Introduction to R programming :

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.

It was designed by **Ross Ihaka and Robert Gentleman** at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R programming language is an implementation of the S programming language.

Why R programming /Advantages of R programming :R is a popular programming language for a variety of reasons, particularly in the field of data science, statistical analysis, and data visualization. Here are some of the key advantages of using R:

1.Statistical Power: R was specifically designed for statistical analysis. It offers a wide range of built-in statistical functions and packages for performing complex data analysis and hypothesis testing. This makes it a go-to choice for statisticians and data analysts.

2.Data Visualization:R provides excellent data visualization capabilities through packages like ggplot2, lattice, and base graphics. You can create high-quality, publication-ready plots and charts to better understand and communicate your data.

3 Data Manipulation: R excels in data manipulation and transformation. Packages like dplyr and tidyr make it easy to filter, summarize, reshape, and clean data efficiently.

4.Rich Ecosystem: R has a vast and active community that has contributed thousands of packages to CRAN (Comprehensive R Archive Network) and other repositories. These packages extend R's functionality, covering a wide range of domains, including machine learning, time series analysis, text mining, and more.

5.Reproducibility: R's scripting capabilities make it an ideal tool for reproducible research. By creating scripts that document your data analysis steps, others can replicate your work and verify your findings.

6. Cross-Platform Compatibility R is available for multiple platforms, including Windows, macOS, and various Linux distributions, ensuring that your code can be run on different operating systems.

7.Integration with Other Languages: R can be integrated with other programming languages like C, C++, and Python. This allows you to leverage R's statistical and data analysis capabilities in conjunction with other languages for more complex tasks.

8.Open Source and Free :R is open-source software, which means it's freely available for anyone to use, modify, and distribute. This makes it a cost-effective choice for individuals and organizations.

Programming Features of R:

1. **R Packages:** One of the major features of R is it has a wide availability of libraries. R has CRAN(Comprehensive R Archive Network), which is a repository holding more than 10, 0000 packages.
2. **Distributed Computing:** Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance. Two new packages **ddR and multidplyr** used for distributed programming in R were released in November 2015.

Disadvantages of R:

1. In the R programming language, the standard of some packages is less than perfect.
2. Although, R commands give little pressure to memory management. So R programming language may consume all available memory.
3. In R basically, nobody to complain if something doesn't work.
4. R programming language is much slower than other programming languages such as Python and MATLAB.

Packages in R : R packages are collections of functions, data sets, and documentation bundled together to extend the capabilities of the R programming language. These packages cover a wide range of domains, from statistical analysis and data visualization to machine learning and text mining. The beauty of R packages is that they allow users to easily access and utilize additional functionality created by the R community. Here are some essential aspects of R packages:

1.Installation: You can install R packages from CRAN (Comprehensive R Archive Network), which is the primary repository for R packages. The installation can be done using the `install.packages()` function. For example:

```
install.packages("package_name")
```

2.Loading: After installation, you need to load a package into your R session using the `library()` function. Loading a package makes its functions and data sets available for use in your R scripts. For example:

library(package_name)

3.Package Documentation: Most packages include detailed documentation, which you can access using the `help()` function or the `?` operator. For example

help(package = "package_name") OR ?function_name

4.Popular R Packages: There are numerous R packages available for various purposes. Some popular R packages include:

-**ggplot2**: For elegant and flexible data visualization.

- **dplyr**: For data manipulation and transformation.

- **tidyr**: For data tidying and reshaping.

- **caret**: For machine learning and model training.

- **lmtest**: For testing linear regression models.

- **tm**: For text mining and natural language processing.

- **forecast**: For time series forecasting.

- **shiny**: For creating interactive web applications with R.

5.Creating Your Own Packages: If you have a collection of functions and code that you want to share with others, you can create your own R package. Tools like `devtools` and `roxygen2` make package development relatively straightforward. This allows you to organize your work and distribute it to the R community.

6.Maintaining Packages:R packages are maintained and updated by their creators or package authors. It's important to keep your packages up to date to ensure compatibility with the latest version of R and to address any potential issues.

7.Searching for Packages: To find R packages related to a specific task or domain, you can use CRAN's search engine or online resources like the RDocumentation website, which provides a comprehensive list of packages with their descriptions and documentation.

