

# R Programming

## UNIT – I

---

### Introduction

#### Overview of R:

##### ***What is R?***

R is a popular programming language used for statistical computing and graphical presentation. Its most common use is to analyse and visualize data.

R is comprised of two parts: the R language itself (which is what most people mean when they talk about R) and a run-time environment.

R is an interpreted language, which means that users access its functions through a command-line interpreter.

Unlike languages such as Python and Java, R is not a general-purpose programming language. Instead, it is considered a *domain-specific language (DSL)*, meaning its functions and use are designed for a specific area of use or domain.

R is equipped with a large set of functions that enable data visualizations, so users can analyse data, model it as required, and then create graphics.

##### ***When was R created?***

R is a popular programming language, especially in certain fields such as data science, academic research, and statistics.

R was created in the early 1990s by University of Auckland statisticians **Ross Ihaka** and **Robert Gentleman**.

Ihaka and Gentleman, who were statistics professors at a university in New Zealand, noticed what Ihaka described as a "common need for improved software" in their computer science labs.

This realization led them to start creating **R**, which is based on the earlier **S** programming language.

Although the professors started working on R in the early 90s, version 1.0.0 was not officially released until February 2000. The latest version of R is **4.4.2**, which was released on 31 October 2024.

##### ***What does R's name mean?***

The **R** language got its name for *two* reasons: because R is the **first letter** in the inventors' names, and because **R** is a play on the name of its parent language **S**, which was originally developed by Bell Telephone Laboratories.

##### ***Is R a low or high-level language?***

R is considered a high-level programming language. High-level languages like R are designed to be easily understood and written by humans, making them more accessible for statisticians, data analysts, and researchers.

## **Why Use R?**

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering, and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has many packages (libraries of functions) that can be used to solve different problems.

## **Who Uses R?**

Plenty of professionals use R in their work, and the language is used across a variety of fields. Some of them are

- Data Science
- Statistics
- Finance
- Social Media
- Academia

## **Careers with R:**

- Data scientist
- Statistical engineer
- Data analyst
- R programmer
- Database administrator
- Researcher
- Financial analyst
- Machine learning scientist
- And more...

## **Which companies use R?**

Here are just a few of the big-name companies that use R in their tech stack:

- Bank of America
- Amazon
- Facebook
- Google
- Accenture
- LinkedIn
- IBM
- Uber
- Deloitte
- The New York Times
- Ford
- HP
- And many more...

## Should I Choose R or Python?

If you are looking to seriously pursue data science and analytics, you will eventually need to learn both languages. **Python**, a **general-purpose** language with a broad ecosystem, is sometimes favoured for its versatility, making it ideal for machine learning, AI, and integration with other tools. **R**, on the other hand, was specifically designed for **statistical computing** and **graphics**, making it a preferred choice for specialized statistical analyses and research.

---

### Installing R:

R programming is a very popular language and to work on that we have to install **RGui**.

Installing R to the local computer is very easy. First, we must know which operating system we are using so that we can download it accordingly.

The official site <https://cloud.r-project.org> provides installer files for major operating systems including Windows, Linux, and Mac OS.

### Install R in Windows:

To install R on your Windows, just follow these steps:

1. Download the R Installer File
2. Run the Installer
3. Install R
4. Add R to the PATH Environment Variable Manually
5. Verify the Installation

### **Step 1: Download the R Installer File**

Go to the official R Project website (<https://cloud.r-project.org>) and download the latest version of R for Windows.

## The Comprehensive R Archive Network

### Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#) ([Debian](#), [Fedora/Redhat](#), [Ubuntu](#))
- [Download R for macOS](#)
- [Download R for Windows](#) ←

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

To proceed, please click the link labelled “**Download R for Windows**”.

## R for Windows

Subdirectories:

|                             |                                                                                                                     |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------|
| <a href="#">base</a>        | Binaries for base distribution. This is what you want to <b>install R for the first time.</b> ←                     |
| <a href="#">contrib</a>     | Binaries of contributed CRAN packages (for R >= 4.0.x).                                                             |
| <a href="#">old contrib</a> | Binaries of contributed CRAN packages for outdated versions of R (for R < 4.0.x).                                   |
| <a href="#">Rtools</a>      | Tools to build R and R packages. This is what you want to build your own packages on Windows, or to build R itself. |

Please do not submit binaries to CRAN. Package developers might want to contact Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the [R FAQ](#) and [R for Windows FAQ](#).

