MDM Question Bank with Answers

MDM DHRP R Programming Question Bank - Complete Study Notes

Unit 1: Introduction to R Programming

Easy Questions (5 Marks each)

1. What is R, and what is its primary use in data analysis?

Answer: R is a free, open-source programming language and environment specifically designed for statistical computing and graphics. It was developed by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand.

Primary uses in data analysis:

- Statistical analysis and modeling
- Data visualization and graphics
- Data manipulation and cleaning
- Machine learning and predictive analytics
- Time series analysis
- Hypothesis testing
- Regression analysis
- Data mining and exploration

R provides extensive libraries and packages that make it particularly powerful for handling complex statistical operations and creating publication-quality plots.

2. Describe the key features that make R a preferred tool for statistical computing.

Answer: Key Features of R:

- **Open Source**: Free to use and modify, with active community development
- **Comprehensive Statistical Library**: Built-in functions for virtually all statistical techniques
- **Extensibility**: Over 18,000+ packages available on CRAN (Comprehensive R Archive Network)
- **Data Handling**: Excellent support for various data formats (CSV, Excel, databases, web data)
- **Graphics and Visualization**: Advanced plotting capabilities with packages like ggplot2
- **Cross-Platform**: Runs on Windows, Mac, and Linux
- **Reproducible Research**: Integration with R Markdown for documenting analysis

- **Memory Management**: Efficient handling of large datasets
- **Integration**: Works well with other languages (Python, C++, SQL)
- **Active Community**: Large user base providing support and continuous development

3. What is RStudio? List its main components.

Answer: RStudio is an Integrated Development Environment (IDE) for R that provides a user-friendly interface for R programming and data analysis.

Main Components of RStudio:

1. **Source Editor**:
    - Code editing with syntax highlighting
    - Auto-completion and error detection
    - Script management
2. **Console**:
    - Interactive R command line
    - Direct code execution
    - Real-time output display
3. **Environment/History Pane**:
    - Environment: Shows all objects, variables, and data in memory
    - History: Records all executed commands
4. **Files/Plots/Packages/Help Pane**:
    - Files: File browser and management
    - Plots: Display area for graphs and visualizations
    - Packages: Package management interface
    - Help: Documentation and help system

4. Describe the process of quitting RStudio.

Answer: Methods to quit RStudio:

1. **Menu Method**:
    - Go to File → Quit Session
    - Or File → Exit (closes entire RStudio)
2. **Keyboard Shortcut**:
    - Ctrl+Q (Windows/Linux) or Cmd+Q (Mac)
3. **Console Command**:

```
q()
# or
quit()
```

4. **Close Button**: Click the X button on the RStudio window

Important Notes:

- RStudio will prompt to save workspace image (.RData)
- Option to save unsaved scripts
- Choose "Don't Save" for temporary work or "Save" to preserve session

Moderate Questions (5 Marks each)

1. Explain the difference between R and RStudio. How do they interact with each other?

Answer: Differences between R and RStudio:

Aspect	R	RStudio
Nature	Programming language and environment	Integrated Development Environment (IDE)
Functionality	Core statistical computing engine	User interface and development tools
Installation	Can work independently	Requires R to be installed first
Interface	Basic command-line interface	Rich graphical user interface
Features	Statistical functions and packages	Code editing, debugging, project management

How they interact:

1. **Dependency**: RStudio is built on top of R and requires R to function
2. **Communication**: RStudio sends commands to the underlying R engine
3. **Enhancement**: RStudio provides additional features like:
   - Syntax highlighting
   - Code completion
   - Integrated help
   - Project management
   - Version control integration
   - Package management interface
4. **Workflow**: User writes code in RStudio → RStudio passes it to R → R executes and returns results → RStudio displays formatted output

---

Unit 2: R Data Structures and Manipulation

Easy Questions (5 Marks each)

1. How do you create a variable in R? Provide an example.

Answer: In R, variables are created using assignment operators. There are three ways to assign values:

Assignment Operators:

- `<-` (preferred)
- `=`
- `->` (reverse assignment)

Examples:

```r
# Using <- (most common)
name <- "John"
age <- 25
height <- 5.8

# Using =
score = 95
grade = "A"

# Using ->
"Data Science" -> course

# Multiple assignments
x <- y <- z <- 10

# Different data types
is_student <- TRUE        # Logical
numbers <- c(1, 2, 3, 4) # Vector
```

Variable Naming Rules:

- Must start with letter or dot
- Can contain letters, numbers, dots, underscores
- Case-sensitive
- Cannot use reserved words

2. Explain the use of conditional statements in R with an example.

Answer: Conditional statements in R control program flow based on logical conditions.

Types of Conditional Statements:

1. **if statement:**

```
x <- 10
if (x > 5) {
    print(students)

# Data frame with different data types
employee_data <- data.frame(
    EmployeeID = c(101, 102, 103, 104),
    Name = c("John", "Sarah", "Mike", "Lisa"),
    Department = c("IT", "HR", "Finance", "IT"),
    Salary = c(75000, 65000, 70000, 80000),
    JoinDate = as.Date(c("2020-01-15", "2019-05-20", "2021-03-10", "2018-11-05")),
    Active = c(TRUE, TRUE, FALSE, TRUE),
    stringsAsFactors = FALSE  # Prevent automatic factor conversion
)
print(employee_data)
```

2. **From existing vectors:**

```
# Create vectors first
names <- c("Product A", "Product B", "Product C")
prices <- c(29.99, 45.50, 12.75)
in_stock <- c(TRUE, FALSE, TRUE)
categories <- c("Electronics", "Clothing", "Books")

# Create data frame from vectors
products <- data.frame(
    ProductName = names,
    Price = prices,
    InStock = in_stock,
    Category = categories
)
print(products)
```

3. **Reading from files:**

```
# From CSV file
# df_csv <- read.csv("data.csv", header = TRUE)

# From Excel file
# library(readxl)
# df_excel <- read_excel("data.xlsx")
```

4. **From lists:**

```r
# Create data frame from list
data_list <- list(
    x = 1:4,
    y = c("a", "b", "c", "d"),
    z = c(TRUE, FALSE, TRUE, FALSE)
)
df_from_list <- data.frame(data_list)
print(df_from_list)
```

5. **Empty data frame with structure:**

```r
# Create empty data frame with specified structure
empty_df <- data.frame(
    ID = integer(),
    Name = character(),
    Score = numeric(),
    stringsAsFactors = FALSE
)
print("Empty data frame structure:")
print(str(empty_df))
```

3. What are factors in R? Explain their importance.

Answer: Factors in R: Factors are data objects used to categorize data and store it as levels. They are particularly useful for storing categorical data like gender, color, grade levels, etc.

Characteristics of Factors:

- Store categorical data efficiently
- Have predefined possible values called "levels"
- Can be ordered (ordinal) or unordered (nominal)
- Stored internally as integers with labels

Creating Factors:

```r
# Basic factor creation
colors <- factor(c("red", "blue", "red", "green", "blue", "red"))
print(colors)
print(levels(colors))  # Show available levels

# Factor with specified levels
grades <- factor(c("A", "B", "C", "B", "A"),
                levels = c("A", "B", "C", "D", "F"))
print(grades)
```

```r
# Ordered factor
education <- factor(c("High School", "Bachelor", "Master", "PhD", "Bachelor"),
                   levels = c("High School", "Bachelor", "Master", "PhD"),
                   ordered = TRUE)
print(education)
print(is.ordered(education))
```

Importance of Factors:

1. **Memory Efficiency:**

```r
# Factors use less memory for repeated categorical data
char_vector <- rep(c("Category A", "Category B", "Category C"), 1000)
factor_vector <- factor(char_vector)

print(object.size(char_vector))
print(object.size(factor_vector))  # Usually smaller
```

2. **Statistical Analysis:**

```r
# Factors are essential for statistical modeling
set.seed(123)
data <- data.frame(
    treatment = factor(c(rep("A", 10), rep("B", 10), rep("C", 10))),
    response = rnorm(30)
)

# ANOVA requires factors
anova_result <- aov(response ~ treatment, data = data)
summary(anova_result)
```

3. **Plotting and Visualization:**

```r
# Factors control order in plots
survey_data <- data.frame(
    satisfaction = factor(c("Poor", "Good", "Excellent", "Poor", "Good"),
                         levels = c("Poor", "Good", "Excellent")),
    count = c(5, 15, 25, 8, 12)
)

# Proper ordering in plots
barplot(table(survey_data$satisfaction))
```

4. **Data Validation:**

```r
# Factors prevent invalid values
valid_colors <- factor(c("red", "blue", "green"),
                      levels = c("red", "blue", "green", "yellow"))

# Attempting to add invalid level results in NA
# valid_colors[4] <- "purple"  # Would create NA
```

4. How can you merge two data frames in R?

Answer: Methods to Merge Data Frames:

1. **Using merge() function:**

```r
# Create sample data frames
df1 <- data.frame(
    ID = c(1, 2, 3, 4),
    Name = c("Alice", "Bob", "Charlie", "Diana"),
    Age = c(25, 30, 35, 28)
)

df2 <- data.frame(
    ID = c(2, 3, 4, 5),
    Department = c("HR", "IT", "Finance", "Marketing"),
    Salary = c(60000, 75000, 70000, 65000)
)

# Inner join (default)
inner_join <- merge(df1, df2, by = "ID")
print("Inner Join:")
print(inner_join)

# Left join
left_join <- merge(df1, df2, by = "ID", all.x = TRUE)
print("Left Join:")
print(left_join)

# Right join
right_join <- merge(df1, df2, by = "ID", all.y = TRUE)
print("Right Join:")
print(right_join)

# Full outer join
full_join <- merge(df1, df2, by = "ID", all = TRUE)
print("Full Outer Join:")
print(full_join)
```