Example of Package

```
# Load the "dplyr" package for data manipulation.

# If not already installed, you can install it using install.packages("dplyr").

library(dplyr)


# Create a simple data frame with people's names, ages, and heights.

data <- data.frame(

  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),

  Age = c(25, 30, 22, 35, 28),
```

```
  Height = c(160, 175, 168, 180, 155)
)

# Print the original data frame.
cat("Original Data:\n")
print(data)

# Use "dplyr" to filter individuals aged 30 and older.
filtered_data <- data %>%
  filter(Age >= 30)

# Print the filtered data.
cat("\nFiltered Data (Ages 30 and Older):\n")
print(filtered_data)

# Use "dplyr" to calculate the average height of the selected individuals.
summary_data <- filtered_data %>%
  summarise(Average_Height = mean(Height))
```

```
# Print the summary data.

cat("\nSummary Data (Average Height):\n")

print(summary_data)
```

Data Types In R: R is a dynamically typed language, which means you don't need to explicitly declare the data type of a variable; R will automatically determine the data type based on the content. R provides several fundamental data types to work with:

1.Numeric (Numeric/Double): Numeric data types in R are used to represent real numbers, such as integers and floating-point numbers. In R, all numbers are typically stored as double-precision floating-point numbers. Example:

```
x <- 42        # integer

y <- 3.14      # floating-point
```

2.Character (Character/String): Character data types represent text or strings. Text in R is enclosed in single or double quotation marks.

```
name <- "John Doe"
```

3. Integer (Integer): In R, integers are a subset of numeric data types, and they are used to represent whole numbers without fractional parts

```
age <- 30L  # The 'L' is used to specify an integer
```

4.Logical (Boolean): Logical data types represent boolean values, which can be either `TRUE` or `FALSE`.

```
is_student <- TRUE
```

5. Complex (Complex):

Complex numbers are represented by pairs of real and imaginary parts.

```
z <- 3 + 2i
```

6.Factor (Categorical): Factors are used to represent categorical data. They are a way of encoding qualitative or nominal data with levels or categories.

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

7.Date and Time (POSIXct and POSIXlt):R has data types for working with date and time information. Two commonly used types are `POSIXct` (for date and time with a specific time zone) and `POSIXlt` (for date and time as a list).

```
current_datetime <- Sys.time()  # Current date and time
```

8. Lists: Lists are versatile data structures that can hold elements of different data types, including other lists.

```
my_list <- list(1, "apple", c(2, 4, 6), TRUE)
```

9.Vectors (Atomic Vectors):

Vectors are one-dimensional data structures that can hold multiple elements of the same data type. Common vector types include numeric, character, and logical vectors.

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

10. Matrices:

A matrix is a two-dimensional data structure in R that contains elements of the same data type. It can be considered as a special case of a data frame.

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

11. Data Frames:Data frames are tabular data structures, similar to a spreadsheet or a database table, where each column can have a different data type. They are commonly used for working with structured data.

data_frame <- data.frame(Name = c("Alice", "Bob", "Charlie"), Age = c(25, 30, 22))

12.Arrays:

Arrays are multi-dimensional data structures in R, similar to matrices but can have more than two dimensions.

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

Example

```
# Numeric data type

x <- 42

cat("Numeric:", x, "\n")
```

```
# Character data type

name <- "John Doe"

cat("Character:", name, "\n")


# Integer data type

age <- 30L

cat("Integer:", age, "\n")


# Logical data type

is_student <- TRUE

cat("Logical:", is_student, "\n")


# Complex data type

z <- 3 + 2i

cat("Complex:", z, "\n")


# Factor data type

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

```
cat("Factor:", gender, "\n")


# Date and Time data types

current_datetime <- Sys.time()

cat("Date and Time:", current_datetime, "\n")


# Lists data type

my_list <- list(1, "apple", c(2, 4, 6), TRUE)

cat("List:", my_list, "\n")


# Vectors (Atomic Vectors)

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

cat("Vector:", numbers, "\n")


# Matrices

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

cat("Matrix:\n")

print(my_matrix)
```

```
# Data Frames

data_frame <- data.frame(

  Name = c("Alice", "Bob", "Charlie"),

  Age = c(25, 30, 22)

)

cat("Data Frame:\n")

print(data_frame)