To proceed, please click the link labelled "**install R for the first time.**"

### R-4.4.2 for Windows

↓  
**Download R-4.4.2 for Windows** (83 megabytes, 64 bit)

[README on the Windows binary distribution](#)  
[New features in this version](#)

This build requires UCRT, which is part of Windows since Windows 10 and Windows Server 2016. On older systems, UCRT has to be installed manually from [here](#).

If you want to double-check that the package you have downloaded matches the package distributed by CRAN, you can compare the [md5sum](#) of the .exe to the [fingerprint](#) on the master server.

To proceed, click the link labelled **Download R-4.4.2 for Windows**. Subsequently, the **R Installer File** will begin to download.

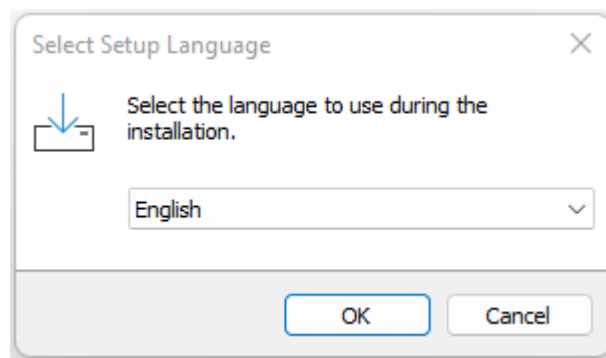
### Step 2: Run the Installer

Open your downloads folder and double-click on the R installer file that you just downloaded.

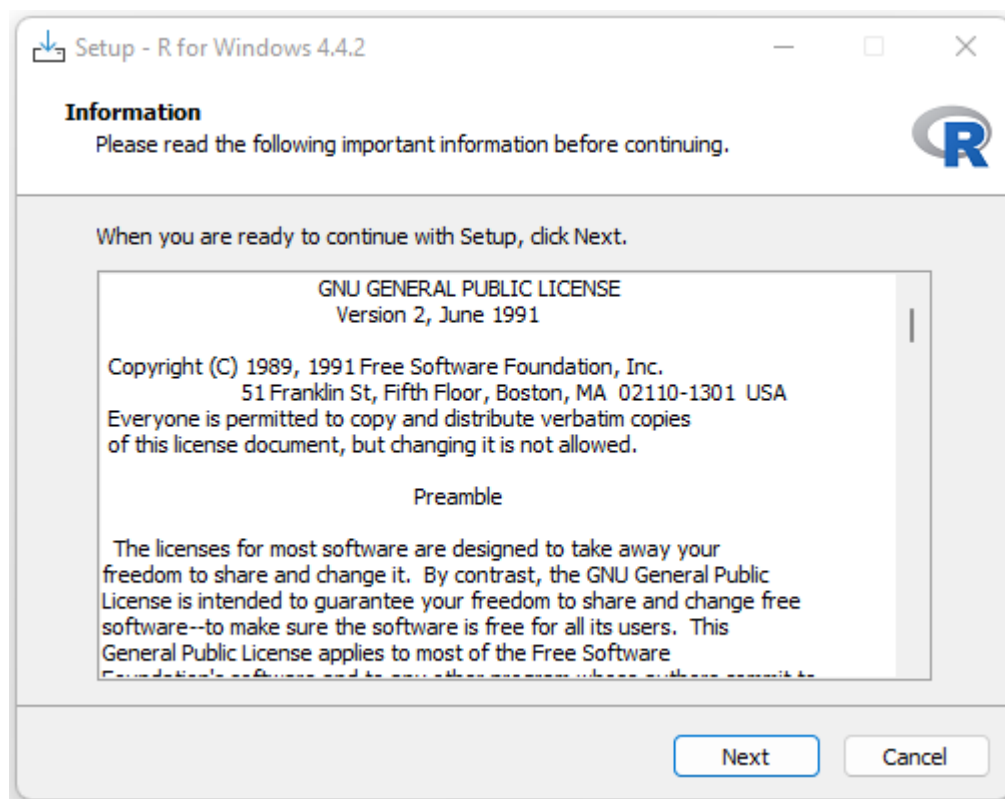
You may be required to grant access based on your security configurations. Please permit it and continue.