2. **Merging by different column names:**

```
df3 <- data.frame(
    EmpID = c(1, 2, 3),
    Name = c("John", "Jane", "Jake")
)

df4 <- data.frame(
    EmployeeID = c(1, 2, 4),
    Position = c("Manager", "Analyst", "Director")
)

# Merge by different column names
merged_diff <- merge(df3, df4, by.x = "EmpID", by.y = "EmployeeID")
print("Merge by different columns:")
print(merged_diff)
```

3. **Using rbind() for vertical merging:**

```
# Combine rows (same column structure)
df_part1 <- data.frame(
    Name = c("A", "B"),
    Score = c(85, 90)
)

df_part2 <- data.frame(
    Name = c("C", "D"),
    Score = c(78, 92)
)

vertical_merge <- rbind(df_part1, df_part2)
print("Vertical merge (rbind):")
print(vertical_merge)
```

4. **Using cbind() for horizontal merging:**

```
# Combine columns (same number of rows)
df_names <- data.frame(Name = c("Alice", "Bob", "Charlie"))
df_ages <- data.frame(Age = c(25, 30, 35))
df_cities <- data.frame(City = c("Delhi", "Mumbai", "Bangalore"))

horizontal_merge <- cbind(df_names, df_ages, df_cities)
print("Horizontal merge (cbind):")
print(horizontal_merge)
```

5. What is the apply() function in R, and how does it work with data frames?

Answer: The apply() Function: The apply() function applies a function over the margins of an array or matrix. For data frames, it's used to apply functions across rows or columns.

Syntax:

```
apply(X, MARGIN, FUN, ...)
# X: array, matrix, or data frame
# MARGIN: 1 for rows, 2 for columns
# FUN: function to apply
# ...: additional arguments to FUN
```

Examples with Data Frames:

1. **Basic apply() usage:**

```
# Create sample data frame
student_scores <- data.frame(
    Math = c(85, 92, 78, 88, 95),
    Science = c(80, 85, 90, 82, 88),
    English = c(88, 78, 85, 90, 82),
    History = c(82, 88, 80, 85, 90)
)
rownames(student_scores) <- c("Alice", "Bob", "Charlie", "Diana", "Eve")

print("Student Scores:")
print(student_scores)

# Apply function to rows (calculate average for each student)
student_averages <- apply(student_scores, 1, mean)
print("Student Averages:")
print(student_averages)

# Apply function to columns (calculate average for each subject)
subject_averages <- apply(student_scores, 2, mean)
print("Subject Averages:")
print(subject_averages)
```

2. **Different functions with apply():**

```
# Calculate various statistics
row_sums <- apply(student_scores, 1, sum)
row_max <- apply(student_scores, 1, max)
row_min <- apply(student_scores, 1, min)
row_sd <- apply(student_scores, 1, sd)
```

```r
# Column statistics
col_var <- apply(student_scores, 2, var)
col_median <- apply(student_scores, 2, median)

print("Row sums:")
print(row_sums)
print("Column variances:")
print(col_var)
```

3. **Custom functions with apply():**

```r
# Custom function to calculate range
calculate_range <- function(x) {
    return(max(x) - min(x))
}

# Apply custom function
score_ranges <- apply(student_scores, 1, calculate_range)
print("Score ranges for each student:")
print(score_ranges)

# Anonymous function
# Calculate coefficient of variation
cv <- apply(student_scores, 2, function(x) sd(x)/mean(x) * 100)
print("Coefficient of variation by subject:")
print(round(cv, 2))
```

4. **apply() family functions:**

```r
# lapply() - returns list
result_list <- lapply(student_scores, mean)
print("lapply result (list):")
print(result_list)

# sapply() - returns vector/matrix
result_vector <- sapply(student_scores, mean)
print("sapply result (vector):")
print(result_vector)

# mapply() - multivariate version
weights <- c(0.3, 0.25, 0.25, 0.2)  # Subject weights
weighted_scores <- mapply(function(score, weight) score * weight,
                         student_scores, weights)
print("Weighted scores:")
print(weighted_scores)
```

Moderate Questions (5 Marks each)

1. Write an R function to calculate the mean of each column in a data frame.

Answer:

```r
# Function to calculate mean of each column in a data frame
calculate_column_means <- function(df, na_remove = TRUE, numeric_only = TRUE) {
    # Input validation
    if (!is.data.frame(df)) {
        stop("Input must be a data frame")
    }

    if (nrow(df) == 0) {
        stop("Data frame is empty")
    }

    # Filter numeric columns if specified
    if (numeric_only) {
        numeric_cols <- sapply(df, is.numeric)
        if (sum(numeric_cols) == 0) {
            stop("No numeric columns found in the data frame")
        }
        df_numeric <- df[, numeric_cols, drop = FALSE]
    } else {
        df_numeric <- df
    }

    # Calculate means using different methods

    # Method 1: Using apply()
    means_apply <- apply(df_numeric, 2, function(x) {
        if (is.numeric(x)) {
            return(mean(x, na.rm = na_remove))
        } else {
            return(NA)
        }
    })

    # Method 2: Using sapply()
    means_sapply <- sapply(df_numeric, function(x) {
        if (is.numeric(x)) {
            return(mean(x, na.rm = na_remove))
        } else {
            return(NA)
```

```
        }
    })

    # Method 3: Using colMeans() for numeric data
    if (all(sapply(df_numeric, is.numeric))) {
        means_colmeans <- colMeans(df_numeric, na.rm = na_remove)
    } else {
        means_colmeans <- NULL
    }

    # Return comprehensive results
    result <- list(
        means = means_apply,
        method_used = "apply",
        original_columns = ncol(df),
        numeric_columns = ncol(df_numeric),
        column_names = names(df_numeric)
    )

    return(result)
}

# Enhanced function with additional statistics
comprehensive_column_stats <- function(df, stats = c("mean", "median", "sd",
"var")) {
    # Input validation
    if (!is.data.frame(df)) {
        stop("Input must be a data frame")
    }

    # Get numeric columns
    numeric_cols <- sapply(df, is.numeric)
    df_numeric <- df[, numeric_cols, drop = FALSE]

    if (ncol(df_numeric) == 0) {
        stop("No numeric columns found")
    }

    # Calculate requested statistics
    results <- list()

    if ("mean" %in% stats) {
        results$mean <- sapply(df_numeric, mean, na.rm = TRUE)
```

```
    }
    if ("median" %in% stats) {
        results$median <- sapply(df_numeric, median, na.rm = TRUE)
    }
    if ("sd" %in% stats) {
        results$standard_deviation <- sapply(df_numeric, sd, na.rm = TRUE)
    }
    if ("var" %in% stats) {
        results$variance <- sapply(df_numeric, var, na.rm = TRUE)
    }

    # Convert to data frame for better presentation
    results_df <- do.call(data.frame, results)
    rownames(results_df) <- names(df_numeric)

    return(results_df)
}

# Example usage:
# Create sample data frame
set.seed(123)
sample_data <- data.frame(
    ID = 1:10,
    Name = paste("Person", 1:10),
    Age = sample(20:60, 10),
    Height = rnorm(10, 170, 10),
    Weight = rnorm(10, 70, 15),
    Income = sample(30000:100000, 10),
    Score1 = rnorm(10, 85, 10),
    Score2 = rnorm(10, 80, 12),
    Active = sample(c(TRUE, FALSE), 10, replace = TRUE)
)

# Add some NA values for testing
sample_data$Height[c(3, 7)] <- NA
sample_data$Income[5] <- NA

print("Sample Data:")
print(sample_data)

# Test the functions
print("Column means using basic function:")
basic_means <- calculate_column_means(sample_data)
```

```
print(basic_means$means)

print("Comprehensive statistics:")
comprehensive_stats <- comprehensive_column_stats(sample_data)
print(round(comprehensive_stats, 2))

# Simple one-liner functions
simple_column_means <- function(df) {
    numeric_cols <- sapply(df, is.numeric)
    return(sapply(df[numeric_cols], mean, na.rm = TRUE))
}

print("Simple function result:")
print(round(simple_column_means(sample_data), 2))
```

2. How can you deal with scope issues when working with functions and objects in R?

Answer: Understanding Scope in R:

Scope refers to the visibility and accessibility of variables and objects within different parts of a program. R follows lexical scoping rules.

