UNIT-II

Factors and Data Frames: Introduction to Factors: Factor Levels, Summarizing a Factor, Ordered Factors, Comparing Ordered Factors, **Introduction to Data Frame**, Subsetting of Data Frames, Extending Data Frames, Sorting Data Frames. **Lists:** Introduction, Creating a List: Creating a Named List, Accessing List Elements, Manipulating List Elements, Merging Lists, Converting Lists to Vectors. **Conditionals and Control Flow:** Relational Operators, Relational Operators and Vectors, Logical Operators, Logical Operators and Vectors, Conditional Statements.

Types of Variables in R Programming

In **R**, a **variable** is a name used to store data values.
R is a **dynamically typed language**, meaning the **type of a variable is determined by the value assigned to it**, not declared explicitly.

Classification of Variables in R

Variables in R are commonly classified into **two main ways**:

1. **Based on Data Nature (Statistical Classification)**
2. **Based on R Data Types (Programming Classification)**

1. Types of Variables Based on Data Nature

This classification is widely used in **statistics and data analysis**.

1.1 Qualitative (Categorical) Variables

These variables represent **non-numeric categories**.

Types of Categorical Variables

a) Nominal Variables

- Categories with **no order**
- Only names or labels

Examples:

- Gender (Male, Female)
- Blood Group (A, B, AB, O)
- Department (CSE, ECE)

In R: Stored using **factors (unordered)**

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

---

b) Ordinal Variables

- Categories with a **meaningful order**

Examples:

- Grades (A, B, C)
- Ratings (Low, Medium, High)
- Performance (Poor → Excellent)

In R: Stored using **ordered factors**

```
grade <- factor(c("A", "B", "C"),
            levels = c("C", "B", "A"),
            ordered = TRUE)
```

---

2. Types of Variables Based on R Data Types

This is the **programming-level classification**.

2.1 Numeric Variables

Used to store **numbers**.

Types:

---

a) Integer

- Whole numbers
- Represented with **L**

```
x <- 10L
typeof(x)
```

aOutput: "integer"

---

b) Double (Numeric)

- Decimal or real numbers
- Default numeric type in R

```
y <- 10.5
typeof(y)
```

Output: "double"

2.2 Character Variables

Used to store **text or strings**.

```
name <- "R Programming"
typeof(name)
```

Output: "character"

2.3 Logical Variables

Used to store **Boolean values**.

```
result <- TRUE
typeof(result)
```

Output: "logical"

2.4 Factor Variables

Used to store **categorical data** efficiently.

```
dept <- factor(c("CSE", "ECE", "CSE"))
typeof(dept)
```

Output: "integer" (internally)

But class is "factor"

2.5 Complex Variables

Used to store **complex numbers**.

```
z <- 3 + 4i
typeof(z)
```

Output: "complex"

2.6 Raw Variables

Used to store **raw bytes**.

```
r <- charToRaw("R")
typeof(r)
```

Output: "raw"

---

3. Special Variable Types in R

3.1 NULL

Represents **no value**.

```
x <- NULL
```

3.2 NA (Missing Value)

Represents **missing or undefined data**.

```
x <- c(10, NA, 20)
is.na(x)
```

3.3 NaN (Not a Number)

```
0/0
```

Output: NaN

3.4 Inf and -Inf

```
1/0     # Inf
-1/0    # -Inf
```

---

4. Variable Scope in R (Brief)

Scope Type	Meaning
Local Variable	Defined inside a function
Global Variable	Defined outside a function

---

5. Summary Table

Variable Type	Description	Example
Integer	Whole numbers	10L
Numeric	Decimal numbers	10.5
Character	Text	"R"
Logical	TRUE/FALSE	TRUE
Factor	Categorical	factor()
Complex	Complex numbers	2+3i
Raw	Binary data	raw()

Factors

In **R**, a **factor** is a special data structure used to handle **categorical (qualitative) data** such as gender, department, grade, status, or rating.

Unlike numeric or character vectors, factors store data as **integer codes with associated category labels called levels**.

1. What is a Factor?

A **factor** is a data type that represents **categorical variables**.

Example of Categorical Data

- Gender: Male, Female
- Grade: A, B, C
- Status: Pass, Fail
- Department: CSE, ECE, MECH

R treats such data efficiently using **factors**.