### Step 3: Install R

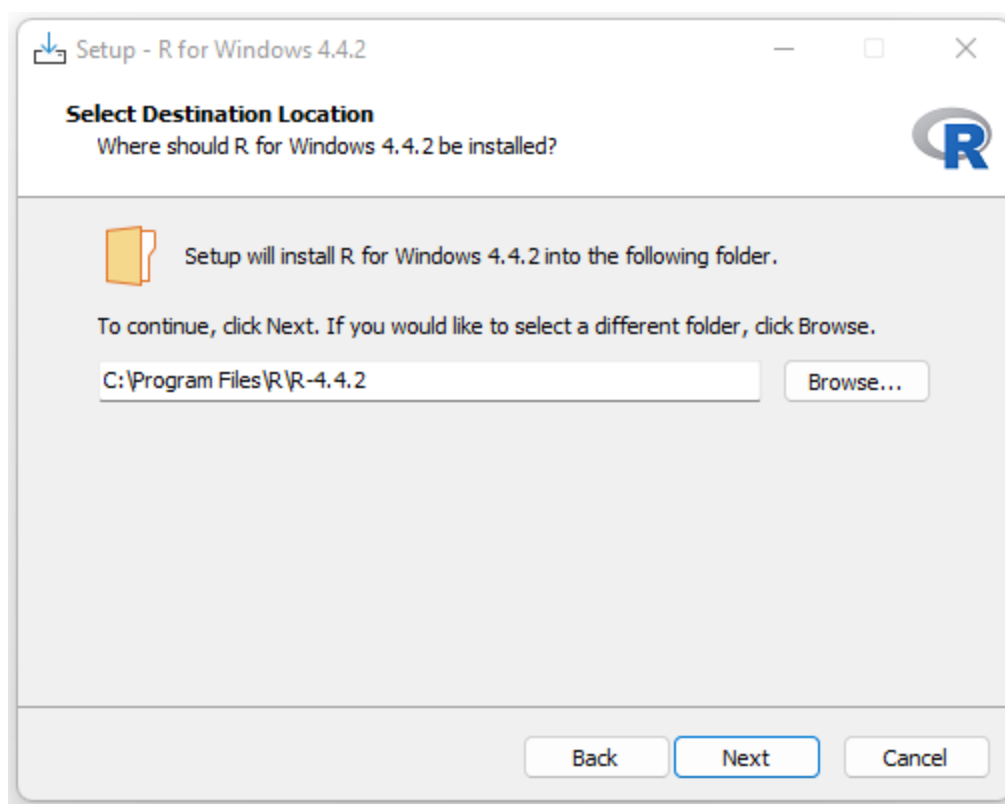
After executing the installer, you will encounter this screen.



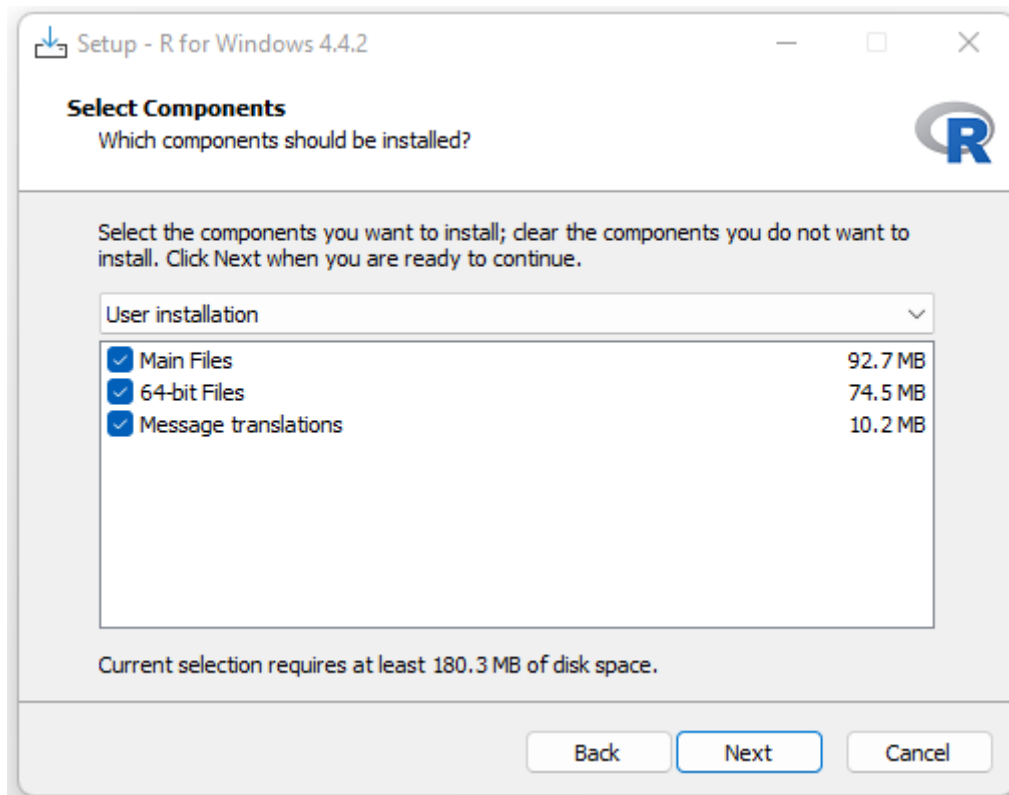
Select your desired language. (Here English) and click on **OK** button.