Types of Scope:

1. **Global Environment vs Local Environment:**

```
# Global variable
global_var <- 100

# Function demonstrating scope
scope_demo <- function() {
    # Local variable
    local_var <- 50

    # Access global variable
    print(paste("Global variable inside function:", global_var))
    print(paste("Local variable:", local_var))

    # Modify global variable using <<-
    global_var <<- 200  # Super assignment

    return(local_var)
}

print(paste("Global variable before function:", global_var))
```

```r
result <- scope_demo()
print(paste("Global variable after function:", global_var))

# Local variable not accessible outside function
# print(local_var)  # This would cause an error
```

2. **Function Parameter Scope:**

```r
# Demonstration of parameter scope
parameter_scope_demo <- function(x, y = 10) {
    # Parameters x and y are local to this function
    z <- x + y

    # Inner function
    inner_function <- function() {
        # Can access variables from parent function
        inner_result <- x * 2  # x is accessible here
        return(inner_result)
    }

    inner_value <- inner_function()

    return(list(sum = z, inner = inner_value))
}

result <- parameter_scope_demo(5)
print(result)
```

Dealing with Scope Issues:

1. **Using Environment Functions:**

```r
# Check current environment
print("Current environment:")
print(environment())

# List objects in global environment
print("Objects in global environment:")
print(ls(envir = .GlobalEnv))

# Function to demonstrate environment manipulation
env_demo <- function() {
    # Create local variables
    local_a <- 10
    local_b <- 20
```

```r
    # List objects in current function environment
    print("Objects in function environment:")
    print(ls(envir = environment()))

    # Access parent environment
    print("Parent environment:")
    print(parent.env(environment()))

    # Assign to global environment explicitly
    assign("global_from_function", local_a + local_b, envir = .GlobalEnv)
}

env_demo()
print(paste("Global variable created from function:", global_from_function))
```

2. **Managing Variable Conflicts:**

```r
# Variable name conflicts
x <- 100  # Global x

conflict_demo <- function(x) {  # Parameter x shadows global x
    print(paste("Parameter x:", x))

    # Access global x explicitly
    global_x <- get("x", envir = .GlobalEnv)
    print(paste("Global x:", global_x))

    # Local x
    x <- x + 50  # Modifies local parameter
    print(paste("Modified local x:", x))

    return(x)
}

result <- conflict_demo(10)
print(paste("Global x after function:", x))  # Still 100
```

3. **Best Practices for Scope Management:**

```r
# Good practice: Explicit parameter passing
calculate_statistics <- function(data, multiplier = 1, add_constant = 0) {
    # Don't rely on global variables
    # Pass all needed values as parameters

```

```r
    result <- (data * multiplier) + add_constant
    return(result)
}

# Bad practice example (avoid this)
bad_function <- function(data) {
    # Relies on global variables - bad practice
    result <- data * global_multiplier + global_constant
    return(result)
}

# Better approach: Return multiple values
comprehensive_analysis <- function(data) {
    # Perform all calculations locally
    mean_val <- mean(data, na.rm = TRUE)
    sd_val <- sd(data, na.rm = TRUE)
    median_val <- median(data, na.rm = TRUE)

    # Return structured result
    return(list(
        mean = mean_val,
        standard_deviation = sd_val,
        median = median_val,
        summary = summary(data)
    ))
}

# Example usage
test_data <- c(1, 2, 3, 4, 5, NA, 7, 8, 9, 10)
analysis_result <- comprehensive_analysis(test_data)
print(analysis_result)
```

4. **Using Closures for Advanced Scope Control:**

```r
# Closure example - function factory
create_counter <- function(initial_value = 0) {
    count <- initial_value

    # Return a function that has access to 'count'
    function() {
        count <<- count + 1  # Modify count in parent environment
        return(count)
    }
}
```

```r
# Create counter instances
counter1 <- create_counter(0)
counter2 <- create_counter(100)

print(counter1())  # 1
print(counter1())  # 2
print(counter2())  # 101
print(counter1())  # 3
print(counter2())  # 102
```

5. **Debugging Scope Issues:**

```r
# Function to debug scope issues
debug_scope <- function() {
    # Show current environment chain
    current_env <- environment()
    level <- 0

    while (!identical(current_env, emptyenv())) {
        cat("Level", level, ": ")
        print(current_env)
        if (level == 0) {
            cat("Objects in this environment:\n")
            print(ls(envir = current_env))
        }
        current_env <- parent.env(current_env)
        level <- level + 1
        if (level > 10) break  # Prevent infinite loop
    }
}

# Function to demonstrate nested scope
nested_scope_demo <- function(a) {
    b <- a * 2

    inner_function <- function(c) {
        d <- b + c  # Can access 'b' from parent function

        innermost_function <- function() {
            # Can access all variables from parent scopes
            result <- a + b + c + d

            # Debug the scope chain
```

```
            cat("\\nScope chain from innermost function:\\n")
            debug_scope()

            return(result)
        }

        return(innermost_function())
    }

    return(inner_function(10))
}

final_result <- nested_scope_demo(5)
print(paste("Final result:", final_result))
```

3. Write a script to work with tables in R, including creating and applying functions.

Answer:

```
# Comprehensive Table Operations Script in R

# Load required libraries
# install.packages(c("dplyr", "tidyr", "knitr"))
library(dplyr)      # For data manipulation
library(tidyr)      # For data reshaping
library(knitr)      # For nice table formatting

# ============================================================================
# PART 1: Creating Tables and Data
# ============================================================================

# Function to create sample sales data
create_sales_data <- function(n_records = 100) {
    set.seed(123)  # For reproducible data

    sales_data <- data.frame(
        OrderID = 1:n_records,
        CustomerID = sample(1000:9999, n_records, replace = TRUE),
        Product = sample(c("Laptop", "Phone", "Tablet", "Watch", "Headphones"),
                        n_records, replace = TRUE),
        Category = sample(c("Electronics", "Accessories"), n_records, replace =
TRUE),
        Region = sample(c("North", "South", "East", "West"), n_records, replace =
TRUE),
        SalesRep = sample(c("Alice", "Bob", "Charlie", "Diana", "Eve"),
```

```r
                         n_records, replace = TRUE),
        Quantity = sample(1:10, n_records, replace = TRUE),
        UnitPrice = round(runif(n_records, 100, 2000), 2),
        Date = sample(seq(as.Date("2023-01-01"), as.Date("2023-12-31"), by =
"day"),
                     n_records, replace = TRUE),
        stringsAsFactors = FALSE
    )

    # Calculate total sales
    sales_data$TotalSales <- sales_data$Quantity * sales_data$UnitPrice

    return(sales_data)
}

# ============================================================================
# PART 2: Table Creation Functions
# ============================================================================

# Function to create frequency tables
create_frequency_table <- function(data, column_name) {
    if (!column_name %in% names(data)) {
        stop(paste("Column", column_name, "not found in data"))
    }

    freq_table <- table(data[[column_name]])

    # Convert to data frame for better handling
    freq_df <- data.frame(
        Category = names(freq_table),
        Frequency = as.numeric(freq_table),
        Percentage = round(as.numeric(freq_table) / sum(freq_table) * 100, 2)
    )

    # Sort by frequency (descending)
    freq_df <- freq_df[order(freq_df$Frequency, decreasing = TRUE), ]
    rownames(freq_df) <- NULL

    return(freq_df)
}

# Function to create cross-tabulation tables
create_crosstab <- function(data, row_var, col_var) {
```

```
    if (!row_var %in% names(data) || !col_var %in% names(data)) {
        stop("One or both variables not found in data")
    }

    # Basic cross-tabulation
    crosstab <- table(data[[row_var]], data[[col_var]])

    # Add margins (totals)
    crosstab_with_margins <- addmargins(crosstab)

    # Calculate percentages
    crosstab_pct <- prop.table(crosstab) * 100

    return(list(
        counts = crosstab,
        with_margins = crosstab_with_margins,
        percentages = round(crosstab_pct, 2)
    ))
}

# Function to create summary tables
create_summary_table <- function(data, group_var, summary_var) {
    if (!group_var %in% names(data) || !summary_var %in% names(data)) {
        stop("One or both variables not found in data")
    }

    if (!is.numeric(data[[summary_var]])) {
        stop("Summary variable must be numeric")
    }

    summary_table <- data %>%
        group_by(!!sym(group_var)) %>%
        summarise(
            Count = n(),
            Mean = round(mean(!!sym(summary_var), na.rm = TRUE), 2),
            Median = round(median(!!sym(summary_var), na.rm = TRUE), 2),
            SD = round(sd(!!sym(summary_var), na.rm = TRUE), 2),
            Min = min(!!sym(summary_var), na.rm = TRUE),
            Max = max(!!sym(summary_var), na.rm = TRUE),
            Total = round(sum(!!sym(summary_var), na.rm = TRUE), 2),
            .groups = 'drop'
        )
```

```r
    return(as.data.frame(summary_table))
}

# ============================================================================
# PART 3: Advanced Table Manipulation Functions
# ============================================================================

# Function to pivot table (wide to long)
pivot_table_long <- function(data, id_cols, value_cols) {
    pivoted <- data %>%
        pivot_longer(
            cols = all_of(value_cols),
            names_to = "Variable",
            values_to = "Value"
        )

    return(pivoted)
}

# Function to create comprehensive sales analysis table
analyze_sales_performance <- function(sales_data) {
    analysis <- sales_data %>%
        group_by(Region, Product) %>%
        summarise(
            Total_Orders = n(),
            Total_Quantity = sum(Quantity),
            Total_Revenue = round(sum(TotalSales), 2),
            Avg_Order_Value = round(mean(TotalSales), 2),
            Avg_Unit_Price = round(mean(UnitPrice), 2),
            .groups = 'drop'
        ) %>%
        arrange(desc(Total_Revenue))

    return(as.data.frame(analysis))
}

# Function to create time-based analysis
time_series_analysis <- function(sales_data) {
    # Extract month and year
    sales_data$Month <- format(sales_data$Date, "%Y-%m")

    monthly_analysis <- sales_data %>%
        group_by(Month) %>%
```

```r
        summarise(
            Orders = n(),
            Revenue = round(sum(TotalSales), 2),
            Avg_Order_Size = round(mean(Quantity), 2),
            Unique_Customers = n_distinct(CustomerID),
            .groups = 'drop'
        ) %>%
        arrange(Month)

    return(as.data.frame(monthly_analysis))
}

# ============================================================================
# PART 4: Table Formatting and Display Functions
# ============================================================================

# Function to format tables nicely
format_table <- function(data, title = "", caption = "") {
    cat("\\n")
    cat("="*nchar(title), "\\n")
    cat(title, "\\n")
    cat("="*nchar(title), "\\n")

    if (caption != "") {
        cat(caption, "\\n\\n")
    }

    # Use kable for better formatting if available
    if (requireNamespace("knitr", quietly = TRUE)) {
        print(knitr::kable(data, format = "simple"))
    } else {
        print(data)
    }

    cat("\\n")
}

# Function to export tables to different formats
export_table <- function(data, filename, format = "csv") {
    switch(format,
           "csv" = write.csv(data, paste0(filename, ".csv"), row.names = FALSE),
           "txt" = write.table(data, paste0(filename, ".txt"),
                              sep = "\\t", row.names = FALSE),
```

```r
            "rds" = saveRDS(data, paste0(filename, ".rds")),
            stop("Unsupported format. Use 'csv', 'txt', or 'rds'")
    )

    cat("Table exported as:", paste0(filename, ".", format), "\\n")
}

# ============================================================================
# PART 5: Main Script Execution
# ============================================================================

# Create sample data
cat("Creating sample sales data...\\n")
sales_data <- create_sales_data(200)

cat("Sample of created data:\\n")
print(head(sales_data, 10))

# Frequency analysis
cat("\\n" %+% "="*50 %+% "\\n")
cat("FREQUENCY ANALYSIS\\n")
cat("="*50 %+% "\\n")

product_freq <- create_frequency_table(sales_data, "Product")
format_table(product_freq, "Product Frequency Distribution")

region_freq <- create_frequency_table(sales_data, "Region")
format_table(region_freq, "Regional Distribution")

# Cross-tabulation analysis
cat("\\n" %+% "="*50 %+% "\\n")
cat("CROSS-TABULATION ANALYSIS\\n")
cat("="*50 %+% "\\n")

product_region_crosstab <- create_crosstab(sales_data, "Product", "Region")
cat("Product vs Region Cross-tabulation (Counts):\\n")
print(product_region_cro"x is greater than 5")
}
```