---

2. Why Factors are Important in R

Factors are important because:

- They **save memory** by storing categories as integers.
- They are **required in statistical models** (ANOVA, regression).
- They help R understand **grouping information**.
- They control how data is handled in **plots and summaries**.

---

3. Creating Factors in R

Syntax

```
factor(x, levels, labels, ordered = FALSE)
```

Simple Example

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

Output

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

By default, **levels are sorted alphabetically**.

---

4. Internal Working of Factors

Internally, R stores factors as **integers** with level labels.

```
as.numeric(gender)
```

Output

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

Female = 1, Male = 2

This makes factors **memory-efficient**.

---

5. Levels in Factors

What are Levels?

Levels are the **unique categories** present in a factor.

Checking Levels

```
levels(gender)
```

Output

```
[1] "Female" "Male"
```

Setting Custom Levels

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

Output

```
[1] "Male" "Female"
```

Renaming Levels

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

Output

```
[1] M F M F
Levels: M F
```

---

6. Creating Factors from Numeric Data

```
marks <- c(45, 78, 90, 60)
result <- factor(ifelse(marks >= 50, "Pass", "Fail"))
result
```

Output

```
[1] Fail Pass Pass Pass
Levels: Fail Pass
```

---

7. Summarizing Factors

Using summary()

```
summary(gender)
```

Output

```
M F
2 2
```

Using table()

```
table(gender)
```

Output

```
gender
M F
2 2
```

Both show **frequency counts**.

---

8. Ordered Factors

What is an Ordered Factor?

An **ordered factor** is a factor where the **levels have a meaningful order**.

Examples

- Low < Medium < High
- Poor < Average < Good < Excellent

Creating an Ordered Factor

```
rating <- factor(
  c("Low", "Medium", "High", "Medium"),
  levels = c("Low", "Medium", "High"),
  ordered = TRUE
)
rating
```

Output

```
[1] Low    Medium High   Medium
Levels: Low < Medium < High
```

Checking Order

```
is.ordered(rating)
```

Output

```
[1] TRUE
```

---

9. Comparing Factors

Unordered Factor Comparison (Invalid)

```
gender[1] < gender[2]
 Error: not meaningful
```

Ordered Factor Comparison (Valid)

```
rating[1] < rating[2]
TRUE
```

```
rating[3] > rating[2]
TRUE
```

---

10. Converting Factors

Factor to Character Demonstration

```
as.character(gender)
```

Factor to Numeric (Correct Way)

```
as.numeric(as.character(factor(c("1", "2", "3"))))
```

⚠ Avoid direct as.numeric() on factors.

---

11. Factors in Data Frames

```
students <- data.frame(
  Name = c("A", "B", "C"),
  Gender = factor(c("Male", "Female", "Male"))
)
str(students)
```

Output

```
'data.frame': 3 obs. of 2 variables:
$ Name  : chr
$ Gender: Factor w/ 2 levels "Female","Male": 2 1 2
```

---

12. Difference Between Character and Factor

Feature	Character	Factor
Data Type	Text	Categorical
Memory Efficient	❌ No	✓ Yes
Levels	❌ No	✓ Yes
Statistical Use	❌ Limited	✓ Important

---

13. Advantages of Factors

Efficient storage
Better statistical modeling

Clear category representation
Required for grouped analysis

---

14. Disadvantages of Factors

Difficult numeric conversion
Errors if levels are incorrect

DataFrames

1. Introduction to Data Frame

A **data frame** is one of the most important data structures in R. It is used to store **tabular data**, similar to a table in a database, spreadsheet (Excel), or CSV file.

Definition

A **data frame** is a collection of **vectors of equal length**, where:

- Each **column** can be of a different data type (numeric, character, factor, logical).
- Each **row** represents an observation.
- Each **column** represents a variable.

  Explanation

  Each column contains one variable:

  - RollNo → numeric variable
  - Name → character variable
  - Marks → numeric variable
- **Each row contains one set of values from each column**:
  - Row 1 → (101, Anil, 85)
  - Row 2 → (102, Bala, 90)
  - Row 3 → (103, Charan, 88)

  So, the student data frame is a **two-dimensional table** where columns represent variables and rows represent observations.

