending on the **file type** you are working with. The most common methods are:

* **CSV Files** → read.csv(), write.csv()
* **Excel Files** → read\_excel(), write\_xlsx()
* **Text Files** → read.table(), write.table()
* **R Data Files** → save(), load(), saveRDS(), readRDS()
* **Databases** → dbReadTable(), dbWriteTable()
* **JSON & XML Files** → fromJSON(), toJSON(), xmlParse()
* **Web Data (HTML Tables)** → read\_html(), html\_table()

**5.Explain the following commands of R: i) Combine Command, ii) History command**

**1. Combine Command (c())**

**The combine command in R is used to create vectors by combining multiple values. It is one of the most basic and essential functions in R.**

**Syntax:**

**c(value1, value2, value3, ...)**

**Example:**

**numbers <- c(10, 20, 30, 40) # Creates a numeric vector**

**print(numbers)**

**# Output: [1] 10 20 30 40**

**Uses of c() Command:**

**✅ Creating numeric, character, or logical vectors  
✅ Used in data manipulation  
✅ Helps in matrix and list creation**

**names <- c("John", "Emma", "Liam") # Character vector**

**print(names)**

**# Output: [1] "John" "Emma" "Liam"**

**values <- c(TRUE, FALSE, TRUE) # Logical vector**

**print(values)**

**# Output: [1] TRUE FALSE TRUE**

**2. History Command (history())**

**The history command is used to view previously executed commands in R. It helps users track their work and recall past commands.**

**Syntax:**

**history()**

**Example:**

**history() # Displays past R commands**

**Uses of history() Command:**

**✅ Helps in debugging and reviewing past commands  
✅ Saves command history for future reference  
✅ Useful for learning and re-executing previous operations**

**6.Explain any 4 types of Data Structure available in R. Give example.**

**In R, data structures are used to store and organize data efficiently. The four main types of data structures in R are:**

1. **Vector**
2. **Matrix**
3. **Data Frame**
4. **List**

**1. Vector (One-Dimensional Data)**

**A vector is the most basic data structure in R. It contains elements of the same data type (numeric, character, or logical).**

**Example:**

**num\_vector <- c(10, 20, 30, 40) # Numeric vector**

**char\_vector <- c("A", "B", "C") # Character vector**

**log\_vector <- c(TRUE, FALSE, TRUE) # Logical vector**

**Key Features:**

**✅ Stores elements of the same type  
✅ Created using c() function  
✅ Used for mathematical operations**

**2. Matrix (Two-Dimensional Data)**

**A matrix is a two-dimensional array where all elements must be of the same type (numeric, character, or logical).**

**Example:**

**mat <- matrix(1:9, nrow = 3, ncol = 3) # Creates a 3x3 matrix**

**print(mat)**

**Output:**

**[,1] [,2] [,3]**

**[1,] 1 4 7**

**[2,] 2 5 8**

**[3,] 3 6 9**

**Key Features:**

**✅ Organized in rows and columns  
✅ Created using matrix() function  
✅ Used in mathematical computations**

**3. Data Frame (Tabular Data)**

**A data frame is a collection of vectors of different data types (numeric, character, logical) organized in columns. It is similar to a spreadsheet.**

**Example:**

**df <- data.frame(Name = c("Alice", "Bob", "Charlie"),**

**Age = c(25, 30, 35),**

**Score = c(90, 85, 95))**

**print(df)**

**Output:**

**Name Age Score**

**1 Alice 25 90**

**2 Bob 30 85**

**3 Charlie 35 95**

**Key Features:**

**✅ Stores data in tabular format  
✅ Each column can have different data types  
✅ Used for statistical analysis and machine learning**

**4. List (Heterogeneous Data)**

**A list is a flexible data structure that can contain different types of data (vectors, matrices, data frames, or other lists).**

**Example:**

**my\_list <- list(Name = "Alice", Age = 25, Scores = c(90, 85, 95))**

**print(my\_list)**

**Key Features:**

**✅ Can store different data types  
✅ Created using list() function  
✅ Used for storing complex data like results from models**

**Comparison of Data Structures**

| **Data Structure** | **Type** | **Example** |
| --- | --- | --- |
| **Vector** | **One-dimensional, same type** | **c(1, 2, 3, 4, 5)** |
| **Matrix** | **Two-dimensional, same type** | **matrix(1:9, nrow=3, ncol=3)** |
| **Data Frame** | **Two-dimensional, different types** | **data.frame(Name, Age, Score)** |
| **List** | **Multi-dimensional, mixed types** | **list(Name="Alice", Age=25, Scores=c(90,85,95))** |

**Conclusion**

* **Vectors store one type of data in a single dimension.**
* **Matrices store data in a 2D format but with the same type.**
* **Data Frames are like tables with different data types in each column.**
* **Lists store different types of data together.**

**These structures are fundamental for data analysis and programming in R. 🚀**

**7.Write a short note about: i) as. character (), ii) as. factor (), iii) as. numeric (), iv) as. integer ()**

In R, we often need to **convert** data from one type to another. R provides functions for **type conversion**, which start with as. followed by the desired data type. Below are four important functions:

**i) as.character() – Converts Data to Character Type**

The as.character() function converts numbers, factors, or logical values into **character strings**.

**Example:**

num <- 100

char\_value <- as.character(num)

print(char\_value)

print(class(char\_value)) # Output: "character"

**Key Features:**

✅ Converts numeric, logical, or factor data into text format  
✅ Used for **labeling or text processing**  
✅ Output is always in **quotes (" ")**

**ii) as.factor() – Converts Data to Factor Type**

The as.factor() function converts a vector into a **factor**, which is useful for **categorical data** (e.g., Gender, Colors, etc.).

**Example:**

colors <- c("Red", "Blue", "Green", "Red")

factor\_colors <- as.factor(colors)

print(factor\_colors)

print(class(factor\_colors)) # Output: "factor"

**Key Features:**

✅ Converts text data into **categories (levels)**  
✅ Used in **statistical analysis and machine learning**  
✅ Helps optimize memory usage by storing unique values

**iii) as.numeric() – Converts Data to Numeric Type**

The as.numeric() function converts text, logical values, or factors into **numeric values**.

**Example:**

char\_num <- "123"

num\_value <- as.numeric(char\_num)

print(num\_value)

print(class(num\_value)) # Output: "numeric"

**Key Features:**

✅ Converts characters, factors, or logical values to numbers  
✅ Returns NA if conversion is not possible (e.g., as.numeric("abc"))  
✅ Used in **mathematical operations and calculations**

**iv) as.integer() – Converts Data to Integer Type**

The as.integer() function converts numbers, characters, or logical values into **integers** (whole numbers).

**Example:**

num <- 10.7

int\_value <- as.integer(num)

print(int\_value)

print(class(int\_value)) # Output: "integer"

**Output:**

[1] 10 # Decimal part is removed

**Key Features:**

✅ Converts values into whole numbers  
✅ **Removes decimal places** (does not round, just truncates)  
✅ Used for **indexing, counting, and discrete values**

**8.How can you import and export data in R? Explain with examples for CSV, Excel, and databases**

In R, we often need to **import data** (bring data into R) and **export data** (save data from R). The most common formats for importing and exporting are:

1. **CSV Files**
2. **Excel Files**
3. **Databases**

**1. Importing and Exporting CSV Files**

**Import CSV (Comma-Separated Values) File**

We use the read.csv() function to import CSV files.

