📘 Phase 1: Foundations of R Programming

🔹 Part 1: Introduction to R

📌 What is R?

- R is a programming language and software environment used for statistical computing, data analysis, and visualization.
- It is widely used in data science, bioinformatics, economics, and more.
- R is open-source and has a massive community.

📌 Why Learn R?

- It is built for statistics and data analysis.
- Easy to create visualizations.
- Powerful packages for machine learning and modeling.
- Trusted by professionals in academia and industry.

📌 Tools You Need:

1. **R** - the core programming language.
2. **RStudio** - the most popular IDE (Integrated Development Environment) for R.

👉 Download from:

- R: https://cran.r-project.org
- RStudio: https://posit.co/download/rstudio-desktop/

🔹 Part 3: Your First R Script

Let's write your first line of code in R:

R code:

```
print("Hello, R!")
```

This prints "Hello, R!" to the console.

🔹 Part 4: Variables and Data Types in R

📌 What is a Variable?

- A variable is like a container that stores a value.
- You assign a value to a variable using the `<-` operator (the preferred way in R).

Example:

R code:

```
name <- "Akil"
age <- 22
```

Here, `name` stores a string, and `age` stores a number.

📌 Basic Data Types in R:

Data Type	Example	Description
Numeric	`3.14, 100`	Decimal or whole numbers
Integer	5L	Whole numbers with `L`
Character	"text"	Text or strings
Logical	`TRUE, FALSE`	Boolean values

📌 **Example**:

R code:

```
num <- 10.5          # numeric
count <- 5L          # integer
word <- "hello"      # character
flag <- TRUE         # logical
```

🔹 Part 5: Basic Operations in R

📌 Arithmetic Operations

You can do basic math just like a calculator:

R code:

```
a <- 10
b <- 3

add <- a + b         # 13
sub <- a - b         # 7
mul <- a * b         # 30
div <- a / b         # 3.333...
mod <- a %% b        # 1 (remainder)
pow <- a ^ b         # 1000
```

🧠 Expert Insight:

- Always use `<-` for assignment instead of `=`, especially in scripts. It helps avoid confusion with function arguments.
- Use `typeof()` to check the type of any variable:

R code:

```
typeof(a)
```

⚠️ Common Mistakes to Avoid:

1. Using = instead of <- for variable assignment.
2. Forgetting to wrap text in quotes (" " or ' ').
3. Confusing numeric with integer (5 vs 5L).

🔹 Part 6: Vectors in R

📌 What is a Vector?

- A **vector** is a basic data structure in R that contains elements of the same type.
- Think of it like a list of numbers or words.

R code:

```
numbers <- c(1, 2, 3, 4, 5)
names <- c("Akil", "Rafi", "Nayeem")
```

Use `c()` to combine values into a vector.

📌 Indexing Vectors

You can access specific elements using square brackets `[]`:

R code:

```
numbers[1]      # First element: 1
numbers[3]      # Third element: 3
```

📌 Vector Operations

You can do math operations on vectors:

R code:

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)

sum <- a + b      # Element-wise addition: c(5, 7, 9)
```

🧠 Expert Insight:

- Vectors are the building blocks of data frames, matrices, and more.
- R performs **vectorized operations**, meaning it's optimized to process entire vectors without loops.

⚠️ Common Mistakes:

- Mixing data types (like numbers and strings) in a vector—R will convert everything to **character**.

R code:

```
mix <- c(1, "a")  # becomes character vector
```

🔹 Part 7: Creating Sequences in R

📌 Sequence with: Operator

The simplest way to make a number sequence:

R code:

```
seq1 <- 1:10
```

☑ Creates a vector: `1 2 3 4 5 6 7 8 9 10`

📌 Sequence with `seq()` Function

For more control:

R code:

```
seq2 <- seq(from = 1, to = 10, by = 2)
```

☑ Creates: `1 3 5 7 9`

🔹 Part 8: Repeating Values

Use `rep()` to repeat values.

R code:

```
rep1 <- rep(5, times = 3)              # 5 5 5
rep2 <- rep(c(1, 2), times = 3)        # 1 2 1 2 1
2
rep3 <- rep(c("a", "b"), each = 2)     # a a b b
```

🔹 Part 9: Naming Vector Elements

You can name vector elements like this:

R code:

```
marks <- c(80, 90, 85)
names(marks) <- c("Math", "English", "Science")
print(marks["English"])
```

☑ **Output:** `90`

⚠️ Common Mistakes:

- Forgetting to use `c()` for combining multiple elements.
- Mixing types (number + string) leads to all elements becoming characters.
- Using incorrect spelling in named elements.

🔹 Part 10: Logical Operations in R

📌 Basic Logical Operators

Logical operations return `TRUE` or `FALSE`.

Operation	Example	Meaning
==	5 == 5	Equal to
!=	5 != 3	Not equal to
>	7 > 5	Greater than
<	2 < 9	Less than
>=	5 >= 5	Greater than or equal
<=	3 <= 2	Less than or equal

🔹 Part 11: Filtering Vectors with Conditions

Use logical conditions to extract values:

R code:

```
ages <- c(15, 20, 25, 30, 35)
adults <- ages[ages >= 18]
print(adults)  # 20 25 30 35
```

☑ Only values that are `TRUE` for the condition get selected.

🔹 Combining Conditions

You can combine multiple logical conditions:

R code:

```
ages[ages >= 20 & ages <= 30]    # AND
ages[ages < 20 | ages > 30]      # OR
```

⚠️ Common Mistakes:

- Forgetting to use square brackets `[]` for filtering.
- Using = instead of == in conditions.
- Mixing up `&` (element-wise AND) with `&&` (single test AND).

📘 Phase 2: Core Data Structures in R

🔹 Part 12: Lists

📌 What is a List?

- A **list** in R can hold different types of data (numbers, strings, vectors, even other lists).
- It's like a flexible container.

R code:

```
mylist <- list(name = "Akil", age = 22, scores = c(80, 85, 90))
```

☑ Access elements:

R code:

```
mylist$name       # "Akil"
mylist[["age"]]   # 22
```

🔹 Part 13: Matrices

📌 What is a Matrix?

- A **matrix** is a 2D table of data (rows & columns), where **all elements must be of the same type**.
- Create with `matrix()`:

R code:

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

☑ Output:

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

🔹 Part 14: Data Frames

📌 What is a Data Frame?

- A **data frame** is like an Excel sheet—**columns can be of different types**.
- It's the most used structure in data science.

R code:

```
df <- data.frame(
  Name = c("Akil", "Rafi"),
  Age = c(22, 23),
  Score = c(90, 88)
)
```

☑ Access:

R code:

```
df$Name          # Column
df[1, ]          # First row
df[ ,2]       # Second column
```

⚠️ Common Mistakes:

- Mixing `[]`, `[[]]`, and `$` wrongly in lists.
- Forgetting that matrices require **same-type elements**.
- Trying to assign a different length vector to a data frame column.

📘 Phase 3: Functions in R

🔹 Part 15: Creating Your Own Functions

📌 Why use functions?

- Functions **reduce repetition**, improve **readability**, and make code **modular**.
- You define a function using `function()`.

R code:

```
greet <- function(name) {
  paste("Hello", name, "welcome to R!")
}
greet("Akil")
```

☑ Output: `"Hello Akil welcome to R!"`

🔹 Part 16: Function with Arguments and Return

R code:

```
add_numbers <- function(a, b) {
  result <- a + b
  return(result)
}
add_numbers(10, 20)
```

☑ Output: `30`

R code:

```
square <- function(x) {
  return(x^2)
}
square(5)
```

☑ Output: `25`

🔹 **Part 17: Default Values in Functions**

R code:

```
say_hello <- function(name = "friend") {
  paste("Hello", name)
}
say_hello()
say_hello("Akil")
```

☑ Output: `"Hello friend"` and `"Hello Akil"`

⚠️ **Common Mistakes:**

- Forgetting to use `return()` (though R often returns the last line automatically).
- Using undefined variables inside functions.
- Not providing required arguments.

📘 Phase 4: Control Structures in R

Control structures let your code **make decisions** and **repeat tasks**.

🔹 Part 18: if, else if, else

📌 Basic Structure

R code:

```
x <- 10
if (x > 0) {
  print("Positive")
} else if (x == 0) {
  print("Zero")
} else {
  print("Negative")
}
```

☑ Output: `"Positive"`

🔹 Part 19: for loop

📌 Loop through numbers

R code:

```
for (i in 1:5) {
  print(i)
}
```

🔹 Part 20: while loop

R code:

```
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1}
```

🔹 Part 21: repeat loop (infinite unless broken)

R code:

```
x <- 1
repeat {
  print(x)
  x <- x + 1
  if (x > 5) {
    break
  }
}
```

⚠️ Common Mistakes:

- Forgetting to update loop variables (infinite loop alert 🚨).
- Using = instead of == in conditions.
- Missing `{}` for multi-line blocks.

📘 Phase 5: Real-World Data Handling in R

🔹 Part 22: Built-in Datasets in R

📌 Explore Built-in Data

R has datasets built in for practice. One example is `mtcars`.

R code:

```
data(mtcars)
head(mtcars)
```

☑ See the top rows of the dataset.

🔹 Part 23: Reading CSV Files

📌 Load CSV File

Use `read.csv()` to load external files.

R code:

```
data <- read.csv("data.csv")    # if file is in working directory
head(data)
```

You can also give a full file path.

🔹 Part 24: Exploring Data

📌 Basic Exploration

R code:

```
str(data)          # structure of dataset
summary(data)      # statistical summary
nrow(data)         # number of rows
ncol(data)         # number of columns
names(data)        # column names
```

⚠️ Common Mistakes:

- Misspelling the file name or wrong file path in `read.csv()`
- Trying to read `.xlsx` with `read.csv()` (needs `readxl` package instead)
- Not checking data types after loading

📘 Phase 6: Data Cleaning & Manipulation with `dplyr`

🔹 Part 25: What is `dplyr`?

`dplyr` is one of the most powerful R packages used for data cleaning, filtering, sorting, and summarizing. It's part of the ***`tidyverse`***.

☑ First, install and load it:

`R code:`

```
install.packages("dplyr")  # Run once
library(dplyr)
```

🔹 Part 26: Select Columns

`R code:`

```
students %>% select(Name, Score)
```

☑ Only shows `Name` and `Score` columns.

🔹 Part 27: Filter Rows

`R code:`

```
students %>% filter(Score > 80)
```

☑ Shows only students who scored above 80.

🔹 Part 28: Arrange (Sort)

`R code:`

```
students %>% arrange(Score)           # Ascending
students %>% arrange(desc(Score))     #
Descending
```

🔹 Part 29: Mutate (Add New Column)

R code:

```
students %>% mutate(Grade = ifelse(Score >= 90,
"A", "B"))
```

☑ Adds a new column `Grade` based on `Score`.

🔹 Part 30: Summarize and Group

R code:

```
students %>%
  group_by(Gender) %>%
  summarize(Average = mean(Score))
```

☑ Gives average score by gender (if you have a `Gender` column).

⚠️ Common Mistakes:

- Forgetting to load `dplyr`
- Using base R syntax with `dplyr` pipelines (mixing styles)
- Column name typos (case-sensitive!)

📊 Chapter 7: Data Visualization with ggplot2

🔹 What is `ggplot2`?

`ggplot2` is a powerful and widely used R package for creating beautiful, customizable, and professional-quality data visualizations.

☑ Install and load it:

R code:

```
install.packages("ggplot2")  # Run once
library(ggplot2)
```

📘 Basic Components of `ggplot2`

🔸 Grammar of Graphics: Core Concept

`ggplot2` builds a plot in layers:

1. Data
2. Aesthetics (aes)
3. Geometries (geom)
4. Themes & Labels

📗 Chapter 8: Working with Dates and Strings in R

🔹 Part 1: Working with Dates (`lubridate` package)

📌 Why it matters:

Dates and times are common in datasets, especially in time series, logs, transactions, etc. R has a powerful package called `lubridate` that makes date handling easy and intuitive.

☑ Installation & Loading

R code:

```
install.packages("lubridate")  # Only once
library(lubridate)
```

- Common Functions in `lubridate`:

Function	Purpose
ymd()	Parse date in "YYYY-MM-DD" format
mdy()	Parse "MM-DD-YYYY" format
dmy()	Parse "DD-MM-YYYY" format
now()	Current date-time
today()	Current date only
year(), month(), day()	Extract parts

☑ Example 1: Parse a date

R code:

```
date1 <- ymd("2025-04-14")
date2 <- dmy("14-04-2025")
date3 <- mdy("04-14-2025")
```

☑ Example 2: Get today's date and extract year

R code:

```
today_date <- today()
year(today_date) # Returns 2025
```

🔹 Part 2: Working with Strings (`stringr` package)

📌 Why it matters:

String data like names, emails, IDs, etc., are very common. Cleaning or analyzing them is a vital skill.

☑ Installation & Loading

R code:

```
install.packages("stringr")  # Only once
library(stringr)
```

- Common Functions in `stringr`:

Function	Purpose
str_length()	Length of stirng
str_sub()	Substring
str_to_upper()	Convert to uppercase
str_to_lower()	Convert to lowercase
str_replace()	Replace part of string
str_detect()	Check if a pattern exists

☑ Example 1: Uppercase a name

R code:

```
name <- "akil"
str_to_upper(name)   # Output: "AKIL"
```

☑ Example 2: Extract domain from email

R code:

```
email <- "akil@example.com"
str_sub(email, start = str_locate(email, "@")[1]
+ 1)
# Output: "example.com"
```

📘 Chapter 9: Importing & Exporting Data in R

🔥 Why it is important:

You need to read data from files like CSV, Excel, or databases and write results back. R supports all major formats.

- 🔹 Common Functions:

File Type	Read Function	Write Function
CSV	read.csv()	write.csv()
Excel	readxl::read_excel()	writexl::write_xlsx()
Text	read.table()	write.table()
RDS	readRDS()	saveRDS()

☑ Example 1: Read CSV

R code:

```
data <- read.csv("data.csv")
head(data)
```

☑ Example 2: Write to CSV

R code:

```
write.csv(data, "output.csv")
```

📘 Chapter 10: Data Manipulation with dplyr

🔥 **Why this chapter is a game changer:**
This chapter teaches how to filter, sort, summarize, and transform data with elegance. `dplyr` is one of the most essential packages for any R data scientist.

🔹 Loading `dplyr`

R code:

```
install.packages("dplyr")   # Only once
library(dplyr)
```

🔸 Core `dplyr` Verbs (Functions)

Function	Purpose
filter()	Subset rows
select()	Choose columns
arrange()	Sort data
mutate()	Add new columns
summary()	Create summary statistics
group_by()	Group data before summary

☑ Example 1: Filter rows where age > 20

R code:

```
filter(students, age > 20)
```

☑ Example 2: Select name and age columns

R code:

```
select(students, name, age)
```

☑ Example 3: Arrange by marks descending

R code:

```
arrange(students, desc(marks))
```

☑ Example 4: Add a new column grade

R code:

```
mutate(students, grade = ifelse(marks >= 80,
"A", "B"))
```

☑ Example 5: Average marks by department

R code:

```
students %>%
  group_by(department) %>%
  summarise(avg_marks = mean(marks))
```

📘 Chapter 11: Data Cleaning & Handling Missing Values

🔥 Why it's essential:

Real-world data is messy. You'll often encounter missing values, duplicate records, or inconsistent formatting. This chapter teaches how to clean and prepare your data before analysis.

☑ Detecting Missing Values

R code:

```
is.na(data)          # Shows TRUE where value is missing
sum(is.na(data))     # Total number of missing values
```

☑ Removing Missing Values

R code:

```
clean_data <- na.omit(data)
```

☑ Replacing Missing Values

R code:

```
data$age[is.na(data$age)] <- mean(data$age, na.rm = TRUE)
```

☑ Removing Duplicates

R code:

```
clean_data <- distinct(data)
```

📘 Chapter 12: Exploratory Data Analysis (EDA)

🔥 **Why EDA is critical:**
EDA is all about understanding your data before **modeling. It helps you detect patterns, spot anomalies, test hypotheses, and check assumptions with summary statistics and visualizations.**

- Basic EDA Functions in R

Function	Purpose
str(data)	Structure of the dataset
summary(data)	Summary statistics
table(data$column)	Frequency of values
hist(data$marks)	Histogram of a numeric variable
boxplot(data$marks)	Boxplot to detect outliers
plot(data$marks, data$age)	Scatter plot

📘 Chapter 13: Control Flow (if, else, loops)

🔥 **Why it matters:**
Control flow lets you write logic. With `if`, `else`, and loops like `for`, `while`, you can automate decisions and repetitive tasks in your code.

- Key Concepts:

Concept	Syntax Example
if	if (x > 5) { print("Big") }
if...else	if (x > 5) { ... } else { ... }
for loop	for (i in 1:5) { print(i) }
while loop	while (x < 10) { x <- x + 1 }