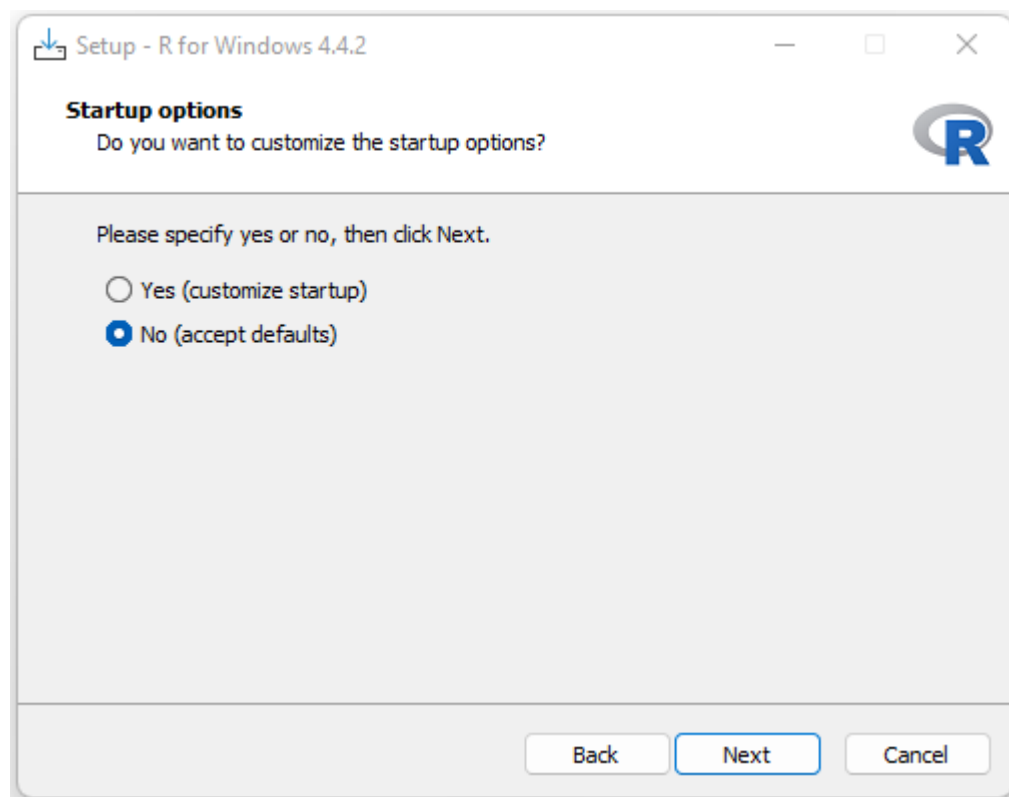
And then click on **Next** button to accept licenses agreements,



