# Answers

## ✅ Unit 1: Introduction to R Programming

### 🔷 1. What is R and What is it Used For?

📘 **Explanation:**

- **R** is a **programming language** developed in 1993 by **Ross Ihaka and Robert Gentleman**.
- It's **free and open-source**, and mainly used for:
  - **Statistical computing**
  - **Data analysis**
  - **Data visualization**
  - **Machine learning** (yes, even that!)
- It supports **vectors, data frames, matrices**, and more.

🧠 **How to remember:**

> "R = Research and Reports" — think of R as the go-to language for anyone doing serious data work.

🔧 **Applications:**

- Academic research
- Business analytics
- Scientific studies
- Machine learning models

### 🔷 2. Key Features of R

| Feature | Description |
| --- | --- |
| 📊 **Statistical Methods** | Built-in support for regression, clustering, etc. |
| 🎨 **Graphics** | Amazing visualizations via `ggplot2`, `lattice`, etc. |
| 📦 **Packages** | 15,000+ on CRAN (like an app store for R) |
| 💻 **Platform Independent** | Runs on Windows, macOS, Linux |
| 🔗 **Integrates with Other Languages** | Works with Python, Java, C/C++ |

| Feature | Description |
| --- | --- |
| 🧠 **Efficient Data Handling** | Can handle big datasets |
| 📚 **Great Community** | Tons of documentation and tutorials |
| 📄 **Reproducible Research** | `RMarkdown` for mixing code + report writing |

🧠 **Tip:**

> Imagine R as a Swiss knife — one tool, many blades (packages/functions).

---

## 🔷 3. What is RStudio?

📘 **Explanation:**

- **RStudio** is an **IDE** (Integrated Development Environment) for R.
- Makes coding in R easier with a neat UI.
- It **does not replace R**, it helps you *use R more efficiently*.

🖥️ **Main Panels in RStudio:**

| Panel | What It Does |
| --- | --- |
| **Console (Bottom Left)** | Where your code runs |
| **Source (Top Left)** | Where you write and save your scripts |
| **Environment/History (Top Right)** | Lists your variables and past commands |
| **Files/Plots/Packages/Help (Bottom Right)** | Manages files, shows graphs, helps with packages |

🧠 **Tip:**

> Think of RStudio as your **lab desk** and R as your **equipment**.

---

## 🔷 4. How to Quit RStudio?

**3 Ways:**

1. **GUI Method (Graphical):**
   - File → Quit Session or just click the ❌.
   - It will ask: “Do you want to save the workspace?”
2. **Command Line:**
   - Type `q()` or `quit()` in the console.
   - It will ask the same save prompt.
3. **Auto-Save Settings:**

- Go to Tools → Global Options → General → “Save workspace on exit”

🧠 **Tip:**

> Always save scripts (`.R` files), not just the workspace. Scripts are reusable, workspaces aren’t reliable for sharing.

---

### 🔷 5. Difference Between R and RStudio

| Feature | R | RStudio |
|---|---|---|
| Type | Programming Language | IDE (Interface) |
| Function | Does the actual computations | Helps you write & manage code |
| Needed? | Mandatory | Optional, but highly recommended |
| Analogy | Car engine | Dashboard |

🧠 **Tip:**

> R = Brain, RStudio = Face. R does the thinking; RStudio shows the expression!

---

### 💡 Summary Flashpoints:

- **R is for data analysis**, stats, and graphs.
- **RStudio** makes using R easier.
- **Packages** are like apps that add power to R.
- **R can integrate** with Python, SQL, Java, etc.
- **Use RStudio’s 4 panels** smartly: Source, Console, Environment, and Viewer.
- Always **save scripts** (.R) not just workspace images.

---

## ✅ Unit 2: R Data Structures and Manipulation

---

### 🔷 1. Creating Variables in R

📘 **Concept:**

Variables store data in R (like numbers, text, TRUE/FALSE, etc.)

✅ **Syntax (3 Ways):**

```r
x <- 10        # Most common
y = "Hello"    # Also valid
20 -> z        # Less used, but allowed
```

- **No data types needed**; R figures it out (called dynamically typed).

🧠 **Tip:**

> Use `<-` to look like a little arrow "putting value into a box".

---

### 🔷 2. Conditional Statements (`if`, `if-else`, `if-else if-else`)

📘 **Concept:**

Used for decision making — run different code depending on condition.

✅ **Syntax:**

```
x <- 0

if (x > 0) {
  print("Positive")
} else if (x < 0) {
  print("Negative")
} else {
  print("Zero")
}
```

🧠 **Tip:**

> Think of them like **flowing rivers**: each condition guides the water (code) down a path.

---

### 🔷 3. Writing Functions (e.g., Sum of Vector Elements)

📘 **Concept:**

Functions are blocks of reusable code.