# Arrays

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

cat("Array:\n")

print(my_array)
```

Variables In R: a variable is a name that you assign to a value or an object. Variables are used to store and manipulate data, making them a fundamental concept in programming and data analysis. Here's a detailed explanation of variables in R:

1.Variable Assignment: You can assign a value to a variable using the assignment operator `<-` or the `=` sign. For example:

```
x <- 42

name <- "John"
```

Both of these statements create variables (`x` and `name`) and assign values to them.

2.Variable Names: Variable names in R are case-sensitive and can contain letters, numbers, periods, and underscores. They must start with a letter or a period followed by a letter. Variable names cannot start with a number or contain spaces.

Valid variable names:

- `age`
- `user_name`
- `.count1`

Invalid variable names:

- `1variable` (starts with a number)
- `user name` (contains a space)
- `@address` (contains an invalid character)

3.Data Types:R is a dynamically typed language, so variables can change their data type depending on the assigned value. Common data types include numeric, character, integer, logical, and others.

```
x <- 42        # Numeric data type

 name <- "John"  # Character data type

   is_student <- TRUE  # Logical data type
```

4.Printing Variables: You can print the value of a variable using the `print()` function or simply by typing the variable name in the R console. For example:

```
   print(x)

   name
```

Both of these will display the values of the `x` and `name` variables.

5.Updating Variables:

You can update the value of a variable by reassigning it:

x <- x + 1 After this, `x` will contain the updated value, which is `43`.

6.Scoping:R uses lexical scoping, which means the availability of a variable depends on where it is defined. Variables can have local or global scope. Variables defined within a function are local to that function and may not be accessible outside it. Global variables can be accessed from anywhere in the script.

7.Variable Names and Style: It's good practice to choose variable names that are descriptive and reflect the data they hold. Additionally, following a consistent naming style (e.g., using lowercase letters with underscores) can make your code more readable.

8.Removing Variables: To remove a variable and free up memory, you can use the `rm()` function. For example:

rm(x)

This will remove the variable `x`.

R has a variety of operators and keywords that are used for performing operations, controlling program flow, and manipulating data. Here's an overview of some of the most commonly used operators and keywords in R:

Operators:

1. Arithmetic Operators*

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division)
- `^` or `**` (Exponentiation)
- `%%` (Modulus, returns the remainder after division)
- `%/%` (Integer division, returns the quotient)

2.Comparison Operators:

- `==` (Equal to)

- `!=` (Not equal to)

- `<` (Less than)

- `>` (Greater than)

- `<=` (Less than or equal to)

- `>=` (Greater than or equal to)

3. Logical Operators:

- `&` (Element-wise logical AND)

- `|` (Element-wise logical OR)

- `!` (Logical NOT)

- `&&` (Short-circuit AND)

- `||` (Short-circuit OR)

4.Assignment Operators:

- `<-` or `=` (Assign a value to a variable)

- `<<-` (Assign a value in the parent environment, used within functions)

5.Special Operators:

- `%in%` (Check if an element is in a vector)

- `%*%` (Matrix multiplication)

- `:` (Create a sequence of numbers)

- `%>%` (Used in pipe chains with the `magrittr` package)

6.Concatenation Operators:

- `c()` (Combine elements into a vector)

- `paste()` (Concatenate strings)

`cat()` (Concatenate and print)

Decision-making: In R, decision-making is an essential part of programming. You can control the flow of your program by using conditional statements to make decisions based on certain conditions. Here are the primary decision-making constructs in R:

1.if Statements:The basic form of an `if` statement allows you to execute a block of code if a specified condition is true. You can also use `else` and `else if` to provide alternative actions when the condition is false.

```
x <- 10

if (x > 5) {

print("x is greater than 5")

  } else if (x == 5) {
```

```
print("x is equal to 5")
  } else {
    print("x is less than 5")
  }
```

2.switch Statements: The `switch` statement allows you to choose among multiple expressions or actions based on a condition or the value of an expression.

```
day <- "Tuesday"
result <- switch(day,
    "Monday" = "Start of the workweek",
    "Tuesday" = "Another workday",
    "Wednesday" = "Midweek",
    "Thursday" = "Almost there",
    "Friday" = "Weekend is coming",
    "Saturday" = "Weekend",
    "Sunday" = "Weekend"
  )
print(result)
```

3.for Loops: For loops allow you to execute a block of code repeatedly for a specific number of iterations. You can use them to iterate through sequences, vectors, or lists.

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

4. while Loops: While loops continue executing a block of code as long as a specified condition remains true. Be cautious to avoid infinite loops by ensuring the condition eventually becomes false.

Example:

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

5.repeat Loops: The `repeat` loop creates an infinite loop that continues executing until the `break` statement is encountered. You can use `repeat` when you need to create loops without a predetermined end condition.

Example

```
x <- 1
```

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

6.next and break Statements: Within loops, you can use the `next` statement to skip the current iteration and proceed to the next iteration. The `break` statement is used to exit a loop prematurely.