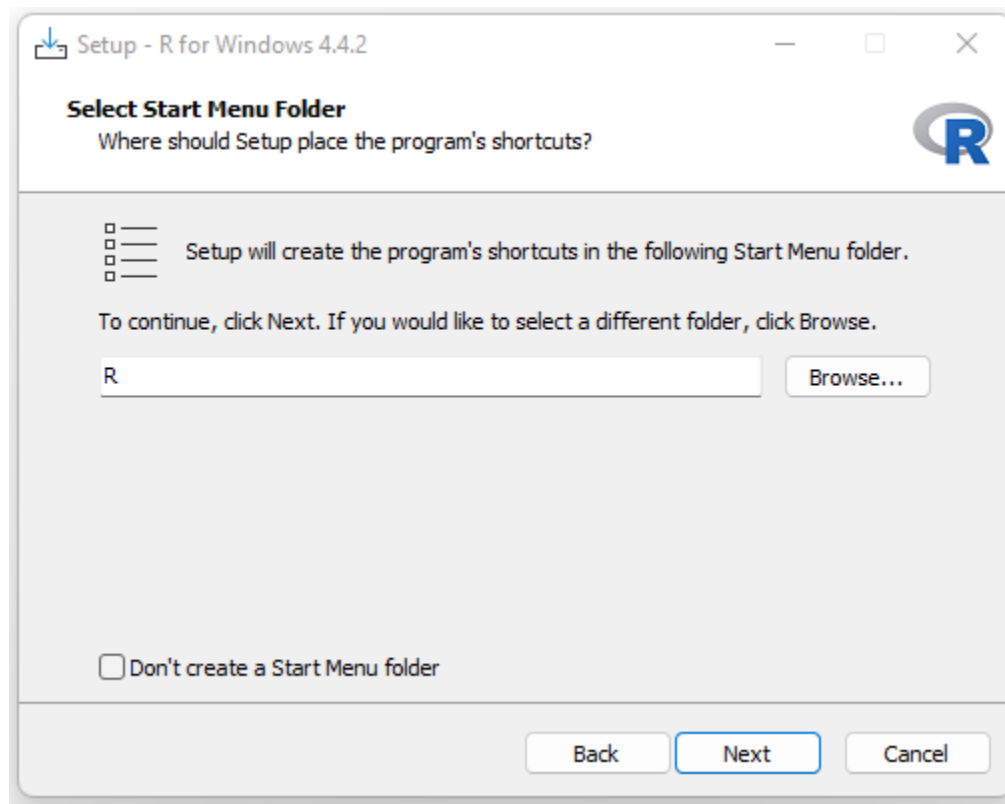
You will be prompted to specify the installation directory. The default location is **C:\Program Files\R\**. You may retain this setting and continue with the installation by clicking on **Next** button.



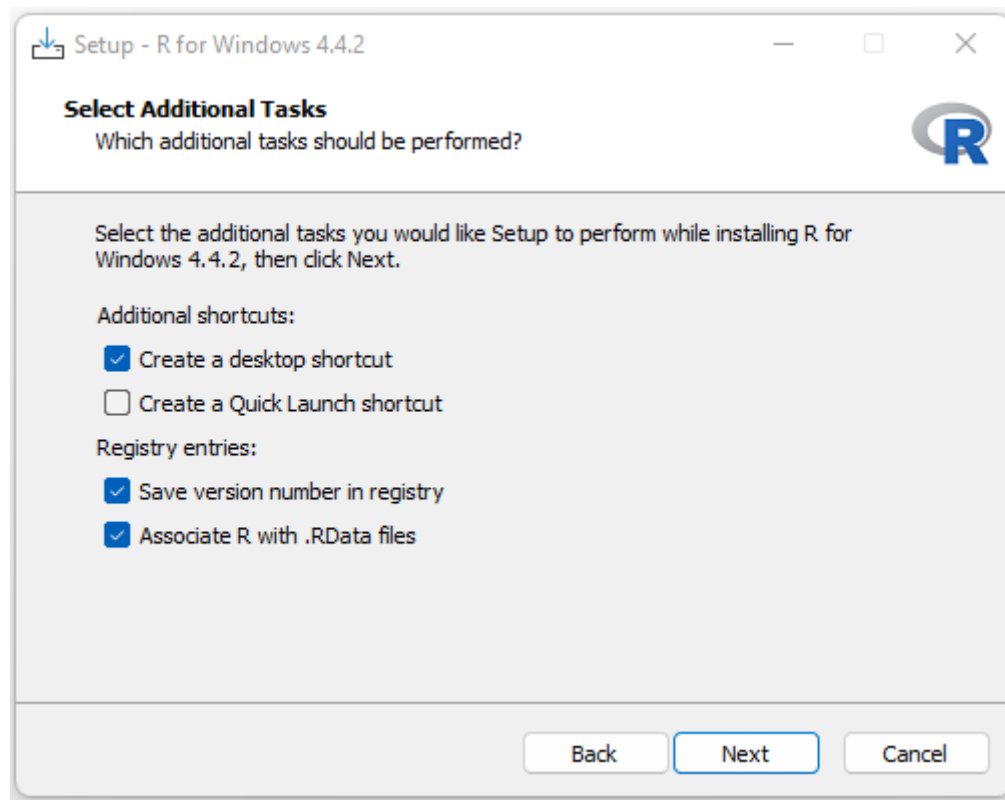
The installer will then prompt you to choose components and where to place shortcuts. Click **Next** button to continue installation.