✅ **Example:**

```
sum_vector <- function(vec) {
  total <- sum(vec)
  return(total)
}

my_vector <- c(1, 2, 3)
print(sum_vector(my_vector))  # Output: 6
```

🧠 **Tip:**

> Wrap your logic inside `function()`, just like packing tools in a toolbox.

---

## 🔷 4. More Data Structures in R

Let's understand the **most common data types and structures** you'll use daily:

| Structure | Description | Example |
|---|---|---|
| **Vector** | 1D list of items (same type) | `c(1,2,3)` |
| **Matrix** | 2D (rows & columns), same type | `matrix(1:6, 2, 3)` |
| **List** | 1D, can hold mixed types | `list(1, "hi", TRUE)` |
| **Data Frame** | Table with rows/columns, different types allowed | `data.frame()` |

---

## 🔷 5. Logical Operations (used inside conditions)

| Operation | Description | Example |
|---|---|---|
| `==` | Equals | `x == 5` |
| `!=` | Not equal | `x != 5` |
| `>` `<` `>=` `<=` | Greater, less, etc. | `x > 3` |
| `&` | AND | `(x > 2 & x < 10)` |
| `` ` `` | `` ` `` | OR |

---

## 🔷 6. Loops (Just basic mention for now)

- R supports `for`, `while`, and `repeat` loops.
- Often replaced with functions like `apply()` or vectorized operations for efficiency.

---

## 🔷 Quick Examples to Practice:

### Create Variables

```
a <- 5
b <- "Data"
```

### If-Else Statement

```
if (a > 0) {
  print("a is positive")
}
```

### Sum Vector Function

```
my_sum <- function(v) {
  return(sum(v))
}
```

### 💡 Summary Flashpoints:

- **Use** `<-` to assign values to variables.
- Use `if`, `else if`, and `else` for logic/decision.
- **Vectors** are the most basic structure.
- **Functions** help reuse code. Syntax:

```
fname <- function(args) {
  # code
  return(value)
}
```

- R decides data types **automatically** (dynamic typing).

---

## ✅ Unit 3: R Packages and Functions

---

### 🔷 1. What is an R Package?

#### 📘 Concept:

An **R package** is a **bundle of functions**, data, and documentation grouped together for a specific task or domain (like a mini toolkit or app).

#### 📦 Packages Help You:

- Avoid rewriting code
- Use advanced tools created by others
- Work faster with powerful pre-built functions

#### 🧠 Analogy:

> Think of packages as **"apps" for R**. Want to draw cool charts? Install `ggplot2`. Want to clean data? Use `dplyr`.

---

### 🔷 2. How to Install and Use a Package

#### ✅ Install a Package (Once):

```
install.packages("ggplot2")
```

#### ✅ Load it into the session (Every time you start R):

```
library(ggplot2)
```

#### 🎨 GUI Alternative:

- RStudio: Tools → Install Packages → Type name → Install

📘 **Tip:**

CRAN (Comprehensive R Archive Network) is like a library of R packages (15,000+!)

---

### 🔷 3. Creating Your Own Function

📘 **Concept:**

Custom functions let you reuse logic easily and make your code neater.

✅ **Syntax:**

```
function_name <- function(parameters) {
  # body
  return(result)
}
```

✅ **Example: Square of a Number**

```
square_number <- function(x) {
  return(x^2)
}
square_number(4)  # Output: 16
```

🧠 **Tip:**

> Always define, test, and reuse your own functions to make your code modular and professional.

---

### 🔷 4. Downloading and Importing Data

📘 **Data can be loaded from:**

- A **CSV file** (local or online)
- **Excel file** (using external package)
- Built-in datasets in R

✅ **Examples:**

📂 **Import CSV from Local:**

```
data <- read.csv("C:/Users/Soham/Documents/data.csv")
```

🌐 **Import CSV from Web:**

```
data <- read.csv("https://people.sc.fsu.edu/~jburkardt/data/csv/airtravel.csv")
```

📊 **View Data:**

```r
head(data)     # Shows first 6 rows
str(data)      # Shows structure (columns & types)
```

📘 Excel Example (needs `readxl` package):

```r
install.packages("readxl")
library(readxl)
data <- read_excel("data.xlsx")
```

---

### 🔷 5. Why Use Functions?

📘 **Benefits:**

- **Avoid repetition**
- **Easier debugging**
- **Cleaner code**
- Can pass any number of arguments
- Helps with big projects or teamwork

✅ **Example: Multiply Two Numbers**

```r
multiply <- function(a, b) {
  return(a * b)
}
multiply(4, 5)  # Output: 20
```

---

### 🔷 Bonus: Types of Functions in R

| Type | Description |
|---|---|
| Built-in | Predefined in R (`sum()`, `mean()`, `sqrt()`, etc.) |
| User-defined | You write them (`function_name <- function(...) {}`) |
| Anonymous (lambda) | Functions without names, used inline |
| Recursive | A function that calls itself (like `factorial`) |

---

### 💡 Summary Flashpoints:

- **Packages = mini software toolkits** that extend R's functionality.
- Use `install.packages()` once, `library()` every time.
- **Functions = reusable logic blocks**.
- Syntax for functions:

```r
name <- function(input) {
  # body
```

```
  return(output)
}
```

- Use `read.csv()` or `read_excel()` to **load datasets** into R.