Key Characteristics

- Two-dimensional structure (rows and columns)
- Column-oriented data storage
- Column names and row names are allowed
- Most commonly used structure for statistical analysis

Creating a Data Frame

```
student <- data.frame(
  ID = c(1, 2, 3),
  Name = c("Ravi", "Sita", "Anil"),
  Marks = c(85, 90, 78),
  Passed = c(TRUE, TRUE, FALSE)
)

student
```

Checking Data Frame Properties

```
class(student)     # data.frame
str(student)       # structure
dim(student)       # dimensions (rows, columns)
names(student)     # column names
```

---

2. Subsetting of Data Frames

Subsetting means **extracting specific rows, columns, or elements** from a data frame.

General Syntax

```
dataframe[row, column]
```

2.1 Selecting Columns

Using Column Names

```
student$Name
student[ , "Marks"]
```

Using Column Index

```
student[ , 2]
```

Multiple Columns

```
student[ , c("Name", "Marks")]
student[ , c(2, 3)]
```

2.2 Selecting Rows

By Row Number

```
student[1, ]

student[1:2, ]
```

By Condition

```
student[student$Marks > 80, ]
```

2.3 Selecting Specific Elements

```
student[2, 3]    # 2nd row, 3rd column
```

2.4 Using subset() Function

```
subset(student, Marks > 80)

subset(student, select = c(Name, Marks))
```

---

3. Extending Data Frames

Extending a data frame means **adding new columns or rows**.

3.1 Adding a New Column

Using $ Operator

```
student$Grade <- c("A", "A+", "B")

student
```

Using cbind()

```
Age <- c(20, 21, 22)

student <- cbind(student, Age)
```

3.2 Adding a New Row

Using rbind()

```
new_student <- data.frame(

  ID = 4,

  Name = "Meena",

  Marks = 88,

  Passed = TRUE,

  Grade = "A",

  Age = 21
)

student <- rbind(student, new_student)
```

Important Rule: Column names and data types must match.

3.3 Modifying Existing Values

```
student$Marks[3] <- 82
student
```

---

4. Sorting Data Frames

Sorting means arranging rows in **ascending or descending order** based on one or more columns.

4.1 Sorting Using order()

Ascending Order

```
student_sorted <- student[order(student$Marks), ]
student_sorted
```

Descending Order

```
student_sorted <- student[order(-student$Marks), ]
```

4.2 Sorting by Multiple Columns

```
student[order(student$Passed, -student$Marks), ]
```

4.3 Sorting Using with()

```
student[with(student, order(Marks)), ]
```

4.4 Sorting Using dplyr (Optional)

```
library(dplyr)

arrange(student, Marks)
arrange(student, desc(Marks))
```

---

5. Difference: Data Frame vs Matrix

Feature	Data Frame	Matrix
Data types	Different allowed	Same only
Column names	Allowed	Allowed
Usage	Statistical analysis	Mathematical operations
Flexibility	High	Low

---

6. Important Exam Points (One-Mark / Short Answers)

- A data frame is a **list of equal-length vectors**
- Use data.frame() to create a data frame
- Subsetting uses [row, column]
- Use $ to access columns
- Use rbind() and cbind() to extend data frames
- Use order() to sort data frames

Lists

1. Introduction to Lists

A **list** in R is a **heterogeneous data structure**, meaning it can store **different types of elements** (numbers, characters, vectors, matrices, data frames, or even other lists) in a single object.

Unlike vectors, lists **do not require all elements to be of the same type**.

Syntax:

```
list()
```

---

2. Creating a List

a) Simple List

```
my_list <- list(10, "R Programming", TRUE)

my_list
```

---

3. Creating a Named List

In a **named list**, each element is assigned a name for easy access.

```
student <- list(
  name = "Ravi",
  age = 20,
  marks = c(85, 90, 88),
  passed = TRUE
)
student
```

Names improve **readability and accessibility**.

---

4. Accessing List Elements

a) Using Index [[ ]]

Returns the **actual element**.

```
student[[1]]      # Access by position
student[["name"]] # Access by name
```

b) Using $ Operator

Used for **named lists only**.

```
student$name
student$marks
```

c) Using [ ]

Returns a **sublist**, not the element.

```
student[1]
```

---

5. Manipulating List Elements

a) Modifying an Element

```
student$age <- 21
```

b) Adding a New Element

```
student$department <- "CSE"
```

c) Removing an Element

```
student$passed <- NULL
```

---

6. Merging Lists

Two or more lists can be merged using the **c() function**.

```
list1 <- list(a = 1, b = 2)
list2 <- list(c = 3, d = 4)

merged_list <- c(list1, list2)
merged_list
```

Elements retain their **names and order**.