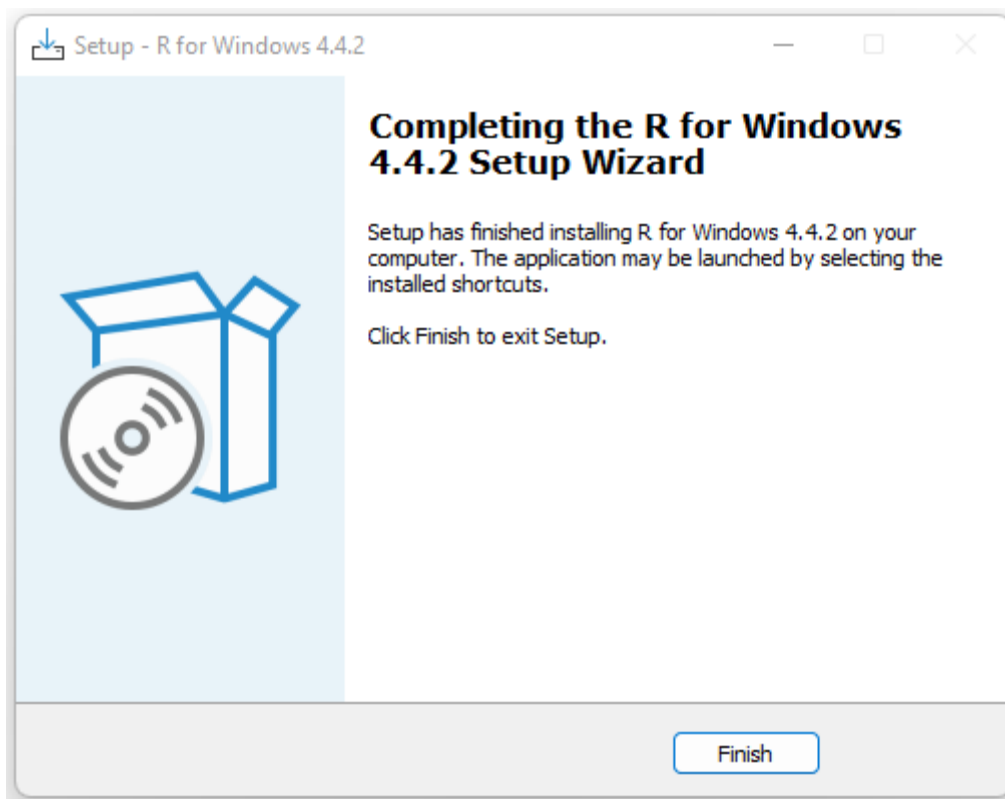
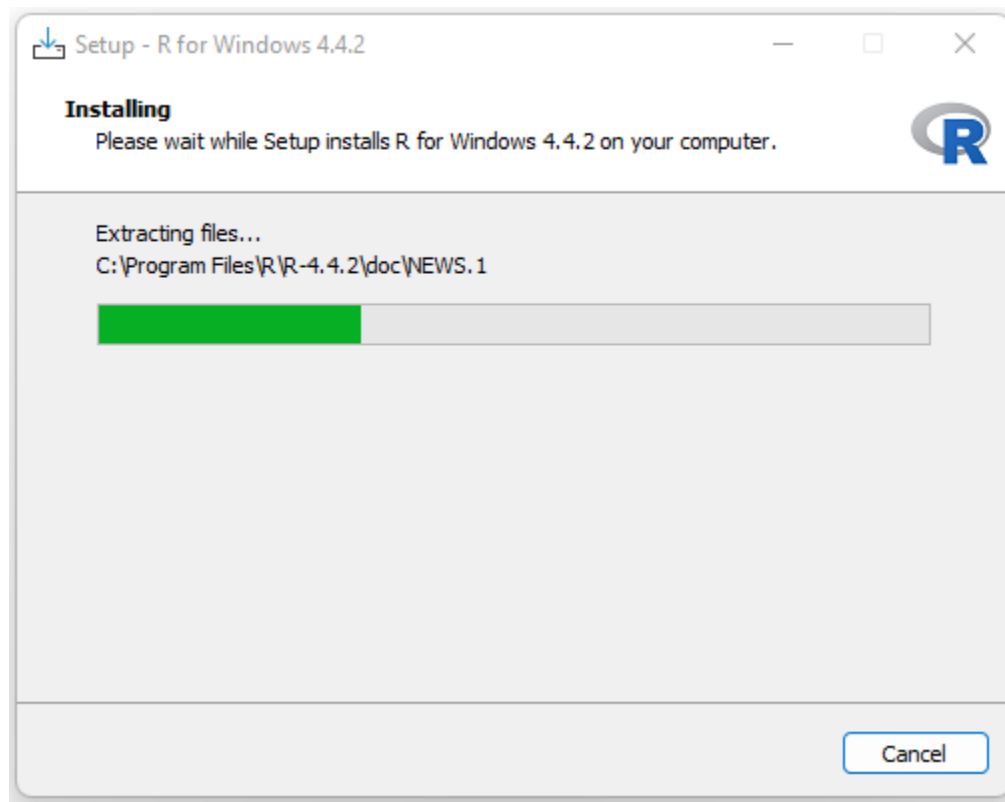
Select **Start-up options**, then click **Next** button.