---

### 🧠 How to Memorize This Unit:

- **"PRUF"** — **P**ackages, **R**eusable code, **U**ser-defined functions, **F**ile import
- Remember the pattern: define → call → reuse
- Make a habit: Whenever you repeat code twice, make it a function.

---

## ✅ Unit 4: Matrices, Arrays, and Lists

---

### 🔷 1. Matrix in R

#### 📘 Concept:

A **matrix** is a **2D rectangular** structure of elements *all of the same type* (like numbers).

#### ✅ Syntax:

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

#### 🔁 Output:

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

#### 📌 Note:

- Filled **column-wise** by default
- Data types must be the same (numeric, logical, etc.)

---

### 🔷 2. Matrix Operations

#### ➕ Arithmetic:

```
mat1 + mat2    # element-wise addition
mat1 * mat2    # element-wise multiplication
mat1 %*% mat2  # matrix multiplication
```

#### 🔁 Transpose:

```
t(mat)         # swaps rows and columns
```

### 🔍 Access Elements:

```
mat[1,2]       # element at row 1, column 2
mat[,2]        # entire 2nd column
```

### ➕ Add Rows / Columns:

```
rbind(mat, c(7,8,9))   # adds new row
cbind(mat, c(10,11))   # adds new column
```

### ❌ Remove Rows / Columns:

```
mat[-2, ]     # removes 2nd row
mat[ , -1]    # removes 1st column
```

---

## 🔷 3. Vector vs Matrix

| Feature | Vector | Matrix |
|---|---|---|
| Dimension | 1D | 2D |
| Indexing | `v[2]` | `m[1,2]` |
| Data Type | Same | Same |
| Creation | `c()` | `matrix()` |

### 🧠 Tip:

> Matrix = Vector + Shape (rows × columns)

---

## 🔷 4. Arrays in R (Higher Dimensional)

### 📘 Concept:

An **array** is an extension of a matrix — it can have **more than 2 dimensions**.

### ✅ Syntax:

```
arr <- array(1:24, dim = c(3, 4, 2))  # 3 rows, 4 cols, 2 layers
print(arr)
```

### 🔍 Accessing Elements:

```
arr[1, 2, 1]   # Row 1, Col 2, Layer 1
```

### 💡 Use Cases:

- Image data (pixels: x, y, color)
- Time series by location
- Multidimensional simulations

🧠 **Tip:**

> Think of arrays as **"matrices stacked like pages in a book"**.

---

## 🔷 5. Lists in R

### 📘 Concept:

A **list** is a flexible structure that can hold elements of **different types** — even vectors, functions, or other lists.

### ✅ Syntax:

```
my_list <- list(name = "Soham", marks = c(80, 90), pass = TRUE)
print(my_list)
```

### 🔍 Accessing Elements:

```
my_list$name       # using name
my_list[[2]]       # using position
```

### 📘 Why Use Lists?

- Useful in returning multiple results from a function
- Storing mixed-type data (e.g., info about a student: name, age, grades, etc.)

---

### 💡 Summary Flashpoints:

| Structure | Type | Key Function | Special Feature |
|---|---|---|---|
| Matrix | Homogeneous, 2D | `matrix()` | Row/Col operations |
| Array | Homogeneous, 3D+ | `array()` | Multi-dim data |
| List | Heterogeneous | `list()` | Can store anything |

---

### 🧠 Memory Tip:

- **Matrix** = "Excel table"
- **Array** = "Stack of Excel sheets"
- **List** = "Drawer with mixed items"

---

### ✅ Practice Snippet:

```
# Matrix Example
m <- matrix(1:6, nrow=2)
m <- cbind(m, c(7, 8))  # add column
print(m[1, 3])          # access element
```

```r
# Array Example
arr <- array(1:12, c(2, 2, 3))
print(arr[2, 1, 2])     # access 2nd row, 1st col, 2nd layer

# List Example
lst <- list(name="Ana", score=95, pass=TRUE)
print(lst$name)
```