**Example:**

data <- read.csv("data.csv") # Imports CSV file

print(head(data)) # Displays first few rows

**Export CSV File**

To save a dataset as a CSV file, we use the write.csv() function.

**Example:**

write.csv(data, "output.csv", row.names = FALSE) # Exports data to CSV

**Key Features of CSV:**

✅ **Simple format**, easy to read and write  
✅ Used for **data exchange** between applications  
✅ Can be opened in **Excel, Google Sheets, and R**

**2. Importing and Exporting Excel Files**

**Import Excel File**

To work with Excel files, we use the readxl package (read\_excel() function).

**Example:**

library(readxl)

excel\_data <- read\_excel("data.xlsx", sheet = 1) # Reads the first sheet

print(head(excel\_data))

**Export Excel File**

To save a dataset as an Excel file, we use the writexl package (write\_xlsx() function).

**Example:**

library(writexl)

write\_xlsx(data, "output.xlsx") # Saves the data as an Excel file

**Key Features of Excel Files:**

✅ Supports **multiple sheets**  
✅ Stores **formatted data (colors, formulas, etc.)**  
✅ Widely used in **business and finance**

**3. Importing and Exporting Data from Databases**

**Import Data from a Database**

To connect to a database (e.g., MySQL, PostgreSQL), we use the DBI and RMySQL packages.

**Example (MySQL Database):**

library(DBI)

# Connect to MySQL database

conn <- dbConnect(RMySQL::MySQL(),

dbname = "my\_database",

host = "localhost",

user = "root",

password = "password")

# Import data from a table

db\_data <- dbGetQuery(conn, "SELECT \* FROM customers")

print(head(db\_data))

**Export Data to a Database**

We can export a dataset to a database using dbWriteTable().

**Example:**

dbWriteTable(conn, "new\_table", data) # Saves data to MySQL table

dbDisconnect(conn) # Close connection

**Key Features of Databases:**

✅ **Stores large datasets efficiently**  
✅ Used for **big data and web applications**  
✅ Allows **complex queries and data filtering**

**9.Define Packages. How do you Load and Install packages in R? Explain briefly.**

A **package** in R is a collection of functions, data, and documentation that extends R’s capabilities. Packages help in performing **data analysis, visualization, machine learning, and more**.

**Examples of Popular R Packages:**

✅ ggplot2 – For data visualization  
✅ dplyr – For data manipulation  
✅ readxl – For reading Excel files  
✅ caret – For machine learning

**How to Install Packages in R?**

Before using a package, we must install it using the install.packages() function.

**Example:**

install.packages("ggplot2") # Installs ggplot2 package

🔹 This downloads the package from CRAN (Comprehensive R Archive Network).

**How to Load a Package in R?**

After installation, we need to load the package using the library() function.

**Example:**

library(ggplot2) # Loads ggplot2 package

🔹 Now, we can use functions from the ggplot2 package.

**How to Check Installed Packages?**

To see a list of installed packages, use:

installed.packages()

**How to Remove a Package?**

If you no longer need a package, you can remove it using:

remove.packages("ggplot2")

**Comparison of Package Functions**

| **Action** | **Function** | **Example** |
| --- | --- | --- |
| **Install a package** | install.packages("package\_name") | install.packages("dplyr") |
| **Load a package** | library(package\_name) | library(dplyr) |
| **Check installed packages** | installed.packages() | installed.packages() |
| **Remove a package** | remove.packages("package\_name") | remove.packages("dplyr") |

**Conclusion**

* **Packages** provide extra functions and tools in R.
* **Installing (install.packages())** is needed only once.
* **Loading (library())** is required in every session.
* Managing packages helps improve **efficiency** in data analysis. ✅

**10.Explain different Control statements used in R with an example.**

Control statements in R help in **decision-making** and **looping**, allowing us to execute code conditionally or repeatedly. The main types of control statements in R are:

**1. Conditional Statements (if, if-else, if-else if)**

Used for making decisions based on conditions.

**Example: if-else Statement**

num <- 10

if (num > 0) {

print("Positive number")

} else {

print("Negative number or zero")

}

**Output:**

[1] "Positive number"

✅ **Executes different code blocks based on conditions.**

**2. Looping Statements (for, while, repeat)**

Used for executing code multiple times.

**Example: for Loop**

for (i in 1:5) {

print(i)

}

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

✅ **Repeats the code for a fixed number of iterations.**

**Example: while Loop**

x <- 1

while (x <= 3) {

print(x)

x <- x + 1

}

**Output:**

[1] 1

[1] 2

[1] 3

✅ **Repeats the code until the condition becomes FALSE.**

**Example: repeat Loop**

x <- 1

repeat {

print(x)

x <- x + 1

if (x > 3) break # Exit loop

}

**Output:**

[1] 1

[1] 2

[1] 3

✅ **Runs indefinitely unless explicitly stopped using break.**

**3. Jump Statements (break, next, return)**

Used to **control the flow** of loops.

**Example: break Statement**

Stops a loop when a condition is met.

for (i in 1:10) {

if (i == 5) break # Stops loop when i = 5

print(i)

}

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

✅ **Used to exit a loop early.**

**Example: next Statement**

Skips the current iteration and moves to the next.

for (i in 1:5) {

if (i == 3) next # Skip 3

print(i)

}

**Output:**

[1] 1

[1] 2

[1] 4

[1] 5

✅ **Used to skip specific values in loops.**

**Comparison of Control Statements**

| **Type** | **Statement** | **Function** | **Example** |
| --- | --- | --- | --- |
| **Conditional** | if-else | Decision making | if (x > 0) print("Positive") |
| **Looping** | for | Fixed repetitions | for (i in 1:5) print(i) |
|  | while | Runs until condition is FALSE | while (x < 5) x <- x + 1 |
|  | repeat | Infinite loop (needs break) | repeat {if (x > 5) break} |
| **Jumping** | break | Exits a loop | if (i == 5) break |
|  | next | Skips an iteration | if (i == 3) next |

**Conclusion**

* **Conditional statements (if-else)** control decision-making.
* **Looping statements (for, while, repeat)** help in iteration.
* **Jump statements (break, next)** manage loop execution.

These control statements are **essential** for writing efficient R programs. ✅

**11.What are the different methods to run the R program? Explain.**

**1. Using R Console**

✅ **Best for:** Running single commands quickly.

* Type R commands directly in the **R Console** and press **Enter**.
* The output appears immediately.

**Example:**

print("Hello, R!")

**Output:**

[1] "Hello, R!"

✔ **Good for testing small code snippets.**

**2. Using R Script (.R File)**

✅ **Best for:** Writing and running multiple lines of code.

* Create a file with a .R extension.
* Write multiple lines of code.
* Click **Run** or use source("script.R") to execute the script.

**Example:**

x <- 5

y <- 10

print(x + y)

✔ **Best for writing long programs.**

**3. Using R Markdown (.Rmd)**

✅ **Best for:** Creating reports with R code and text.

* Used for **data analysis reports and presentations**.
* Allows mixing R code with text.

✔ **Good for sharing analysis with others.**

**4. Using Command Line (Batch Mode)**

✅ **Best for:** Running R scripts without opening RStudio.

* Open the **Command Prompt (Windows)** or **Terminal (Mac/Linux)**.
* Type:

Rscript script.R

✔ **Used for automation and scheduled tasks.**

**5. Using Jupyter Notebook**

✅ **Best for:** Running R step by step in a notebook format.

* Works like Python’s **Jupyter Notebook**.
* Install IRKernel to use R in Jupyter.