2. **if-else statement:**

```r
age <- 18
if (age >= 18) {
    print("You are eligible to vote")
```

```
} else {
    print("You are not eligible to vote")
}
```

3. **if-else if-else statement:**

```
score <- 85
if (score >= 90) {
    grade <- "A"
} else if (score >= 80) {
    grade <- "B"
} else if (score >= 70) {
    grade <- "C"
} else {
    grade <- "F"
}
print(paste("Your grade is:", grade))
```

4. **ifelse() function (vectorized):**

```
numbers <- c(1, 5, 10, 15, 20)
result <- ifelse(numbers > 10, "High", "Low")
print(result)  # "Low" "Low" "Low" "High" "High"
```

Moderate Questions (5 Marks each)

1. Write an R function to calculate the sum of elements in a vector.

Answer:

```
# Method 1: Using built-in sum function
calculate_sum <- function(vector) {
    if (!is.numeric(vector)) {
        stop("Input must be a numeric vector")
    }
    return(sum(vector))
}

# Method 2: Manual calculation using loop
manual_sum <- function(vector) {
    if (!is.numeric(vector)) {
        stop("Input must be a numeric vector")
    }

    total <- 0
    for (i in 1:length(vector)) {
```

```r
        total <- total + vector[i]
    }
    return(total)
}

# Method 3: Using Reduce function
reduce_sum <- function(vector) {
    if (!is.numeric(vector)) {
        stop("Input must be a numeric vector")
    }
    return(Reduce("+", vector))
}

# Example usage:
numbers <- c(1, 2, 3, 4, 5)
print(calculate_sum(numbers))  # Output: 15
print(manual_sum(numbers))     # Output: 15
print(reduce_sum(numbers))     # Output: 15

# Handle NA values
numbers_with_na <- c(1, 2, NA, 4, 5)
sum_with_na_handling <- function(vector) {
    return(sum(vector, na.rm = TRUE))
}
print(sum_with_na_handling(numbers_with_na))  # Output: 12
```

Unit 3: R Packages and Functions

Easy Questions (5 Marks each)

1. What is an R package? Why is it used?

Answer: An R package is a collection of R functions, data, and compiled code organized in a standardized format that extends R's capabilities.

Components of an R Package:

- R functions and code
- Documentation and help files
- Sample datasets
- Compiled code (C, C++, Fortran)
- Metadata (DESCRIPTION file)

Why R Packages are Used:

1. **Extend Functionality**: Add specialized functions not available in base R
2. **Code Reusability**: Share and reuse code across projects
3. **Standardization**: Consistent structure and documentation
4. **Community Contribution**: Access to thousands of packages from experts
5. **Specialized Domains**: Packages for specific fields (bioinformatics, finance, etc.)
6. **Quality Assurance**: Tested and peer-reviewed code
7. **Documentation**: Comprehensive help and examples

Popular Packages:

- `dplyr`: Data manipulation
- `ggplot2`: Data visualization
- `tidyr`: Data tidying
- `lubridate`: Date/time handling
- `caret`: Machine learning

2. How do you install a package in RStudio?

Answer: Methods to Install Packages in RStudio:

1. **Using Console Commands:**

```r
# Install from CRAN
install.packages("ggplot2")

# Install multiple packages
install.packages(c("dplyr", "tidyr", "ggplot2"))

# Install from specific repository
install.packages("ggplot2", repos = "https://cran.r-project.org")
```

2. **Using RStudio GUI:**
   - Go to Tools → Install Packages
   - Type package name in the dialog box
   - Click Install
3. **Using Packages Pane:**
   - Click on "Packages" tab in bottom-right panel
   - Click "Install" button
   - Enter package name and click Install
4. **From GitHub:**

```r
# First install devtools
install.packages("devtools")
# Then install from GitHub
devtools::install_github("username/packagename")
```

5. **From Local File:**

```r
install.packages("path/to/package.tar.gz", repos = NULL, type = "source")
```

Loading Packages:

```r
# Load package
library(ggplot2)
# or
require(ggplot2)
```

3. Write a simple R function to calculate the square of a number.

Answer:

```r
# Simple square function
square <- function(x) {
    return(x^2)
}

# Alternative without explicit return
square_alt <- function(x) {
    x^2
}

# With input validation
square_safe <- function(x) {
    if (!is.numeric(x)) {
        stop("Input must be numeric")
    }
    return(x^2)
}

# Vectorized version for multiple numbers
square_vector <- function(x) {
    if (!is.numeric(x)) {
        stop("Input must be numeric")
    }
    return(x^2)
}
```

```r
# With default parameter
square_default <- function(x = 1) {
    return(x^2)
}

# Example usage:
print(square(5))          # Output: 25
print(square_alt(4))      # Output: 16
print(square_safe(3))     # Output: 9
print(square_vector(c(1, 2, 3, 4)))  # Output: 1 4 9 16
print(square_default())   # Output: 1 (using default)
```

4. Explain how to download and import data in R.

Answer: Methods to Download and Import Data in R:

1. **CSV Files:**

```r
# From local file
data <- read.csv("file.csv")

# From URL
url <- "https://example.com/data.csv"
data <- read.csv(url)

# With specific parameters
data <- read.csv("file.csv", header = TRUE, sep = ",", stringsAsFactors = FALSE)
```

2. **Excel Files:**

```r
# Install and load required package
install.packages("readxl")
library(readxl)

# Read Excel file
data <- read_excel("file.xlsx")
data <- read_excel("file.xlsx", sheet = "Sheet1")
```

3. **Text Files:**

```r
# Fixed width
data <- read.table("file.txt", header = TRUE, sep = "\t")

# Custom delimiter
data <- read.delim("file.txt", sep = "|")
```

4. **From Databases:**

```r
library(DBI)
library(RSQLite)

# Connect to database
con <- dbConnect(RSQLite::SQLite(), "database.db")
data <- dbGetQuery(con, "SELECT * FROM table_name")
dbDisconnect(con)
```

5. **Web Scraping:**

```r
library(rvest)
library(httr)

# Download and parse HTML
url <- "https://example.com"
page <- read_html(url)
data <- html_table(page)
```

6. **APIs:**

```r
library(httr)
library(jsonlite)

# GET request
response <- GET("https://api.example.com/data")
data <- fromJSON(content(response, "text"))
```

Moderate Questions (5 Marks each)

1. Describe the process of creating a custom function in R. Provide an example.

Answer: Process of Creating Custom Functions in R:

Function Syntax:

```r
function_name <- function(parameter1, parameter2 = default_value) {
    # Function body
    # Calculations and operations
    return(result)  # Optional explicit return
}
```

Steps to Create a Custom Function:

1. **Define Function Name**: Choose descriptive name
2. **Specify Parameters**: Input arguments with optional defaults
3. **Write Function Body**: Logic and calculations

4. **Return Value**: Use return() or last expression
5. **Test Function**: Verify with different inputs
6. **Document Function**: Add comments and examples