```
  for (i in 1:10) {
    if (i %% 2 == 0) {
      next  # Skip even numbers
    }
    print(i)
    if (i == 5) {
      break  # Exit the loop when i is 5
    }
```

Functions: Functions are a fundamental concept in R programming. They allow you to encapsulate a block of code into a reusable unit, which can be called with different arguments and can return a value or perform some actions. Functions in R follow a specific structure and can be user-defined or built-in. Here's an overview of functions in R:

Defining a Function: You can define your own functions in R using the `function` keyword. A basic structure of a function includes:

- The `function` keyword.

- A set of parameters that the function accepts.

- A body of the function, which contains the code to be executed.

- An optional `return()` statement to return a value.

Here's an example of a simple function that calculates the square of a number:

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

Calling a Function: To call a function, you simply use the function name followed by parentheses and provide any required arguments. For example:

```
result <- square(5)
print(result)  # This will print "25"
```

Function Parameters: Functions can accept multiple parameters (arguments), and these parameters can have default values. Parameters are defined within the function's parentheses. For example:

```
greet <- function(name, message = "Hello") {
  cat(message, name, "\n")
}
greet("Alice")              # Prints "Hello Alice"
greet("Bob", "Hi there")    # Prints "Hi there Bob"
```

Function Return: A function can return a value using the `return()` statement, but it's not always necessary. If a function doesn't include a `return()` statement, it will return the result of the last evaluated expression in the function.

```
add <- function(a, b) {
  a + b  # This will be implicitly returned
}
result <- add(3, 4)
print(result)  # This will print "7"
```

Built-in Functions:R comes with a rich set of built-in functions, covering a wide range of operations. For example, `mean()`, `sum()`, `length()`, and `paste()` are all built-in functions that perform common tasks.

```
numbers <- c(1, 2, 3, 4, 5)
```

```
mean_value <- mean(numbers)
```

Anonymous Functions:R supports anonymous (unnamed) functions, often used with functions like `apply()` and `sapply()` for applying a function to elements of a data structure. Anonymous functions are defined using the `function` keyword without assigning them a name.

```
squared_values <- sapply(numbers, function(x) x^2)
```

Function Documentation: It's good practice to document your functions using comments and documentation tools like "roxygen2" to provide information about the function's purpose, arguments, and usage.

```
#' Calculate the square of a number.

#' This function takes a numeric input and returns its square.

#' @param x Numeric value to be squared.

#' @return The square of the input.

#' @examples

#' square(5) # Returns 25

square <- function(x) {

  result <- x^2

  return(result)

}
```

Example for built in functions

```
# Create a vector of numbers
numbers <- c(2, 4, 6, 8, 10)

# Calculate the mean of the numbers
mean_value <- mean(numbers)
cat("Mean:", mean_value, "\n")

# Calculate the sum of the numbers
sum_value <- sum(numbers)
cat("Sum:", sum_value, "\n")

# Calculate the length of the vector
length_value <- length(numbers)
cat("Length:", length_value, "\n")

# Create a vector of strings
fruits <- c("apple", "banana", "cherry", "date")

# Concatenate the strings using the paste function
concatenated_string <- paste(fruits, collapse = ", ")
cat("Concatenated Fruits:", concatenated_string, "\n")
```

Example for loops

```
# for loop

cat("Using a for loop to print numbers 1 to 5:\n")

for (i in 1:5) {

cat(i, " ")

}

cat("\n\n")

# while loop

cat("Using a while loop to count down from 5 to 1:\n")

x <- 5

while (x >= 1) {

cat(x, " ")

x <- x - 1

}

cat("\n\n")

# repeat loop

cat("Using a repeat loop for an infinite loop (break on condition):\n")

y <- 1
```

```
repeat {
cat(y, " ")
y <- y + 1
if (y > 5) {
  break
  }
  }
```