---

## ✅ Unit 5: Data Frames

---

### 🔷 1. What is a Data Frame?

📘 **Concept:**

A **data frame** is a **tabular structure** (like an Excel sheet) where:

- Each **column** can have a **different data type** (numeric, character, logical, etc.)
- Each **row** represents a single observation/record

✅ **Example:**

```r
df <- data.frame(Name = c("John", "Alice"), Age = c(23, 25), Passed = c(TRUE, FALSE))
print(df)
```

🔍 **Output:**

```
   Name Age Passed
1  John  23   TRUE
2 Alice  25  FALSE
```

🧠 **Tip:**

> Data frame = **Matrix + Flexibility** (each column can be of a different type)

---

### 🔷 2. Creating a Data Frame

✅ **Code:**

```r
names <- c("Sam", "Lily")
ages <- c(22, 24)
grades <- c("A", "B")
df <- data.frame(Name = names, Age = ages, Grade = grades)
```

🔍 **Check Structure:**

```
str(df)
```

---

## 🔷 3. Accessing Data in a Data Frame

| Method | Purpose | Example |
|---|---|---|
| `$` | Access column | `df$Name` |
| `df[row, col]` | Access by index | `df[1,2]` (row 1, column 2) |
| `df[ , col]` | Entire column | `df[, 1]` |
| `df[row, ]` | Entire row | `df[1, ]` |

### 🧠 Tip:

> `$` is like calling a field by its name (like `df$Age` gets you the whole Age column)

---

## 🔷 4. Factors in R

### 📘 Concept:

**Factors** represent **categorical data**. Internally, they are stored as numbers with labels.

### ✅ Example:

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

### 🔍 Output:

```
[1] Male   Female Male   Female
Levels: Female Male
```

### 📘 Why Important?

- Used in statistical modeling (like regression)
- Helps reduce memory (since categories are stored efficiently)

---

## 🔷 5. Merging Two Data Frames

### ✅ Example:

```
df1 <- data.frame(ID = c(1, 2), Name = c("Alex", "Brian"))
df2 <- data.frame(ID = c(1, 2), Score = c(85, 90))
merged_df <- merge(df1, df2, by = "ID")
```

### 🔍 Output:

```
  ID  Name Score
1  1  Alex    85
```

```
2  2 Brian    90
```

🧠 **Tip:**

> `merge()` is like **joining two tables using a common key column** (e.g., ID)

---

## 🔷 6. Using `apply()` with Data Frames

📘 **Concept:**

`apply()` is used to apply a function (like `mean`, `sum`) to rows or columns.

✅ **Syntax:**

```
apply(X, MARGIN, FUN)
```

- `X`: data frame or matrix
- `MARGIN = 1` → rows
- `MARGIN = 2` → columns

✅ **Example:**

```
df <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6))
apply(df, 2, mean)   # Mean of each column
```

🔍 **Output:**

```
A B
2 5
```

---

## 🔷 7. Writing Functions on Data Frames

✅ **Example: Mean of Each Column**

```
mean_columns <- function(df) {
  return(apply(df, 2, mean))
}

data <- data.frame(x = c(1, 2, 3), y = c(4, 5, 6))
print(mean_columns(data))   # Output: x = 2, y = 5
```

---

## 🔷 8. Scope Issues in Functions

📘 **Problem:**

Same variable name inside and outside a function → confusion

✅ **Example:**

```r
a <- 10
myfunc <- function() {
  a <- 5
  print(a)  # prints local a
}
myfunc()
print(a)      # prints global a (10)
```

🧠 **Global Assignment:**

Use `<<-` to assign to global from inside a function:

```r
f <- function() {
  x <<- 100
}
```

---

## 🔷 9. Working with Tables and Applying Functions

✅ **Full Script:**

```r
data <- data.frame(Name = c("Tom", "Jerry", "Spike"), Marks = c(80, 90, 85))

# Frequency table: how many scored > 85
table(data$Marks > 85)

# Grade classification function
grade_function <- function(score) {
  if (score >= 90) return("A")
  else if (score >= 80) return("B")
  else return("C")
}

# Add Grade column
data$Grade <- sapply(data$Marks, grade_function)
print(data)
```

---

## 💡 Summary Flashpoints:

- **Data frame** = table with **columns of different types**
- `data.frame()` creates data frames
- Use `merge()` to join data frames on a column
- `apply(df, 2, FUN)` applies functions to each column
- Use `$` or indexing to access parts of a data frame
- **Factors** are useful for categorical data

- `sapply()` is used to apply a function to each element in a vector or column

---

### 🧠 Memory Tip:

- Data frame = "Excel sheet"
- `apply()` → Apply function to rows/columns
- `merge()` → Merge/join tables
- `$` → Access column by name
- `factor()` → Use when working with **categories** (like Male/Female)

---