Comprehensive Example:

```r
# Complex function to calculate statistics for a numeric vector
calculate_statistics <- function(data, na_remove = TRUE, round_digits = 2) {
    # Input validation
    if (!is.numeric(data)) {
        stop("Error: Input must be a numeric vector")
    }

    if (length(data) == 0) {
        stop("Error: Input vector is empty")
    }

    # Remove NA values if specified
    if (na_remove) {
        clean_data <- data[!is.na(data)]
        if (length(clean_data) == 0) {
            stop("Error: No valid data after removing NA values")
        }
    } else {
        clean_data <- data
    }

    # Calculate statistics
    stats <- list(
        count = length(clean_data),
        mean = round(mean(clean_data, na.rm = na_remove), round_digits),
        median = round(median(clean_data, na.rm = na_remove), round_digits),
        min = min(clean_data, na.rm = na_remove),
        max = max(clean_data, na.rm = na_remove),
        std_dev = round(sd(clean_data, na.rm = na_remove), round_digits),
        variance = round(var(clean_data, na.rm = na_remove), round_digits)
    )

    # Return results
    return(stats)
}

# Example usage:
```

```r
test_data <- c(1, 2, 3, 4, 5, NA, 7, 8, 9, 10)
result <- calculate_statistics(test_data)
print(result)

# Function with multiple return values
analyze_grades <- function(scores) {
    avg_score <- mean(scores)
    letter_grades <- ifelse(scores >= 90, "A",
                           ifelse(scores >= 80, "B",
                           ifelse(scores >= 70, "C",
                           ifelse(scores >= 60, "D", "F"))))

    return(list(
        average = avg_score,
        grades = letter_grades,
        pass_rate = sum(scores >= 60) / length(scores) * 100
    ))
}

# Test the function
student_scores <- c(85, 92, 78, 88, 95, 67, 73, 89)
grade_analysis <- analyze_grades(student_scores)
print(grade_analysis)
```

Unit 4: Matrices, Arrays, and Lists

Easy Questions (5 Marks each)

1. How do you create a matrix in R? Provide an example.

Answer: Methods to Create Matrices in R:

1. **Using matrix() function:**

```r
# Basic matrix creation
mat1 <- matrix(1:12, nrow = 3, ncol = 4)
print(mat1)

# Specify data and dimensions
mat2 <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
print(mat2)

# Fill by rows instead of columns
mat3 <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
print(mat3)
```

```r
# Matrix with specific values
mat4 <- matrix(0, nrow = 3, ncol = 3)  # Matrix of zeros
print(mat4)
```

2. **Using rbind() and cbind():**

```r
# Combine rows
row1 <- c(1, 2, 3)
row2 <- c(4, 5, 6)
mat_rbind <- rbind(row1, row2)
print(mat_rbind)

# Combine columns
col1 <- c(1, 4)
col2 <- c(2, 5)
col3 <- c(3, 6)
mat_cbind <- cbind(col1, col2, col3)
print(mat_cbind)
```

3. **Converting arrays or vectors:**

```r
# From vector
vec <- 1:12
dim(vec) <- c(3, 4)  # Convert to 3x4 matrix
print(vec)

# Create named matrix
mat_named <- matrix(1:6, nrow = 2, ncol = 3,
                   dimnames = list(c("Row1", "Row2"),
                                  c("Col1", "Col2", "Col3")))
print(mat_named)
```

2. Describe the operations that can be performed on matrices in R.

Answer: Matrix Operations in R:

1. **Basic Arithmetic Operations:**

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

# Element-wise operations
addition <- A + B
subtraction <- A - B
multiplication <- A * B  # Element-wise
```

```r
division <- A / B

# Scalar operations
scalar_mult <- A * 2
scalar_add <- A + 5
```

2. **Matrix Algebra:**

```r
# Matrix multiplication
mat_mult <- A %*% B

# Transpose
transpose_A <- t(A)

# Determinant
det_A <- det(A)

# Inverse (if square and invertible)
if (det(A) != 0) {
    inverse_A <- solve(A)
}

# Eigenvalues and eigenvectors
eigen_result <- eigen(A)
```

3. **Indexing and Subsetting:**

```r
mat <- matrix(1:12, nrow = 3, ncol = 4)

# Access elements
element <- mat[2, 3]         # Row 2, Column 3
row <- mat[2, ]              # Entire row 2
column <- mat[, 3]           # Entire column 3
submatrix <- mat[1:2, 2:4]  # Submatrix
```

4. **Matrix Properties:**

```r
# Dimensions
nrow(mat)  # Number of rows
ncol(mat)  # Number of columns
dim(mat)   # Both dimensions

# Summary statistics
rowSums(mat)    # Sum of each row
colSums(mat)    # Sum of each column
```

```r
rowMeans(mat)   # Mean of each row
colMeans(mat)   # Mean of each column
```

3. What is the difference between a vector and a matrix in R?

Answer: Differences between Vector and Matrix:

Aspect	Vector	Matrix
Dimensions	One-dimensional	Two-dimensional (rows × columns)
Structure	Linear sequence of elements	Rectangular array of elements
Creation	`c()`, `seq()`, `:`	`matrix()`, `rbind()`, `cbind()`
Indexing	`vector[i]`	`matrix[i, j]`
Storage	Stored as a single sequence	Stored column-wise by default

Examples:

```r
# Vector examples
vector1 <- c(1, 2, 3, 4, 5)          # Numeric vector
vector2 <- c("a", "b", "c")          # Character vector
vector3 <- 1:10                      # Sequence vector

print(length(vector1))  # Length: 5
print(vector1[3])       # Access 3rd element

# Matrix examples
matrix1 <- matrix(1:6, nrow = 2, ncol = 3)
matrix2 <- matrix(c("a", "b", "c", "d"), nrow = 2)

print(dim(matrix1))     # Dimensions: 2 3
print(matrix1[2, 3])    # Access element at row 2, column 3

# Conversion between vector and matrix
vec_to_mat <- matrix(vector1, nrow = 1)  # Row vector
mat_to_vec <- as.vector(matrix1)         # Convert matrix to vector
```

Key Differences:

- Vectors have only length, matrices have rows and columns
- Matrix indexing requires two indices [row, column]
- Matrices can perform matrix algebra operations
- Vectors are simpler for sequential data, matrices for tabular data

4. How do you add or delete rows and columns in a matrix?

Answer: Adding and Deleting Rows and Columns:

1. **Adding Rows:**

```r
# Original matrix
original <- matrix(1:6, nrow = 2, ncol = 3)
print("Original matrix:")
print(original)

# Add row using rbind()
new_row <- c(7, 8, 9)
matrix_with_row <- rbind(original, new_row)
print("After adding row:")
print(matrix_with_row)

# Add multiple rows
new_rows <- matrix(c(10, 11, 12, 13, 14, 15), nrow = 2, ncol = 3)
matrix_multiple_rows <- rbind(original, new_rows)
print("After adding multiple rows:")
print(matrix_multiple_rows)
```

2. **Adding Columns:**

```r
# Add column using cbind()
new_col <- c(7, 8)
matrix_with_col <- cbind(original, new_col)
print("After adding column:")
print(matrix_with_col)

# Add multiple columns
new_cols <- matrix(c(9, 10, 11, 12), nrow = 2, ncol = 2)
matrix_multiple_cols <- cbind(original, new_cols)
print("After adding multiple columns:")
print(matrix_multiple_cols)
```

3. **Deleting Rows:**

```r
# Delete specific rows (negative indexing)
matrix_without_row1 <- original[-1, ]      # Delete row 1
print("After deleting row 1:")
print(matrix_without_row1)

# Delete multiple rows
large_matrix <- matrix(1:15, nrow = 5, ncol = 3)
matrix_without_rows <- large_matrix[-c(2, 4), ]  # Delete rows 2 and 4
```

```r
print("After deleting rows 2 and 4:")
print(matrix_without_rows)
```

4. **Deleting Columns:**

```r
# Delete specific columns
matrix_without_col2 <- original[, -2]      # Delete column 2
print("After deleting column 2:")
print(matrix_without_col2)

# Delete multiple columns
matrix_without_cols <- original[, -c(1, 3)]  # Delete columns 1 and 3
print("After deleting columns 1 and 3:")
print(matrix_without_cols)
```

Moderate Questions (5 Marks each)

1. How would you handle higher-dimensional arrays in R? Explain with an example.

Answer: Higher-Dimensional Arrays in R:

Arrays in R can have more than two dimensions, useful for complex data structures like 3D data, time series with multiple variables, or multi-dimensional datasets.

Creating Higher-Dimensional Arrays:

1. **3D Array Example:**

```r
# Create a 3D array (2x3x4)
# Dimensions: 2 rows, 3 columns, 4 layers
array_3d <- array(1:24, dim = c(2, 3, 4))
print("3D Array:")
print(array_3d)

# Access dimensions
print(paste("Dimensions:", paste(dim(array_3d), collapse = "x")))

# Named dimensions
dimnames(array_3d) <- list(
    Rows = c("R1", "R2"),
    Cols = c("C1", "C2", "C3"),
    Layers = c("L1", "L2", "L3", "L4")
)
print("3D Array with names:")
print(array_3d)
```

2. **4D Array Example:**

```r
# 4D array for time series data across multiple locations and variables
# Dimensions: 10 time points, 5 locations, 3 variables, 2 scenarios
array_4d <- array(rnorm(300), dim = c(10, 5, 3, 2))
dimnames(array_4d) <- list(
    Time = paste("T", 1:10, sep = ""),
    Location = paste("Loc", 1:5, sep = ""),
    Variable = c("Temperature", "Humidity", "Pressure"),
    Scenario = c("Current", "Future")
)
```

Array Operations:

1. **Indexing and Subsetting:**

```r
# Access specific elements
element <- array_3d[1, 2, 3]  # Row 1, Col 2, Layer 3
print(paste("Element [1,2,3]:", element))

# Extract slices
layer_1 <- array_3d[, , 1]    # All rows and cols from layer 1
row_1 <- array_3d[1, , ]      # Row 1 from all layers
col_2 <- array_3d[, 2, ]      # Column 2 from all layers

print("Layer 1:")
print(layer_1)
```

2. **Array Functions:**

```r
# Apply functions across dimensions
# apply(array, margin, function)
# margin: 1=rows, 2=columns, 3=layers, etc.