Select **Start Menu Folder**, then click **Next** button.



Select **Additional Tasks**, then click **Next** button.

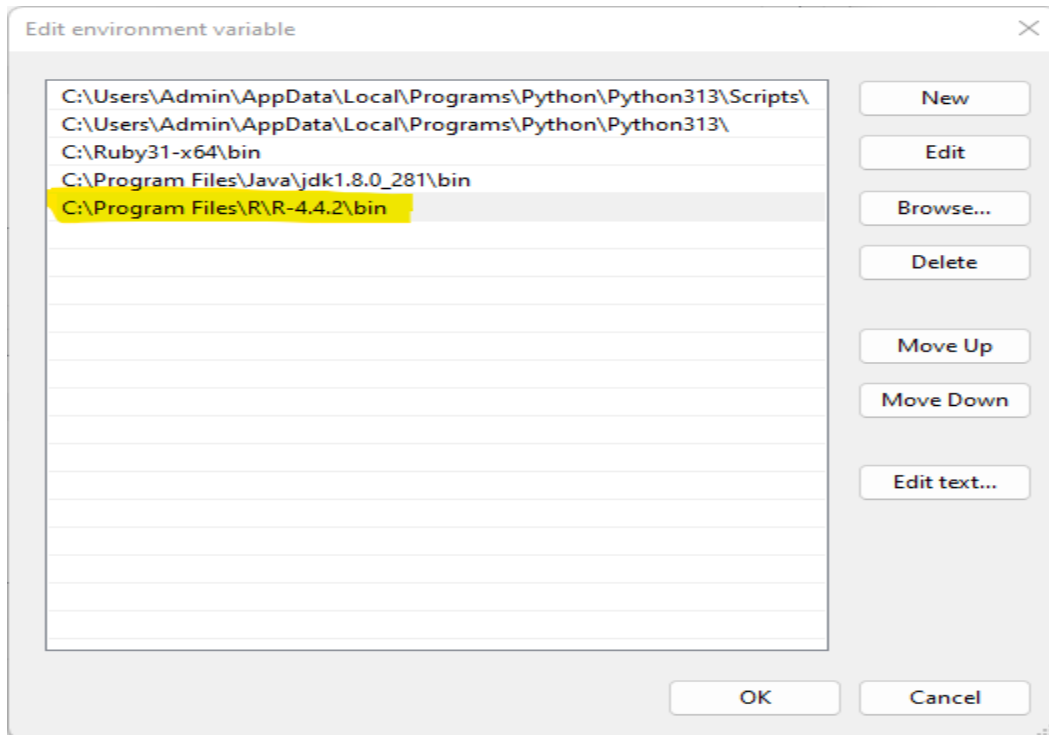


Setup has finished installing R for Windows 4.4.2. Click **Finish** to exit the installer.

#### Step 4: Add R to the PATH Environment Variable Manually

It is essential to manually set that the PATH environment variable of your system is configured properly. To add the PATH environment variable, please follow these steps.

- Copy the path where R is installed to the bin folder. (Usually, it is in C:\Program Files\R\R-4.4.2\bin)
- Right-click on **This PC** and select **Properties**.
- Click on **Advanced System Settings** and then **Environment Variables**.
- Under User and System Variables sections, find and select Path, then click Edit.
- Click New and paste the path.



Click **OK** to close all dialogs and apply the changes.

#### Step 5: Verify the Installation

After the installation is finished, you can confirm that R has been installed correctly by opening a command prompt (cmd) and entering the following command:

**R --version**

```
C:\Users\Admin>R --version
R version 4.4.2 (2024-10-31 ucrt) -- "Pile of Leaves"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under the terms of the
GNU General Public License versions 2 or 3.
For more information about these matters see
https://www.gnu.org/licenses/.
```

## Install R in Linux(Ubuntu):

Installing R on Linux is as simple as following three steps.