✔ **Best for data science and visualization.**

**Comparison of Methods**

| **Method** | **Best For** |
| --- | --- |
| **R Console** | Quick testing |
| **R Script (.R)** | Writing complete programs |
| **R Markdown (.Rmd)** | Reports and presentations |
| **Command Line** | Running scripts without opening R |
| **Jupyter Notebook** | Interactive data analysis |

**Conclusion**

* **Use R Console** for quick tests.
* **Use R Script** for full programs.
* **Use R Markdown** for reports.
* **Use Command Line** for automation.
* **Use Jupyter Notebook** for step-by-step execution.

These methods help **run R efficiently** based on different needs. ✅

Bottom of Form

**12.How do you use R for simple math? Explain with an example.**

R can be used as a **calculator** to perform simple math operations like **addition, subtraction, multiplication, division, exponentiation, and more**.

**1. Basic Arithmetic Operations**

| **Operator** | **Operation** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 10 - 4 | 6 |
| \* | Multiplication | 6 \* 2 | 12 |
| / | Division | 15 / 3 | 5 |
| ^ or \*\* | Exponentiation | 2^3 | 8 |
| %% | Modulus (Remainder) | 10 %% 3 | 1 |
| %/% | Integer Division | 10 %/% 3 | 3 |

**2. Example of Simple Math in R**

# Performing basic math operations

a <- 10

b <- 5

sum\_result <- a + b # Addition

diff\_result <- a - b # Subtraction

prod\_result <- a \* b # Multiplication

div\_result <- a / b # Division

exp\_result <- a^2 # Exponentiation

# Printing results

print(sum\_result) # Output: 15

print(diff\_result) # Output: 5

print(prod\_result) # Output: 50

print(div\_result) # Output: 2

print(exp\_result) # Output: 100

✔ **R performs calculations just like a calculator!**

**3. Using R for Advanced Math Functions**

| **Function** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| sqrt(x) | Square root | sqrt(16) | 4 |
| abs(x) | Absolute value | abs(-7) | 7 |
| log(x) | Natural logarithm | log(10) | 2.30 |
| exp(x) | Exponential function | exp(2) | 7.39 |
| round(x, n) | Rounds to n decimal places | round(3.567, 2) | 3.57 |
| ceiling(x) | Rounds up | ceiling(4.3) | 5 |
| floor(x) | Rounds down | floor(4.7) | 4 |

**Example:**

print(sqrt(25)) # Output: 5

print(abs(-10)) # Output: 10

print(round(3.456, 1)) # Output: 3.5

✔ **R has built-in functions for mathematical calculations.**

**4. Using R for Trigonometric Calculations**

| **Function** | **Description** | **Example** |
| --- | --- | --- |
| sin(x) | Sine | sin(pi/2) |
| cos(x) | Cosine | cos(0) |
| tan(x) | Tangent | tan(pi/4) |

**Example:**

print(sin(pi/2)) # Output: 1

print(cos(0)) # Output: 1

print(tan(pi/4)) # Output: 1

✔ **Useful for scientific and engineering calculations.**

**Conclusion**

* **R can perform all types of math operations.**
* **Basic operators (+, -, \*, /, ^)** handle simple calculations.
* **Functions (sqrt(), log(), round())** help with advanced math.
* **Trigonometric functions (sin(), cos(), tan())** are available for scientific computations

13.Describe the Looping statements used in R with example.

Looping statements in R **repeat a block of code multiple times**, which helps in automating repetitive tasks. The main looping statements in R are:

**1. for Loop**

✅ **Best for:** Running a loop a fixed number of times.

**Syntax:**

for (variable in sequence) {

# Code to execute

}

**Example:** Print numbers from 1 to 5.

for (i in 1:5) {

print(i)

}

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

✔ **Runs the loop 5 times, increasing i from 1 to 5.**

**2. while Loop**

✅ **Best for:** Running a loop **until** a condition becomes FALSE.

**Syntax:**

while (condition) {

# Code to execute

}

**Example:** Print numbers from 1 to 5.

x <- 1

while (x <= 5) {

print(x)

x <- x + 1

}

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

✔ **Loop stops when x > 5.**

**3. repeat Loop**

✅ **Best for:** Running an **infinite loop** (must use break to stop).

**Syntax:**

repeat {

# Code to execute

if (condition) {

break # Stops the loop

}

}

**Example:** Print numbers from 1 to 3 using repeat.

x <- 1

repeat {

print(x)

x <- x + 1

if (x > 3) break

}

**Output:**

[1] 1

[1] 2

[1] 3

✔ **Uses break to stop the infinite loop.**

**4. Jump Statements (break & next)**

Jump statements help control loops.

✅ **break** – Exits the loop.  
✅ **next** – Skips current iteration and moves to the next one.

**Example: Using break in a for loop**

for (i in 1:5) {

if (i == 3) break # Stop loop when i = 3

print(i)

}

**Output:**

[1] 1

[1] 2

✔ **Loop stops at 3.**

**Example: Using next in a for loop**

for (i in 1:5) {

if (i == 3) next # Skip 3

print(i)

}

**Output:**

[1] 1

[1] 2

[1] 4

[1] 5

✔ **Skips i = 3, but continues with other numbers.**

**Comparison of Looping Statements**

| **Loop Type** | **Use Case** | **Stops When?** | **Example** |
| --- | --- | --- | --- |
| **for Loop** | Known number of repetitions | Runs for fixed iterations | for (i in 1:5) print(i) |
| **while Loop** | Runs until condition is FALSE | Condition becomes FALSE | while (x <= 5) print(x) |
| **repeat Loop** | Infinite loop unless stopped | break statement is used | repeat {if (x > 3) break} |
| **break** | Stops a loop immediately | Condition inside loop is met | if (i == 3) break |
| **next** | Skips an iteration and continues | Condition inside loop is met | if (i == 3) next |

**Conclusion**

* **for Loop** – Used for fixed repetitions.
* **while Loop** – Runs until a condition is FALSE.
* **repeat Loop** – Runs infinitely unless break is used.
* **break & next** – Control loop flow

**14.Define R studio and explain its characteristics.**

**1. Definition of RStudio**

✅ **RStudio** is an **Integrated Development Environment (IDE)** for **R programming**. It provides a **user-friendly interface** for writing, running, and debugging R code.

👉 It is available in two versions:

* **RStudio Desktop** (runs on your computer).
* **RStudio Server** (runs on a remote server and accessed via a web browser).

**2. Characteristics of RStudio**

🔹 **1. Four-Pane Layout** – RStudio has four main sections:

* **Source Pane**: Write and edit scripts.
* **Console Pane**: Execute R commands.
* **Environment/History Pane**: Shows variables and past commands.
* **Files/Plots/Packages/Help Pane**: Manages files, plots, and installed packages.

🔹 **2. Script Editor** – Allows writing and saving .R scripts for reuse.

🔹 **3. Interactive Console** – Runs commands instantly and displays results.

🔹 **4. Data Visualization** – Supports **graphs and plots** using libraries like ggplot2.

🔹 **5. Package Management** – Easy installation and loading of R **packages**.

🔹 **6. Debugging Tools** – Helps in identifying errors in the code.

🔹 **7. Markdown Support** – Allows creating reports using **R Markdown (.Rmd)**.

🔹 **8. Code Completion** – Suggests functions and variables while typing.

🔹 **9. Cross-Platform** – Works on **Windows, macOS, and Linux**.