---

7. Converting Lists to Vectors

A list can be converted into a vector **only if all elements are compatible**.

Using unlist()

```
num_list <- list(1, 2, 3, 4)
vec <- unlist(num_list)
vec
```

If elements are of different types, R performs **type coercion**.

```
mixed_list <- list(1, "R", TRUE)
unlist(mixed_list)
```

Key Differences: List vs Vector

Feature	Vector	List
Data Type	Same	Different
Indexing	[ ]	[ ], [[ ]], $
Structure	Simple	Complex

Conditionals and Control Flow

1. Introduction

Conditionals and Control Flow allow an R program to:

- Make **decisions**
- Execute different blocks of code based on **conditions**
- Control the **order of execution**

In R, decisions are based on **logical values**:
TRUE or FALSE

2. Relational Operators in R

Definition

Relational operators are used to **compare two values**.
The result is always a **logical value** (TRUE or FALSE).

List of Relational Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Example

```
a <- 10

b <- 5


a > b

a == b

a != b
```

Output:

```
TRUE

FALSE

TRUE
```

3. Relational Operators with Vectors

Vectorized Comparison

R performs **element-wise comparison** on vectors.

```
x <- c(2, 4, 6)

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

```
x > y
```

Output:

```
FALSE FALSE TRUE
```

Recycling Rule

If one vector is shorter, R repeats (recycles) it.

```
x <- c(10, 20, 30)
y <- 15

x >= y
```

Output:

```
FALSE TRUE TRUE
```

Warning occurs if lengths are incompatible.

---

4. Logical Operators in R

Definition

Logical operators are used to **combine or negate conditions**.

Logical Operators

Operator	Meaning
&	AND (element-wise)
!	NOT
&&	AND (first element only)

Example

```
x <- TRUE
y <- FALSE

x & y
x | y
!x
```

Output:

```
FALSE
TRUE
FALSE
```

---

5. Logical Operators with Vectors

Element-wise Logical Operations

```
a <- c(TRUE, FALSE, TRUE)
b <- c(FALSE, FALSE, TRUE)

a & b
a | b
```

Output:

```
FALSE FALSE TRUE
TRUE  FALSE TRUE
```

Short-Circuit Operators (&&, ||)

Used mainly in if statements.

```
a <- c(TRUE, FALSE)
b <- c(FALSE, TRUE)
a && b
```

Output:

```
FALSE
```

(Only first elements are evaluated)

---

6. Conditional Statements in R

6.1 if Statement

Executes code **only when condition is TRUE**.

```
x <- 10
if (x > 5) {
  print("x is greater than 5")
}
```

6.2 if-else Statement

Provides **two-way decision making**.

```
x <- 3
if (x > 5) {
  print("Greater than 5")
} else {
  print("Less than or equal to 5")
}
```

6.3 else if Ladder

Used for **multiple conditions**.

```
marks <- 82

if (marks >= 90) {
  grade <- "A"
} else if (marks >= 75) {
  grade <- "B"
} else if (marks >= 60) {
  grade <- "C"
} else {
  grade <- "Fail"
}

grade
```

7. Vectorized Conditional: ifelse()

Purpose

ifelse() works **element-wise on vectors**.

Syntax

```
ifelse(condition, value_if_true, value_if_false)
```

Example

```
marks <- c(35, 45, 75)

result <- ifelse(marks >= 40, "Pass", "Fail")
result
```

Output:

```
"Fail" "Pass" "Pass"
```

---

8. switch() Statement

Purpose

Used for **menu-based selection**.

Example

```
choice <- 3
switch(choice,
       "Addition",
       "Subtraction",
       "Multiplication",
       "Division")
```

Output:

```
"Multiplication"
```

---

9. Important Notes (Exam Points)

- if works only with **single logical value**
- ifelse() is used for **vectors**
- &, | → vector comparisons
- &&, || → single comparison
- R follows **vectorization and recycling**