# Sum across layers (margin 3)
sum_layers <- apply(array_3d, c(1, 2), sum)
print("Sum across layers:")
print(sum_layers)

# Mean across rows (margin 1)
mean_rows <- apply(array_3d, c(2, 3), mean)
print("Mean across rows:")
print(mean_rows)

# Total sum of each layer
layer_sums <- apply(array_3d, 3, sum)
```

```
print("Sum of each layer:")
print(layer_sums)
```

3. **Real-world Example - Climate Data:**

```
# Climate data: temperature readings
# Dimensions: 12 months, 4 locations, 5 years
set.seed(123)
climate_data <- array(
    rnorm(240, mean = 20, sd = 5),  # Random temperature data
    dim = c(12, 4, 5),
    dimnames = list(
        Month = month.abb,
        Location = c("Delhi", "Mumbai", "Chennai", "Kolkata"),
        Year = 2019:2023
    )
)

# Calculate annual averages for each location
annual_avg <- apply(climate_data, c(2, 3), mean)
print("Annual temperature averages by location:")
print(round(annual_avg, 2))

# Monthly averages across all years and locations
monthly_avg <- apply(climate_data, 1, mean)
print("Monthly averages:")
print(round(monthly_avg, 2))

# Find maximum temperature for each year
yearly_max <- apply(climate_data, 3, max)
print("Yearly maximum temperatures:")
print(round(yearly_max, 2))
```

4. **Array Manipulation:**

```
# Reshape arrays
new_array <- array(array_3d, dim = c(3, 2, 4))

# Convert to data frame for analysis
df_from_array <- as.data.frame.table(array_3d)
colnames(df_from_array) <- c("Row", "Col", "Layer", "Value")
head(df_from_array)

# Arithmetic operations on arrays
```

```
array_doubled <- array_3d * 2
array_sum <- array_3d + 10
```

Unit 5: Data Frames - Complete Question Bank

Easy Questions (5 Marks each)

1. What is a data frame in R? How is it different from a matrix?

Data Frame Definition: A data frame is a two-dimensional data structure in R that stores data in a tabular format with rows and columns. It's similar to a spreadsheet or database table where each column can contain different data types.

Key Characteristics of Data Frames:

- Columns can have different data types (numeric, character, logical, factor)
- Each column must have the same length
- Rows represent observations, columns represent variables
- Most commonly used data structure for statistical analysis

Differences between Data Frame and Matrix:

Aspect	Data Frame	Matrix
Data Types	Mixed types (numeric, character, logical)	Single type only
Flexibility	Each column can be different type	All elements same type
Usage	Statistical analysis, real-world data	Mathematical operations
Column Names	Always has column names	Optional column names
Indexing	df$column, df[row, col]	matrix[row, col]
Functions	data.frame()	matrix()

Examples:

```
# Data Frame - Mixed data types
df <- data.frame(
    Name = c("Alice", "Bob", "Charlie"),
    Age = c(25, 30, 35),
    Married = c(TRUE, FALSE, TRUE),
    Salary = c(50000, 60000, 70000)
)
print("Data Frame:")
print(df)
```

```r
print(str(df))  # Structure shows different data types

# Matrix - Single data type
mat <- matrix(c(25, 30, 35, 50000, 60000, 70000),
              nrow = 3, ncol = 2)
print("Matrix:")
print(mat)
print(str(mat))  # All elements converted to same type
```

2. How do you create a data frame in R? Provide an example.

Methods to Create Data Frames:

1. Using data.frame() function:

```r
# Basic data frame creation
students <- data.frame(
    ID = 1:5,
    Name = c("Alice", "Bob", "Charlie", "Diana", "Eve"),
    Age = c(20, 21, 22, 20, 23),
    Grade = c("A", "B", "A", "C", "B"),
    Passed = c(TRUE, TRUE, TRUE, FALSE, TRUE)
)
print(students)

# Data frame with different data types
employee_data <- data.frame(
    EmployeeID = c(101, 102, 103, 104),
    Name = c("John", "Sarah", "Mike", "Lisa"),
    Department = c("IT", "HR", "Finance", "IT"),
    Salary = c(75000, 65000, 70000, 80000),
    JoinDate = as.Date(c("2020-01-15", "2019-05-20", "2021-03-10", "2018-11-05")),
    Active = c(TRUE, TRUE, FALSE, TRUE),
    stringsAsFactors = FALSE  # Prevent automatic factor conversion
)
print(employee_data)
```

2. From existing vectors:

```r
# Create vectors first
names <- c("Product A", "Product B", "Product C")
prices <- c(29.99, 45.50, 12.75)
in_stock <- c(TRUE, FALSE, TRUE)
categories <- c("Electronics", "Clothing", "Books")
```

```r
# Create data frame from vectors
products <- data.frame(
    ProductName = names,
    Price = prices,
    InStock = in_stock,
    Category = categories
)
print(products)
```

3. Reading from files:

```r
# From CSV file
# df_csv <- read.csv("data.csv", header = TRUE)

# From Excel file
# library(readxl)
# df_excel <- read_excel("data.xlsx")
```

4. From lists:

```r
# Create data frame from list
data_list <- list(
    x = 1:4,
    y = c("a", "b", "c", "d"),
    z = c(TRUE, FALSE, TRUE, FALSE)
)
df_from_list <- data.frame(data_list)
print(df_from_list)
```

5. Empty data frame with structure:

```r
# Create empty data frame with specified structure
empty_df <- data.frame(
    ID = integer(),
    Name = character(),
    Score = numeric(),
    stringsAsFactors = FALSE
)
print("Empty data frame structure:")
print(str(empty_df))
```

3. What are factors in R? Explain their importance.

Factors in R: Factors are data objects used to categorize data and store it as levels. They are particularly useful for storing categorical data like gender, color, grade levels, etc.

Characteristics of Factors:

- Store categorical data efficiently
- Have predefined possible values called "levels"
- Can be ordered (ordinal) or unordered (nominal)
- Stored internally as integers with labels

Creating Factors:

```
# Basic factor creation
colors <- factor(c("red", "blue", "red", "green", "blue", "red"))
print(colors)
print(levels(colors))  # Show available levels

# Factor with specified levels
grades <- factor(c("A", "B", "C", "B", "A"),
                levels = c("A", "B", "C", "D", "F"))
print(grades)

# Ordered factor
education <- factor(c("High School", "Bachelor", "Master", "PhD", "Bachelor"),
                   levels = c("High School", "Bachelor", "Master", "PhD"),
                   ordered = TRUE)
print(education)
print(is.ordered(education))
```

Importance of Factors:

1. Memory Efficiency:

```
# Factors use less memory for repeated categorical data
char_vector <- rep(c("Category A", "Category B", "Category C"), 1000)
factor_vector <- factor(char_vector)

print(object.size(char_vector))
print(object.size(factor_vector))  # Usually smaller
```

2. Statistical Analysis:

```
# Factors are essential for statistical modeling
set.seed(123)
data <- data.frame(
    treatment = factor(c(rep("A", 10), rep("B", 10), rep("C", 10))),
    response = rnorm(30)
)
```

```r
# ANOVA requires factors
anova_result <- aov(response ~ treatment, data = data)
summary(anova_result)
```

3. Plotting and Visualization:

```r
# Factors control order in plots
survey_data <- data.frame(
    satisfaction = factor(c("Poor", "Good", "Excellent", "Poor", "Good"),
                         levels = c("Poor", "Good", "Excellent")),
    count = c(5, 15, 25, 8, 12)
)

# Proper ordering in plots
barplot(table(survey_data$satisfaction))
```

4. Data Validation:

```r
# Factors prevent invalid values
valid_colors <- factor(c("red", "blue", "green"),
                      levels = c("red", "blue", "green", "yellow"))

# Attempting to add invalid level results in NA
# valid_colors[4] <- "purple"  # Would create NA
```

4. How can you merge two data frames in R?

Methods to Merge Data Frames:

1. Using merge() function:

```r
# Create sample data frames
df1 <- data.frame(
    ID = c(1, 2, 3, 4),
    Name = c("Alice", "Bob", "Charlie", "Diana"),
    Age = c(25, 30, 35, 28)
)

df2 <- data.frame(
    ID = c(2, 3, 4, 5),
    Department = c("HR", "IT", "Finance", "Marketing"),
    Salary = c(60000, 75000, 70000, 65000)
)

# Inner join (default)
```

```r
inner_join <- merge(df1, df2, by = "ID")
print("Inner Join:")
print(inner_join)

# Left join
left_join <- merge(df1, df2, by = "ID", all.x = TRUE)
print("Left Join:")
print(left_join)

# Right join
right_join <- merge(df1, df2, by = "ID", all.y = TRUE)
print("Right Join:")
print(right_join)

# Full outer join
full_join <- merge(df1, df2, by = "ID", all = TRUE)
print("Full Outer Join:")
print(full_join)
```

2. Merging by different column names:

```r
df3 <- data.frame(
    EmpID = c(1, 2, 3),
    Name = c("John", "Jane", "Jake")
)

df4 <- data.frame(
    EmployeeID = c(1, 2, 4),
    Position = c("Manager", "Analyst", "Director")
)

# Merge by different column names
merged_diff <- merge(df3, df4, by.x = "EmpID", by.y = "EmployeeID")
print("Merge by different columns:")
print(merged_diff)
```

3. Using rbind() for vertical merging:

```r
# Combine rows (same column structure)
df_part1 <- data.frame(
    Name = c("A", "B"),
    Score = c(85, 90)
)

df_part2 <- data.frame(
```

```r
    Name = c("C", "D"),
    Score = c(78, 92)
)

vertical_merge <- rbind(df_part1, df_part2)
print("Vertical merge (rbind):")
print(vertical_merge)
```

4. Using cbind() for horizontal merging:

```r
# Combine columns (same number of rows)
df_names <- data.frame(Name = c("Alice", "Bob", "Charlie"))
df_ages <- data.frame(Age = c(25, 30, 35))
df_cities <- data.frame(City = c("Delhi", "Mumbai", "Bangalore"))

horizontal_merge <- cbind(df_names, df_ages, df_cities)
print("Horizontal merge (cbind):")
print(horizontal_merge)
```

5. What is the apply() function in R, and how does it work with data frames?

The apply() Function: The apply() function applies a function over the margins of an array or matrix. For data frames, it's used to apply functions across rows or columns.

Syntax:

```r
apply(X, MARGIN, FUN, ...)
# X: array, matrix, or data frame
# MARGIN: 1 for rows, 2 for columns
# FUN: function to apply
# ...: additional arguments to FUN
```

Examples with Data Frames:

1. Basic apply() usage:

```r
# Create sample data frame
student_scores <- data.frame(
    Math = c(85, 92, 78, 88, 95),
    Science = c(80, 85, 90, 82, 88),
    English = c(88, 78, 85, 90, 82),
    History = c(82, 88, 80, 85, 90)
)
rownames(student_scores) <- c("Alice", "Bob", "Charlie", "Diana", "Eve")

print("Student Scores:")
```

```r
print(student_scores)

# Apply function to rows (calculate average for each student)
student_averages <- apply(student_scores, 1, mean)
print("Student Averages:")
print(student_averages)

# Apply function to columns (calculate average for each subject)
subject_averages <- apply(student_scores, 2, mean)
print("Subject Averages:")
print(subject_averages)
```

2. Different functions with apply():

```r
# Calculate various statistics
row_sums <- apply(student_scores, 1, sum)
row_max <- apply(student_scores, 1, max)
row_min <- apply(student_scores, 1, min)
row_sd <- apply(student_scores, 1, sd)

# Column statistics
col_var <- apply(student_scores, 2, var)
col_median <- apply(student_scores, 2, median)

print("Row sums:")
print(row_sums)
print("Column variances:")
print(col_var)
```

3. Custom functions with apply():

```r
# Custom function to calculate range
calculate_range <- function(x) {
    return(max(x) - min(x))
}

# Apply custom function
score_ranges <- apply(student_scores, 1, calculate_range)
print("Score ranges for each student:")
print(score_ranges)

# Anonymous function
# Calculate coefficient of variation
cv <- apply(student_scores, 2, function(x) sd(x)/mean(x) * 100)
```

```r
print("Coefficient of variation by subject:")
print(round(cv, 2))
```

4. apply() family functions:

```r
# lapply() - returns list
result_list <- lapply(student_scores, mean)
print("lapply result (list):")
print(result_list)

# sapply() - returns vector/matrix
result_vector <- sapply(student_scores, mean)
print("sapply result (vector):")
print(result_vector)

# mapply() - multivariate version
weights <- c(0.3, 0.25, 0.25, 0.2)  # Subject weights
weighted_scores <- mapply(function(score, weight) score * weight,
                          student_scores, weights)
print("Weighted scores:")
print(weighted_scores)
```

Moderate Questions (5 Marks each)

1. Write an R function to calculate the mean of each column in a data frame.

```r
# Function to calculate mean of each column in a data frame
calculate_column_means <- function(df, na_remove = TRUE, numeric_only = TRUE) {
    # Input validation
    if (!is.data.frame(df)) {
        stop("Input must be a data frame")
    }

    if (nrow(df) == 0) {
        stop("Data frame is empty")
    }

    # Filter numeric columns if specified
    if (numeric_only) {
        numeric_cols <- sapply(df, is.numeric)
        if (sum(numeric_cols) == 0) {
            stop("No numeric columns found in the data frame")
        }
        df_numeric <- df[, numeric_cols, drop = FALSE]
    } else {
```

```r
        df_numeric <- df
    }

    # Calculate means using different methods

    # Method 1: Using apply()
    means_apply <- apply(df_numeric, 2, function(x) {
        if (is.numeric(x)) {
            return(mean(x, na.rm = na_remove))
        } else {
            return(NA)
        }
    })

    # Method 2: Using sapply()
    means_sapply <- sapply(df_numeric, function(x) {
        if (is.numeric(x)) {
            return(mean(x, na.rm = na_remove))
        } else {
            return(NA)
        }
    })

    # Method 3: Using colMeans() for numeric data
    if (all(sapply(df_numeric, is.numeric))) {
        means_colmeans <- colMeans(df_numeric, na.rm = na_remove)
    } else {
        means_colmeans <- NULL
    }

    # Return comprehensive results
    result <- list(
        means = means_apply,
        method_used = "apply",
        original_columns = ncol(df),
        numeric_columns = ncol(df_numeric),
        column_names = names(df_numeric)
    )

    return(result)
}

# Enhanced function with additional statistics
```

```r
comprehensive_column_stats <- function(df, stats = c("mean", "median", "sd",
"var")) {
    # Input validation
    if (!is.data.frame(df)) {
        stop("Input must be a data frame")
    }

    # Get numeric columns
    numeric_cols <- sapply(df, is.numeric)
    df_numeric <- df[, numeric_cols, drop = FALSE]

    if (ncol(df_numeric) == 0) {
        stop("No numeric columns found")
    }

    # Calculate requested statistics
    results <- list()

    if ("mean" %in% stats) {
        results$mean <- sapply(df_numeric, mean, na.rm = TRUE)
    }
    if ("median" %in% stats) {
        results$median <- sapply(df_numeric, median, na.rm = TRUE)
    }
    if ("sd" %in% stats) {
        results$standard_deviation <- sapply(df_numeric, sd, na.rm = TRUE)
    }
    if ("var" %in% stats) {
        results$variance <- sapply(df_numeric, var, na.rm = TRUE)
    }

    # Convert to data frame for better presentation
    results_df <- do.call(data.frame, results)
    rownames(results_df) <- names(df_numeric)

    return(results_df)
}

# Example usage:
# Create sample data frame
set.seed(123)
sample_data <- data.frame(
    ID = 1:10,
```

```r
    Name = paste("Person", 1:10),
    Age = sample(20:60, 10),
    Height = rnorm(10, 170, 10),
    Weight = rnorm(10, 70, 15),
    Income = sample(30000:100000, 10),
    Score1 = rnorm(10, 85, 10),
    Score2 = rnorm(10, 80, 12),
    Active = sample(c(TRUE, FALSE), 10, replace = TRUE)
)

# Add some NA values for testing
sample_data$Height[c(3, 7)] <- NA
sample_data$Income[5] <- NA

print("Sample Data:")
print(sample_data)

# Test the functions
print("Column means using basic function:")
basic_means <- calculate_column_means(sample_data)
print(basic_means$means)

print("Comprehensive statistics:")
comprehensive_stats <- comprehensive_column_stats(sample_data)
print(round(comprehensive_stats, 2))

# Simple one-liner functions
simple_column_means <- function(df) {
    numeric_cols <- sapply(df, is.numeric)
    return(sapply(df[numeric_cols], mean, na.rm = TRUE))
}

print("Simple function result:")
print(round(simple_column_means(sample_data), 2))
```

2. How can you deal with scope issues when working with functions and objects in R?

Understanding Scope in R:

Scope refers to the visibility and accessibility of variables and objects within different parts of a program. R follows lexical scoping rules.

Types of Scope:

1. Global Environment vs Local Environment:

```r
# Global variable
global_var <- 100

# Function demonstrating scope
scope_demo <- function() {
    # Local variable
    local_var <- 50

    # Access global variable
    print(paste("Global variable inside function:", global_var))
    print(paste("Local variable:", local_var))

    # Modify global variable using <<-
    global_var <<- 200  # Super assignment

    return(local_var)
}

print(paste("Global variable before function:", global_var))
result <- scope_demo()
print(paste("Global variable after function:", global_var))

# Local variable not accessible outside function
# print(local_var)  # This would cause an error
```

2. Function Parameter Scope:

```r
# Demonstration of parameter scope
parameter_scope_demo <- function(x, y = 10) {
    # Parameters x and y are local to this function
    z <- x + y

    # Inner function
    inner_function <- function() {
        # Can access variables from parent function
        inner_result <- x * 2  # x is accessible here
        return(inner_result)
    }

    inner_value <- inner_function()

    return(list(sum = z, inner = inner_value))
}
```

```r
result <- parameter_scope_demo(5)
print(result)
```

Dealing with Scope Issues:

1. Using Environment Functions:

```r
# Check current environment
print("Current environment:")
print(environment())

# List objects in global environment
print("Objects in global environment:")
print(ls(envir = .GlobalEnv))

# Function to demonstrate environment manipulation
env_demo <- function() {
    # Create local variables
    local_a <- 10
    local_b <- 20

    # List objects in current function environment
    print("Objects in function environment:")
    print(ls(envir = environment()))

    # Access parent environment
    print("Parent environment:")
    print(parent.env(environment()))

    # Assign to global environment explicitly
    assign("global_from_function", local_a + local_b, envir = .GlobalEnv)
}

env_demo()
print(paste("Global variable created from function:", global_from_function))
```

2. Managing Variable Conflicts:

```r
# Variable name conflicts
x <- 100  # Global x

conflict_demo <- function(x) {  # Parameter x shadows global x
    print(paste("Parameter x:", x))

    # Access global x explicitly
```

```
    global_x <- get("x", envir = .GlobalEnv)
    print(paste("Global x:", global_x))

    # Local x
    x <- x + 50  # Modifies local parameter
    print(paste("Modified local x:", x))

    return(x)
}

result <- conflict_demo(10)
print(paste("Global x after function:", x))  # Still 100
```

3. Best Practices for Scope Management:

```
# Good practice: Explicit parameter passing
calculate_statistics <- function(data, multiplier = 1, add_constant = 0) {
    # Don't rely on global variables
    # Pass all needed values as parameters

    result <- (data * multiplier) + add_constant
    return(result)
}

# Bad practice example (avoid this)
bad_function <- function(data) {
    # Relies on global variables - bad practice
    result <- data * global_multiplier + global_constant
    return(result)
}

# Better approach: Return multiple values
comprehensive_analysis <- function(data) {
    # Perform all calculations locally
    mean_val <- mean(data, na.rm = TRUE)
    sd_val <- sd(data, na.rm = TRUE)
    median_val <- median(data, na.rm = TRUE)

    # Return structured result
    return(list(
        mean = mean_val,
        standard_deviation = sd_val,
        median = median_val,
        summary = summary(data)
```

```
    ))
}

# Example usage
test_data <- c(1, 2, 3, 4, 5, NA, 7, 8, 9, 10)
analysis_result <- comprehensive_analysis(test_data)
print(analysis_result)
```

4. Using Closures for Advanced Scope Control:

```
# Closure example - function factory
create_counter <- function(initial_value = 0) {
    count <- initial_value

    # Return a function that has access to 'count'
    function() {
        count <<- count + 1  # Modify count in parent environment
        return(count)
    }
}

# Create counter instances
counter1 <- create_counter(0)
counter2 <- create_counter(100)

print(counter1())  # 1
print(counter1())  # 2
print(counter2())  # 101
print(counter1())  # 3
print(counter2())  # 102
```

5. Debugging Scope Issues:

```
# Function to debug scope issues
debug_scope <- function() {
    # Show current environment chain
    current_env <- environment()
    level <- 0

    while (!identical(current_env, emptyenv())) {
        cat("Level", level, ": ")
        print(current_env)
        if (level == 0) {
            cat("Objects in this environment:\n")
            print(ls(envir = current_env))
```

```
        }
        current_env <- parent.env(current_env)
        level <- level + 1
        if (level > 10) break  # Prevent infinite loop
    }
}

# Function to demonstrate nested scope
nested_scope_demo <- function(a) {
    b <- a * 2

    inner_function <- function(c) {
        d <- b + c  # Can access 'b' from parent function

        innermost_function <- function() {
            # Can access all variables from parent scopes
            result <- a + b + c + d

            # Debug the scope chain
            cat("\nScope chain from innermost function:\n")
            debug_scope()

            return(result)
        }

        return(innermost_function())
    }

    return(inner_function(10))
}

final_result <- nested_scope_demo(5)
print(paste("Final result:", final_result))
```

3. Write a script to work with tables in R, including creating and applying functions.

```
# Comprehensive Table Operations Script in R

# Load required libraries
# install.packages(c("dplyr", "tidyr", "knitr"))
# library(dplyr)      # For data manipulation
# library(tidyr)      # For data reshaping
# library(knitr)      # For nice table formatting
```

```r
# ==============================================================================
# PART 1: Creating Tables and Data
# ==============================================================================

# Function to create sample sales data
create_sales_data <- function(n_records = 100) {
    set.seed(123)  # For reproducible data

    sales_data <- data.frame(
        OrderID = 1:n_records,
        CustomerID = sample(1000:9999, n_records, replace = TRUE),
        Product = sample(c("Laptop", "Phone", "Tablet", "Watch", "Headphones"),
                        n_records, replace = TRUE),
        Category = sample(c("Electronics", "Accessories"), n_records, replace =
TRUE),
        Region = sample(c("North", "South", "East", "West"), n_records, replace =
TRUE),
        SalesRep = sample(c("Alice", "Bob", "Charlie", "Diana", "Eve"),
                         n_records, replace = TRUE),
        Quantity = sample(1:10, n_records, replace = TRUE),
        UnitPrice = round(runif(n_records, 100, 2000), 2),
        Date = sample(seq(as.Date("2023-01-01"), as.Date("2023-12-31"), by =
"day"),
                     n_records, replace = TRUE),
        stringsAsFactors = FALSE
    )

    # Calculate total sales
    sales_data$TotalSales <- sales_data$Quantity * sales_data$UnitPrice

    return(sales_data)
}

# ==============================================================================
# PART 2: Table Creation Functions
# ==============================================================================

# Function to create frequency tables
create_frequency_table <- function(data, column_name) {
    if (!column_name %in% names(data)) {
        stop(paste("Column", column_name, "not found in data"))
    }
```

```r
    freq_table <- table(data[[column_name]])

    # Convert to data frame for better handling
    freq_df <- data.frame(
        Category = names(freq_table),
        Frequency = as.numeric(freq_table),
        Percentage = round(as.numeric(freq_table) / sum(freq_table) * 100, 2)
    )

    # Sort by frequency (descending)
    freq_df <- freq_df[order(freq_df$Frequency, decreasing = TRUE), ]
    rownames(freq_df) <- NULL

    return(freq_df)
}

# Function to create cross-tabulation tables
create_crosstab <- function(data, row_var, col_var) {
    if (!row_var %in% names(data) || !col_var %in% names(data)) {
        stop("One or both variables not found in data")
    }

    # Basic cross-tabulation
    crosstab <- table(data[[row_var]], data[[col_var]])

    # Add margins (totals)
    crosstab_with_margins <- addmargins(crosstab)

    # Calculate percentages
    crosstab_pct <- prop.table(crosstab) * 100

    return(list(
        counts = crosstab,
        with_margins = crosstab_with_margins,
        percentages = round(crosstab_pct, 2)
    ))
}

# Function to create summary tables
create_summary_table <- function(data, group_var, summary_var) {
    if (!group_var %in% names(data) || !summary_var %in% names(data)) {
        stop("One or both variables not found in data")
    }
```

```
    if (!is.numeric(data[[summary_var]])) {
        stop("Summary variable must be numeric")
    }

    # Manual grouping and summarization
    unique_groups <- unique(data[[group_var]])

    summary_results <- data.frame(
        Group = character(),
        Count = numeric(),
        Mean = numeric(),
        Median = numeric(),
        SD = numeric(),
        Min = numeric(),
        Max = numeric(),
        Total = numeric(),
        stringsAsFactors = FALSE
    )

    for (group in unique_groups) {
        group_data <- data[data[[group_var]] == group, summary_var]
        group_data <- group_data[!is.na(group_data)]

        if (length(group_data) > 0) {
            summary_results <- rbind(summary_results, data.frame(
                Group = group,
                Count = length(group_data),
                Mean = round(mean(group_data), 2),
                Median = round(median(group_data), 2),
                SD = round(sd(group_data), 2),
                Min = min(group_data),
                Max = max(group_data),
                Total = round(sum(group_data), 2),
                stringsAsFactors = FALSE
            ))
        }
    }

    # Sort by total descending
    summary_results <- summary_results[order(summary_results$Total, decreasing =
TRUE), ]
    rownames(summary_results) <- NULL
```

```r
    return(summary_results)
}

# ============================================================================
# PART 3: Advanced Table Manipulation Functions
# ============================================================================

# Function to analyze sales performance
analyze_sales_performance <- function(sales_data) {
    # Get unique combinations of Region and Product
    unique_combinations <- unique(sales_data[c("Region", "Product")])

    analysis_results <- data.frame(
        Region = character(),
        Product = character(),
        Total_Orders = numeric(),
        Total_Quantity = numeric(),
        Total_Revenue = numeric(),
        Avg_Order_Value = numeric(),
        Avg_Unit_Price = numeric(),
        stringsAsFactors = FALSE
    )

    for (i in 1:nrow(unique_combinations)) {
        region <- unique_combinations$Region[i]
        product <- unique_combinations$Product[i]

        subset_data <- sales_data[sales_data$Region == region & sales_data$Product
== product, ]

        if (nrow(subset_data) > 0) {
            analysis_results <- rbind(analysis_results, data.frame(
                Region = region,
                Product = product,
                Total_Orders = nrow(subset_data),
                Total_Quantity = sum(subset_data$Quantity),
                Total_Revenue = round(sum(subset_data$T
```