🔹 **10. Integration with Git** – Supports version control using **Git and GitHub**.

**3. Why Use RStudio?**

✔ **Easier coding** with an organized interface.  
✔ **Faster debugging** with built-in tools.  
✔ **Better visualization** for data analysis.  
✔ **Supports automation** and reproducible research.

**15.Explain the following: i) Arithmetic operators, ii) Assignment operators, iii) Comparison operators, iv) Logical operators**

**(repeated question)**

16.How can you perform basic statistical operations (mean, median, standard deviation, correlation) in R? Explain with examples.

R provides built-in functions to perform **basic statistical operations** like **mean, median, standard deviation, and correlation**.

**1. Mean (Average)**

✅ **Definition:** The sum of all values divided by the total number of values.  
✅ **Function in R:** mean()

**Example:**

numbers <- c(10, 20, 30, 40, 50)

mean\_value <- mean(numbers)

print(mean\_value)

**Output:**

[1] 30

✔ **The average of (10, 20, 30, 40, 50) is 30.**

**2. Median**

✅ **Definition:** The middle value of a sorted dataset.  
✅ **Function in R:** median()

**Example:**

numbers <- c(10, 20, 30, 40, 50)

median\_value <- median(numbers)

print(median\_value)

**Output:**

[1] 30

✔ **Since 30 is the middle value, it is the median.**

**3. Standard Deviation**

✅ **Definition:** Measures how spread out the values are from the mean.  
✅ **Function in R:** sd()

**Example:**

numbers <- c(10, 20, 30, 40, 50)

sd\_value <- sd(numbers)

print(sd\_value)

**Output:**

[1] 15.81

✔ **A higher standard deviation means more variation in data.**

**4. Correlation**

✅ **Definition:** Measures the relationship between two variables.  
✅ **Function in R:** cor()  
✅ **Values range from -1 to 1:**

* **1** → Perfect positive correlation
* **0** → No correlation
* **-1** → Perfect negative correlation

**Example:**

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

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

correlation\_value <- cor(x, y)

print(correlation\_value)

**Output:**

[1] 1

✔ **Since y = 2x, the correlation is 1 (perfect positive correlation).**

**Conclusion**

| **Statistic** | **Function in R** | **Purpose** |
| --- | --- | --- |
| **Mean** | mean() | Finds the average value |
| **Median** | median() | Finds the middle value |
| **Standard Deviation** | sd() | Measures data spread |
| **Correlation** | cor() | Measures relationship between variables |

**17.How do you create vector objects? What are the different operations that can be used with vector object? Give an example.**

**What is a Vector in R?**

A **vector** is a basic data structure in R that holds **multiple values** of the **same data type** (numeric, character, logical, etc.).

**2. How to Create a Vector?**

✅ **Use c() function** to create a vector.

**Example: Creating Different Types of Vectors**

# Numeric vector

num\_vector <- c(10, 20, 30, 40, 50)

# Character vector

char\_vector <- c("Apple", "Banana", "Cherry")

# Logical vector

log\_vector <- c(TRUE, FALSE, TRUE)

✔ **Vectors store multiple values in a single object.**

**3. Operations on Vectors**

You can perform **mathematical and logical operations** on vectors.

**i) Arithmetic Operations on Vectors**

✅ **Operations apply to each element of the vector.**

x <- c(2, 4, 6)

y <- c(1, 2, 3)

# Addition

print(x + y) # [1] 3 6 9

# Subtraction

print(x - y) # [1] 1 2 3

# Multiplication

print(x \* y) # [1] 2 8 18

# Division

print(x / y) # [1] 2 2 2

✔ **Element-wise calculations are performed.**

**ii) Logical Operations on Vectors**

✅ **Compares elements and returns TRUE or FALSE.**

a <- c(5, 10, 15)

b <- c(10, 10, 10)

print(a > b) # [1] FALSE FALSE TRUE

print(a == b) # [1] FALSE TRUE FALSE

✔ **Checks conditions for each element.**

**iii) Accessing Vector Elements (Indexing)**

✅ **Use square brackets [ ] to access elements.**

v <- c(10, 20, 30, 40, 50)

print(v[1]) # First element → 10

print(v[3]) # Third element → 30

print(v[2:4]) # Second to fourth elements → 20 30 40

✔ **Indexes start from 1 in R.**

**iv) Vector Functions**

| **Function** | **Purpose** | **Example** |
| --- | --- | --- |
| length() | Count elements | length(v) → 5 |
| sum() | Sum of elements | sum(v) → 150 |
| min() | Smallest value | min(v) → 10 |
| max() | Largest value | max(v) → 50 |
| sort() | Sorts vector | sort(v) → 10 20 30 40 50 |

✔ **R has built-in functions for vector operations.**

**4. Conclusion**

* **Vectors store multiple values of the same type.**
* **Operations like addition, comparison, and indexing work element-wise.**
* **Useful functions (sum(), min(), sort()) help in data analysi**

**18.Define Data type and Data structures in R. Write the steps to convert number data to text data and vice versa.**

**What is a Data Type in R?**

✅ A **data type** defines the kind of values a variable can store.

**Common Data Types in R**

| **Data Type** | **Description** | **Example** |
| --- | --- | --- |
| **Numeric** | Numbers (integers, decimals) | x <- 10.5 |
| **Integer** | Whole numbers | y <- as.integer(5) |
| **Character** | Text values | z <- "Hello" |
| **Logical** | Boolean values (TRUE or FALSE) | t <- TRUE |
| **Factor** | Categorical data | f <- factor(c("Male", "Female")) |

**2. What is a Data Structure in R?**

✅ A **data structure** organizes and stores multiple values efficiently.

**Common Data Structures in R**

| **Data Structure** | **Description** | **Example** |
| --- | --- | --- |
| **Vector** | 1D collection of the same type | v <- c(1, 2, 3) |
| **Matrix** | 2D collection of the same type | m <- matrix(1:6, nrow=2, ncol=3) |
| **Data Frame** | 2D collection of mixed types | df <- data.frame(Name=c("A", "B"), Age=c(25,30)) |
| **List** | Stores different data types | l <- list(1, "Text", TRUE) |

**3. Converting Number Data to Text Data (Numeric → Character)**

✅ Use the as.character() function.

**Example:**

num <- 100

text <- as.character(num)

print(text) # "100"

print(class(text)) # "character"

✔ **Converts 100 (numeric) to "100" (character).**

**4. Converting Text Data to Number Data (Character → Numeric)**

✅ Use the as.numeric() function.

**Example:**

text <- "200"

num <- as.numeric(text)

print(num) # 200

print(class(num)) # "numeric"

✔ **Converts "200" (character) to 200 (numeric).**

**Conclusion**

* **Data types** define the kind of values stored (e.g., numeric, character).
* **Data structures** organize multiple values (e.g., vectors, data frames).
* **Conversions** between numeric and character data are done using as.character() and as.numeric().

**19.Explain the use of conditional statements (if-else) and looping constructs (for, while) in R with examples.**

Loops allow repetitive execution of code blocks.

**1. for Loop**

A for loop iterates over a sequence.

**Example**

for (i in 1:5) {

print(paste("Iteration:", i))

}

**Output:**

[1] "Iteration: 1"

[1] "Iteration: 2"

[1] "Iteration: 3"

[1] "Iteration: 4"

[1] "Iteration: 5"

**Looping Over a Vector**

numbers <- c(10, 20, 30, 40)

for (num in numbers) {

print(num \* 2)

}

**Output:**

[1] 20

[1] 40

[1] 60

[1] 80

**2. while Loop**

A while loop executes as long as a condition remains TRUE.

**Example**

x <- 1

while (x <= 5) {

print(paste("x is", x))

x <- x + 1

}

**Output:**

[1] "x is 1"

[1] "x is 2"

[1] "x is 3"

[1] "x is 4"

[1] "x is 5"

**20.List and explain the commands which are used to manipulate the vectors.**

Vectors are fundamental data structures in R. The following commands are used to manipulate vectors:

**1. Creating Vectors**

* c(): Combines elements into a vector.

vec <- c(10, 20, 30)

* seq(): Creates a sequence of numbers.

seq\_vec <- seq(1, 10, by = 2)

* rep(): Repeats elements in a vector.

rep\_vec <- rep(1:3, times = 2)

**2. Accessing and Modifying Elements**

* []: Access elements using indexing.

vec[2] # Second element

vec[c(1, 3)] # Multiple elements

* vec[n] <- value: Modify an element.

vec[2] <- 100

**3. Vector Operations**

* Arithmetic operations apply to all elements.

vec + 2 # Adds 2 to each element

vec \* 3 # Multiplies each element by 3

**4. Sorting and Ordering**

* sort(): Sorts a vector.

sort(vec) # Ascending order

* order(): Returns the indices for sorting.

vec[order(vec)]

**5. Aggregation and Filtering**

* sum(), mean(), max(), min() compute summary statistics.

sum(vec) # Sum of elements

mean(vec) # Average value

* Logical conditions filter elements.

vec[vec > 20] # Elements greater than 20

These commands help efficiently manipulate and analyze vectors in R. 🚀

**21.How is Data frame converted into list? Explain with an example.**

In R, a **data frame** is a table-like structure that can hold different data types in its columns. It can be converted into a **list** using the as.list() or split() function.

**1. Using as.list() – Column-wise Conversion**

This method converts each column of a data frame into a separate element in a list.

**Example:**

# Create a data frame

df <- data.frame(Name = c("Alice", "Bob", "Charlie"),

Age = c(25, 30, 35),

Score = c(90, 85, 88))

# Convert to list

list\_df <- as.list(df)

# Print the list

print(list\_df)

**Output:**

$Name

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

$Age

[1] 25 30 35

$Score

[1] 90 85 88

🔹 **Each column becomes an element in the list.**

**2. Using split() – Row-wise Conversion**

The split() function creates a list where each row of the data frame becomes an element.

**Example:**

list\_rows <- split(df, seq(nrow(df)))

print(list\_rows)

**Output:**

$`1`

Name Age Score

1 Alice 25 90

$`2`

Name Age Score

2 Bob 30 85

$`3`

Name Age Score

3 Charlie 35 88

🔹 **Each row is stored as a separate data frame inside the list.**

**3. Using lapply() – Convert Each Row into a Named List**

Another way to transform a data frame into a list where each row is stored as a named list.

**Example:**

list\_rows\_named <- lapply(1:nrow(df), function(i) as.list(df[i, ]))

print(list\_rows\_named)

**Output:**

[[1]]

$Name

[1] "Alice"

$Age

[1] 25

$Score

[1] 90

[[2]]

$Name

[1] "Bob"

$Age

[1] 30

$Score

[1] 85

🔹 **Each row is converted into a named list inside a larger list.**

**Conclusion:**

| **Method** | **Description** |
| --- | --- |
| as.list(df) | Converts columns into list elements. |
| split(df, seq(nrow(df))) | Converts rows into small data frames inside a list. |
| lapply(1:nrow(df), function(i) as.list(df[i, ])) | Converts each row into a named list. |

These techniques allow efficient restructuring of data in R for further processing. 🚀

**22.Explain the concept of manipulating data structures with R code.**

In R, data structures are used to store and process data. The main data structures include **vectors, lists, matrices, data frames, and factors**. We can manipulate them by accessing, modifying, and performing operations on them.

**1. Manipulating Vectors (1D Data)**

Vectors store elements of the same type.

vec <- c(10, 20, 30, 40) # Create a vector

vec[2] # Access second element

vec[2] <- 100 # Modify element

✅ **Use:** Storing simple numeric or character data.

**2. Manipulating Lists (Different Data Types)**

Lists store different types of data.

my\_list <- list(Name = "Alice", Age = 25, Scores = c(90, 85))

my\_list$Age # Access Age

my\_list$Age <- 26 # Modify Age

✅ **Use:** Storing mixed data like text, numbers, and vectors.

**3. Manipulating Matrices (2D Numeric Data)**

Matrices store numeric data in rows and columns.

mat <- matrix(1:9, nrow = 3) # Create matrix

mat[2,3] # Access row 2, column 3

mat[1,1] <- 99 # Modify an element

✅ **Use:** Working with structured numeric data.

**4. Manipulating Data Frames (Tabular Data)**

Data frames store tabular data with different data types in columns.

df <- data.frame(Name = c("Alice", "Bob"), Age = c(25, 30))

df$Age # Access Age column

df$Age[1] <- 26 # Modify first Age value

df$City <- c("NY", "LA") # Add new column

✅ **Use:** Handling real-world datasets.

**5. Manipulating Factors (Categorical Data)**

Factors store categorical data.

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

levels(gender) # Get factor levels

levels(gender) <- c("M", "F") # Rename levels

✅ **Use:** Storing and analyzing categories like Gender, Status, etc.

**Conclusion**

* **Vectors** – Simple data storage
* **Lists** – Store mixed data types
* **Matrices** – Numeric data in rows & columns
* **Data Frames** – Real-world tabular data
* **Factors** – Categorical variables
* These manipulations help in organizing and analyzing data efficiently in R.

**23.What is a data frame? Explain with example the commands which are used to create the data frame.**

A **data frame** is a table-like structure in R that stores data in **rows and columns**, where each column can have a different data type (numeric, character, or factor). It is similar to a spreadsheet or a database table.

**Creating a Data Frame in R**

We can create a data frame using the data.frame() function.

**Example:**

# Creating a data frame

df <- data.frame(

Name = c("Alice", "Bob", "Charlie"), # Character column

Age = c(25, 30, 35), # Numeric column

Score = c(90, 85, 88) # Numeric column

)

# Print the data frame

print(df)

**Output:**

Name Age Score

1 Alice 25 90

2 Bob 30 85

3 Charlie 35 88

✅ **Here,** the Name column contains character values, while Age and Score contain numeric values.

**Accessing Data in a Data Frame**

We can access data from a data frame using different methods.

**1. Access a Column**

df$Age # Access Age column

df[["Score"]] # Another way to access Score column

**2. Access a Row**

df[1, ] # Access first row

**3. Access a Specific Value**

df[2, 3] # Access value at row 2, column 3 (Score of Bob)

**Adding a New Column**

df$City <- c("NY", "LA", "SF") # Add a new column

**Conclusion**

* A **data frame** stores data in rows & columns.
* It allows **different data types** in columns.
* The data.frame() function creates a data frame.
* We can access, modify, and add data easily.

**24.Define array data type and explain transpose of array with an example**.

An **array** in R is a multi-dimensional data structure that stores elements of the **same data type** (numeric, character, or logical). It is like an extension of a matrix but can have more than two dimensions.

**Creating an Array in R**

We use the array() function to create an array.

**Example:**

# Creating a 3D array

arr <- array(1:12, dim = c(3, 2, 2))

print(arr)

✅ **Here,**

* 1:12 fills the array with numbers from 1 to 12.
* dim = c(3, 2, 2) means **3 rows, 2 columns, and 2 layers**.

**Transpose of an Array**

The **transpose** of an array swaps its rows and columns.

* For **2D arrays (matrices)**, we use t().
* For **multi-dimensional arrays**, we use aperm().

**Example: Transposing a 2D Array (Matrix)**

# Creating a 2D array (Matrix)

mat <- array(1:6, dim = c(2, 3))

# Transposing the matrix

t\_mat <- t(mat)

print(t\_mat)

✅ **Rows become columns and vice versa.**

**Example: Transposing a 3D Array**

# Transposing a 3D array

t\_arr <- aperm(arr, c(2, 1, 3))

print(t\_arr)

✅ **Here, rows and columns are swapped in a multi-dimensional array.**

**Conclusion**

✔ **Array** stores multi-dimensional data of the same type.  
✔ **array()** creates an array.  
✔ **t()** transposes a matrix.  
✔ **aperm()** transposes multi-dimensional arrays.

**25.Explain the concept of Sorting in relating to Vector and Dataframe.**

Sorting in R is used to arrange elements of a **vector** or **data frame** in **ascending or descending order**. We use the sort() function for vectors and the order() function for data frames.

**1. Sorting a Vector**

Vectors store one-dimensional data, and we can sort them using sort().

**Example:**

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

sorted\_vec <- sort(vec) # Ascending order

print(sorted\_vec)

**Output:**

[1] 10 20 30 50

**Sorting in Descending Order:**

sorted\_vec\_desc <- sort(vec, decreasing = TRUE)

print(sorted\_vec\_desc)

**Output:**

[1] 50 30 20 10

**sort() is used to sort numeric and character vectors.**

**2. Sorting a Data Frame**

Data frames store **tabular data**, and we use order() to sort them based on a column.

**Example:**

df <- data.frame(Name = c("Alice", "Bob", "Charlie"),

Age = c(25, 30, 22),

Score = c(90, 85, 95))

# Sorting by Age (Ascending)

sorted\_df <- df[order(df$Age), ]

print(sorted\_df)

**Output:**

Name Age Score

3 Charlie 22 95

1 Alice 25 90

2 Bob 30 85

**Rows are sorted based on the "Age" column.**

**Sorting by Multiple Columns:**

sorted\_df <- df[order(df$Score, df$Age), ] # First by Score, then Age

**Conclusion**

✔ **Vectors** – Use sort() to arrange values.  
✔ **Data Frames** – Use order() to sort rows by a column.  
✔ Sorting helps in **data analysis** by arranging data logically

**26.What is the use of dimnames()? Explain it with example.**

The dimnames() function in R is used to assign or retrieve **row and column names** for matrices and arrays. It helps in making data more readable and meaningful.

**1. Using dimnames() with a Matrix**

**Example: Assigning Row & Column Names**

# Creating a matrix

mat <- matrix(1:6, nrow = 2, ncol = 3)

# Assigning row and column names

dimnames(mat) <- list(c("Row1", "Row2"), c("Col1", "Col2", "Col3"))

# Printing the matrix

print(mat)

**Output:**

Col1 Col2 Col3

Row1 1 3 5

Row2 2 4 6

**Here, "Row1" & "Row2" are row names, and "Col1" to "Col3" are column names.**

**2. Using dimnames() with an Array**

**Example: Assigning Names to a 3D Array**

# Creating a 3D array

arr <- array(1:12, dim = c(2, 3, 2))

# Assigning names to dimensions

dimnames(arr) <- list(c("A", "B"), c("X", "Y", "Z"), c("Layer1", "Layer2"))

# Printing the array

print(arr)

**Here, names are assigned to rows, columns, and layers in the 3D array.**

**3. Retrieving Dim Names**

print(dimnames(mat)) # Get row & column names of matrix

print(dimnames(arr)) # Get names of array dimensions

**Conclusion**

✔ **dimnames()** assigns or retrieves row & column names for matrices and arrays.  
✔ Helps in making data more **organized and readable**.  
✔ Useful in **data analysis & visualization**. ✅

**27.With the help of an example explain how matrix is converted to data frame.**

A **matrix** is a two-dimensional structure where all elements are of the same type, while a **data frame** can hold different data types in each column. To convert a matrix to a data frame, we use the as.data.frame() function.

E**xample:**

1. **Create a Matrix**

# Create a 3x3 matrix with numbers 1 to 9

m <- matrix(1:9, nrow = 3, ncol = 3)

print(m)

*Output:*

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

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

1. **Convert Matrix to Data Frame**

# Convert matrix 'm' into a data frame

df <- as.data.frame(m)

print(df)

*Output:*

V1 V2 V3

1 1 4 7

2 2 5 8

3 3 6 9

1. **(Optional) Assign Custom Column and Row Names**

# Change column names

colnames(df) <- c("Column1", "Column2", "Column3")

# Change row names

rownames(df) <- c("Row1", "Row2", "Row3")

print(df)

Column1 Column2 Column3

Row1 1 4 7

Row2 2 5 8

Row3 3 6 9

**Explanation:**

* **Step 1:** A matrix m is created using the matrix() function.
* **Step 2:** The as.data.frame(m) function converts the matrix into a data frame df. Each matrix column becomes a data frame column.
* **Step 3 (Optional):** Custom names improve readability and understanding of the data.

This conversion is useful when you want the flexibility of a data frame for further data analysis.

**28.List and explain the commands which are used to manipulate the matrix.**

In R, a matrix is a two-dimensional, homogeneous data structure. Several commands allow us to create, access, modify, and combine matrices. Here are the key commands:

1. **matrix()**
   * **Purpose:** Create a matrix from a vector of elements.
   * **Example:**

m <- matrix(1:9, nrow = 3, ncol = 3)

print(m)

*Creates a 3x3 matrix filled with numbers 1 to 9.*

1. **Indexing with []**
   * **Purpose:** Access or modify specific elements.
   * **Example:**

# Access element at row 2, column 3

element <- m[2, 3]

# Modify element at row 1, column 1

m[1, 1] <- 10

*m[i, j] accesses the element in row i and column j; m[i, ] gets the entire row; m[, j] gets the entire column.*

1. **dim()**
   * **Purpose:** Retrieve the dimensions (number of rows and columns) of the matrix.
   * **Example:**

dims <- dim(m)

print(dims)

1. **nrow() and ncol()**
   * **Purpose:** Get the number of rows and columns respectively.
   * **Example:**

num\_rows <- nrow(m)

num\_cols <- ncol(m)

1. **t()**
   * **Purpose:** Transpose the matrix (swap rows and columns).
   * **Example:**

m\_transposed <- t(m)

print(m\_transposed)

1. **apply()**
   * **Purpose:** Apply a function to the rows or columns.
   * **Example:**

# Sum of each row

row\_sums <- apply(m, 1, sum)

# Mean of each column

col\_means <- apply(m, 2, mean)

*The second argument, 1, applies the function to rows; 2 applies it to columns.*

1. **rbind() and cbind()**
   * **Purpose:** Combine matrices by rows or columns.
   * **Example:**

m2 <- matrix(10:18, nrow = 3)

new\_matrix <- rbind(m, m2) # Combines matrices row-wise

new\_matrix2 <- cbind(m, m2) # Combines matrices column-wise

1. **rownames() and colnames()**
   * **Purpose:** Assign or retrieve names for the rows and columns.
   * **Example:**

colnames(m) <- c("A", "B", "C")

rownames(m) <- c("Row1", "Row2", "Row3")

print(m)

**Conclusion:**

* **Creation:** Use matrix() to build a matrix.
* **Access/Modify:** Index using [] to get or change elements.
* **Dimensions:** dim(), nrow(), and ncol() help retrieve size information.
* **Transpose:** Use t() to swap rows and columns.
* **Apply Functions:** apply() lets you perform operations along rows or columns.
* **Combining:** rbind() and cbind() join matrices.
* **Naming:** rownames() and colnames() improve readability.

**28.What is the use of $ command? How $ command is used in lists explain in detail.**

The **$** operator allows you to access components of an object (like lists and data frames) by name. It simplifies extraction without requiring numeric indices.

**1. Accessing Elements of a List**

Lists in R are collections of elements that can be of different types (vectors, matrices, data frames, etc.).

**Example:**

# Creating a list

my\_list <- list(name = "John", age = 25, scores = c(90, 85, 88))

# Accessing elements using $

my\_list$name # Output: "John"

my\_list$age # Output: 25

my\_list$scores # Output: 90 85 88

* Here, my\_list$name retrieves "John", my\_list$age retrieves 25, and my\_list$scores retrieves a numeric vector.
* **2. Modifying Elements of a List Using $**

You can use $ to update list elements.

**Example:**

# Updating age

my\_list$age <- 30

# Adding a new element

my\_list$city <- "New York"

print(my\_list)

* This changes "age" to 30 and adds a new element "city".

**3. Using $ with Data Frames**

Data frames in R are structured like tables, and $ allows access to columns.

**Example:**

# Creating a data frame

df <- data.frame(Name = c("Alice", "Bob", "Charlie"),

Age = c(25, 30, 22),

Score = c(90, 85, 88))

# Accessing columns using $

df$Name # Output: "Alice" "Bob" "Charlie"

df$Age # Output: 25 30 22

* $ extracts a column as a vector.

**Modifying a Column**

df$Age <- df$Age + 1 # Increments age by 1

print(df)

**4. Using $ in Function Calls**

mean(df$Score) # Calculates the mean of Score column

* This computes the mean of scores.
* **Key Points About $ Operator**

1. Works with **named elements** in **lists** and **data frames**.
2. Provides **convenient access** compared to [] and [[]].
3. Can be used for **modification** and **function calls**.
4. Only allows **one element at a time** (unlike [ ], which can return multiple columns).

**29.How matrix is converted to dataframe? Explain with an example.**

In **R**, a **matrix** is a two-dimensional data structure where all elements must be of the same type, whereas a **dataframe** is a table-like structure where columns can have different data types. The function as.data.frame() is used to convert a matrix into a dataframe.

**Syntax:**

as.data.frame(matrix\_name)

**Example:**

# Creating a matrix

mat <- matrix(1:9, nrow = 3, ncol = 3)

# Converting matrix to dataframe

df <- as.data.frame(mat)

# Printing the dataframe

print(df)

**Output:**

V1 V2 V3

1 1 4 7

2 2 5 8

3 3 6 9

**Key Points:**

1. A matrix is created using matrix().
2. The as.data.frame() function is used to convert it into a dataframe.
3. The dataframe automatically assigns column names (V1, V2, V3).
4. The matrix structure changes to a dataframe format, allowing different data types per column if modified.
5. 5.This method is useful when working with tabular data that requires flexibility in data types.

**30.Define Matrix and explain transpose of matrix with an example.**

**Matrix in R**

A **matrix** in R is a two-dimensional data structure that contains elements of the same type (numeric, character, or logical) arranged in rows and columns. Matrices are created using the matrix() function.

**Syntax to create a matrix in R**

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

* data: Vector of elements to be filled in the matrix
* nrow: Number of rows
* ncol: Number of columns
* byrow: Logical value indicating whether elements should be filled row-wise (TRUE) or column-wise (FALSE)
* dimnames: Optional parameter for row and column names

**Example of a Matrix in R**

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

print(mat)

**Output:**

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

[1,] 1 2 3

[2,] 4 5 6

**Transpose of a Matrix in R**

The **transpose of a matrix** is obtained by interchanging its rows and columns. In R, the transpose of a matrix is calculated using the t() function.

**Example of Transposing a Matrix in R**

# Creating a matrix

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

# Transposing the matrix

transposed\_mat <- t(mat)

# Printing the original and transposed matrices

print("Original Matrix:")

print(mat)

print("Transposed Matrix:")

print(transposed\_mat)

**Output:**

[1] "Original Matrix:"

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

[1,] 1 2 3

[2,] 4 5 6

[1] "Transposed Matrix:"

[,1] [,2]

[1,] 1 4

[2,] 2 5

[3,] 3 6

Here, the original **2×3** matrix is transformed into a **3×2** matrix by interchanging rows and columns.

Let me know if you need further clarification!

**31.Explain selection and Display concepts relating to Matrix objects.**

**Selection and Display of Matrix Objects in R**

In R, **selection** refers to extracting specific elements, rows, or columns from a matrix, while **display** refers to printing or viewing matrix elements.

**1. Displaying a Matrix**

To display a matrix, we use the print() function or simply call the matrix name.

**Example: Display a Matrix**

# Creating a matrix

mat <- matrix(1:9, nrow = 3, ncol = 3)

# Displaying the matrix

print(mat)

**Output:**

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

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

Alternatively, just typing mat in the console will also display the matrix.

**2. Selecting Elements from a Matrix**

Matrix elements can be selected using **indexing** in R. The general syntax for selecting elements is:

matrix[row, column]

* If you specify both row and column, you get a single element.
* If you leave one of them blank, you get an entire row or column.

**a) Selecting a Single Element**

# Selecting element at row 2, column 3

mat[2, 3]

**Output:**

[1] 8

**b) Selecting an Entire Row**

# Selecting the second row

mat[2, ]

**Output:**

[1] 2 5 8

**c) Selecting an Entire Column**

# Selecting the third column

mat[, 3]

**Output:**

[1] 7 8 9

**d) Selecting Multiple Rows and Columns**

# Selecting rows 1 and 2, and columns 2 and 3

mat[c(1,2), c(2,3)]

**Output:**

[,1] [,2]

[1,] 4 7

[2,] 5 8

**3. Selecting Elements Using Logical Conditions**

You can use logical conditions to select elements from a matrix.

**Example: Selecting Elements Greater than 5**

mat[mat > 5]

**Output:**

[1] 6 7 8 9

This returns all elements in the matrix that are greater than 5.

1. **Selecting Elements Using Column and Row Names**
2. Instead of using numeric indices, we can name rows and columns and select elements using names.
3. **Example: Selecting Elements by Name**

# Creating a matrix with row and column names

mat\_named <- matrix(1:9, nrow=3, ncol=3,

dimnames = list(c("Row1", "Row2", "Row3"),

c("Col1", "Col2", "Col3")))

# Selecting by row and column name

mat\_named["Row2", "Col3"]

**Output:**

[1] 8

**Summary of Selection Methods**

| **Selection Type** | **Syntax Example** |
| --- | --- |
| Single element | mat[2,3] → Returns element at row 2, column 3 |
| Entire row | mat[2, ] → Returns all elements in row 2 |
| Entire column | mat[, 3] → Returns all elements in column 3 |
| Multiple rows/columns | mat[c(1,2), c(2,3)] → Returns sub-matrix |
| Based on condition | mat[mat > 5] → Returns elements greater than 5 |
| Using names | mat\_named["Row2", "Col3"] → Returns named element |

**32.List and explain the commands which are used to manipulate the List.**

**List Manipulation Commands in R**

In R, a **list** is a collection of elements that can be of different data types (e.g., vectors, matrices, data frames, or even other lists). Lists are highly flexible and can be manipulated using various commands.

**1. Creating a List**

The list() function is used to create a list.

**Example:**

my\_list <- list(name = "John", age = 25, scores = c(85, 90, 88))

print(my\_list)

**Output:**

$name

[1] "John"

$age

[1] 25

$scores

[1] 85 90 88

**2. Accessing Elements of a List**

**a) Using $ Operator (For named elements)**

my\_list$name

**Output:**

[1] "John"

**b) Using [[ ]] (For specific elements by index or name)**

my\_list[[2]] # Access second element

**Output:**

[1] 25

my\_list[["scores"]] # Access element using name

**Output:**

[1] 85 90 88

**c) Using [ ] (For subsetting as a list)**

Using single brackets [ ] returns a list rather than extracting the element.

my\_list[2]

**Output:**

$age

[1] 25

It still maintains list properties.

**3. Modifying Elements in a List**

**a) Updating an Existing Element**

my\_list$age <- 26 # Updating age

**b) Adding a New Element**

my\_list$city <- "New York"

**c) Removing an Element**

my\_list$age <- NULL # Removes the "age" element

**d) Changing an Element’s Value**

my\_list$scores <- c(95, 92, 89) # Updating scores

**4. Combining Lists**

**a) Using c() to Merge Two Lists**

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

list2 <- list(c = "hello", d = "world")

combined\_list <- c(list1, list2)

print(combined\_list)

**Output:**

$a

[1] 1

$b

[1] 2

$c

[1] "hello"

$d

[1] "world"

**5. Checking the Structure of a List**

**a) Using str()**

str(my\_list)

This provides an overview of the list structure.

**b) Using length()**

r

CopyEdit

length(my\_list) # Returns number of elements in the list

**6. Converting Lists to Other Data Structures**

**a) Converting List to a Vector**

r

CopyEdit

unlist(my\_list)

This converts all elements into a single vector.

**b) Converting List to Data Frame**

If the list contains equal-length elements, it can be converted into a data frame.

r

CopyEdit

df <- as.data.frame(my\_list)

**7. Looping Through a List**

We can use lapply() or sapply() to apply functions over list elements.

**a) Using lapply() (Returns List)**

r

CopyEdit

lapply(my\_list, length)

**b) Using sapply() (Returns Simplified Output)**

r

CopyEdit

sapply(my\_list, length)

**Summary of List Manipulation Commands in R**

| **Operation** | **Command** | **Example** |
| --- | --- | --- |
| **Create a list** | list() | my\_list <- list(a=1, b="text", c=c(1,2,3)) |
| **Access element (by name)** | $ | my\_list$a |
| **Access element (by index)** | [[ ]] | my\_list[[2]] |
| **Subset as a list** | [ ] | my\_list[1:2] |
| **Modify element** | $ <- | my\_list$a <- "new value" |
| **Add new element** | $ <- | my\_list$d <- 100 |
| **Remove element** | NULL | my\_list$b <- NULL |
| **Merge lists** | c() | c(list1, list2) |
| **Convert list to vector** | unlist() | unlist(my\_list) |
| **Convert list to data frame** | as.data.frame() | df <- as.data.frame(my\_list) |
| **Check structure** | str() | str(my\_list) |
| **Loop through list** | lapply(), sapply() | lapply(my\_list, length) |

How data frame is Converted into Matrix, Explain with an example.

**Converting a Data Frame into a Matrix in R**

A **data frame** in R is a table-like structure where each column can have different data types, whereas a **matrix** can only contain elements of the same type. When converting a data frame into a matrix, R automatically coerces all elements to the same type.

**1. Using as.matrix() to Convert a Data Frame to a Matrix**

The as.matrix() function converts a data frame into a matrix.

**Example: Convert a Numeric Data Frame to a Matrix**

# Creating a numeric data frame

df <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6), C = c(7, 8, 9))

# Converting to matrix

mat <- as.matrix(df)

# Printing the matrix

print(mat)

**Output:**

A B C

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

Here, since all the elements are numeric, the conversion is straightforward.

**2. Handling Mixed Data Types in Data Frames**

If a data frame contains different types (e.g., characters and numbers), R converts everything into **character** when using as.matrix().

**Example: Mixed Data Type Data Frame**

# Creating a mixed data frame

df\_mixed <- data.frame(Name = c("Alice", "Bob", "Charlie"), Score = c(85, 90, 78))

# Converting to matrix

mat\_mixed <- as.matrix(df\_mixed)

# Printing the matrix

print(mat\_mixed)

**Output:**

Name Score

[1,] "Alice" "85"

[2,] "Bob" "90"

[3,] "Charlie" "78"

Since the Name column is character data, R converts the entire matrix into **character type**.

**3. Converting Specific Columns to a Matrix**

To avoid type coercion, select only numeric columns before conversion.

**Example: Convert Only Numeric Columns**

# Selecting numeric columns

numeric\_matrix <- as.matrix(df\_mixed[, 2, drop = FALSE])

# Printing the matrix

print(numeric\_matrix)

**Output:**

Score

[1,] 85

[2,] 90

[3,] 78

**4. Checking the Structure After Conversion**

To verify the type after conversion, use class() or mode().

class(mat) # Check object type

mode(mat) # Check data type stored in matrix

If a data frame contained mixed types, mode(mat) would return "character".

**Summary of Data Frame to Matrix Conversion**

| **Case** | **Command** | **Notes** |
| --- | --- | --- |
| Convert numeric data frame to matrix | as.matrix(df) | Works directly |
| Convert mixed data frame to matrix | as.matrix(df\_mixed) | Converts everything to **character** |
| Convert only numeric columns | as.matrix(df[, numeric\_columns]) | Avoids coercion |

**33.Write a short note on: i) Factors, ii) Data Frames, iii) Vectors, iv) List**

**i) Factors**

Factors in R are used to represent categorical data. They store unique values as **levels** and are useful for statistical analysis and plotting.  
Example:

colors <- factor(c("Red", "Blue", "Red", "Green"))

print(colors)

**Output:**

[1] Red Blue Red Green

Levels: Blue Green Red

Factors help in efficient storage and manipulation of categorical variables.

**ii) Data Frames**

A **data frame** is a tabular structure similar to a spreadsheet where columns can contain different data types (numeric, character, logical, etc.). It is created using data.frame().  
Example:

df <- data.frame(Name = c("Alice", "Bob"), Age = c(25, 30))

print(df)

**Output:**

Name Age

1 Alice 25

2 Bob 30

Data frames are widely used for storing and analyzing structured data.

**iii) Vectors**

A **vector** is the simplest data structure in R, containing elements of the same type (numeric, character, logical, etc.). It is created using c().  
Example:

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

print(v)

**Output:**

[1] 1 2 3 4 5

Vectors are the foundation of data manipulation in R.

**iv) List**

A **list** is a collection of different data types, including vectors, matrices, data frames, and even other lists. It is created using list().  
Example:

my\_list <- list(Name = "John", Age = 25, Scores = c(85, 90, 88))

print(my\_list)

**Output:**

$Name

[1] "John"

$Age

[1] 25

$Scores

[1] 85 90 88