### Step 1:

In the initial step, it is essential to update all necessary files within our system by executing the command ***sudo apt-get update***.

### Step 2:

In the second step, we will proceed to install the R programming language on our system by utilizing the command ***sudo apt-get install r-base***.

### Step 3:

In the final step, we need to type **R** and **hit enter** to start using the R editor.

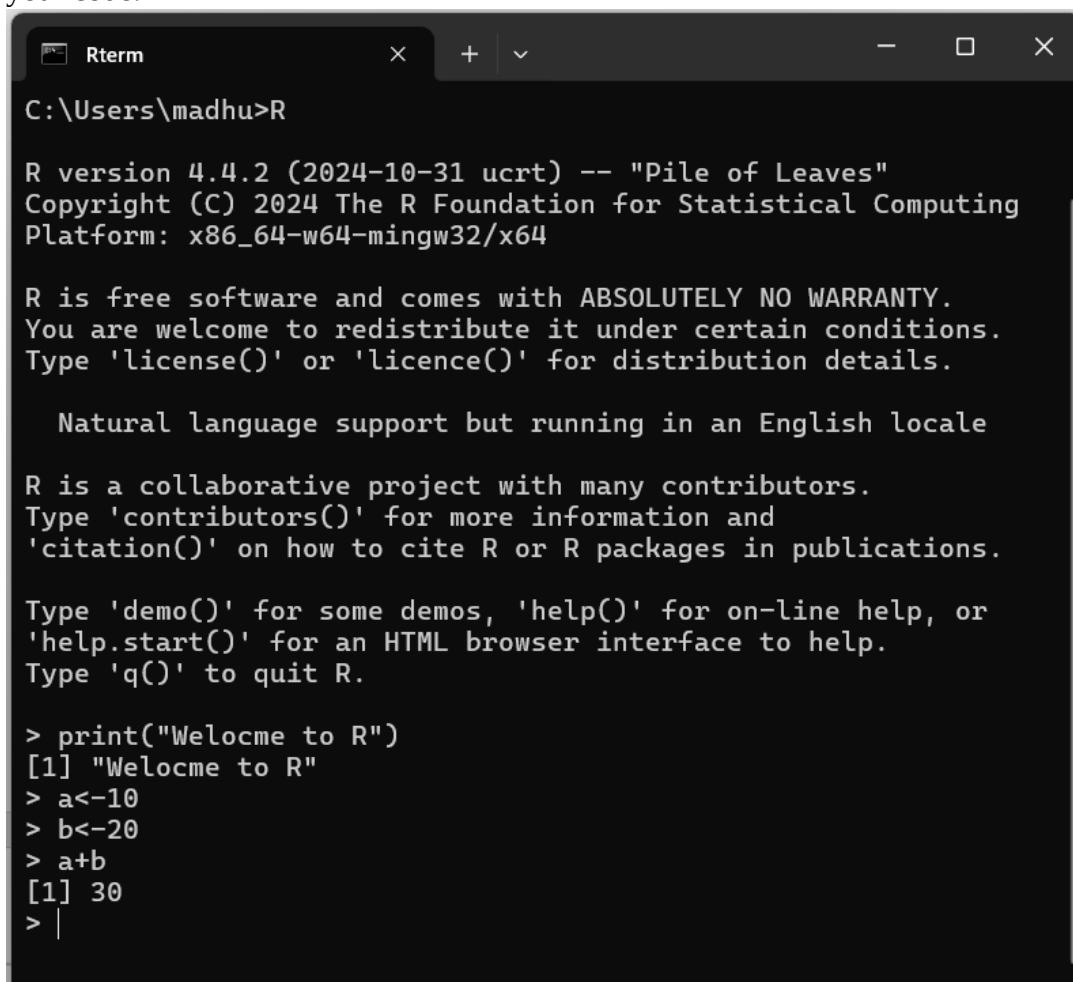
---

## Running R:

After installing R in our system, there are several ways to work with R.

- ***Using command prompt***

To start the R command prompt, just type **R** in the Windows command prompt. After pressing enter, the interpreter will open, and you will see a prompt where you can write your code.



```
C:\Users\madhu>R

R version 4.4.2 (2024-10-31 ucrt) -- "Pile of Leaves"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

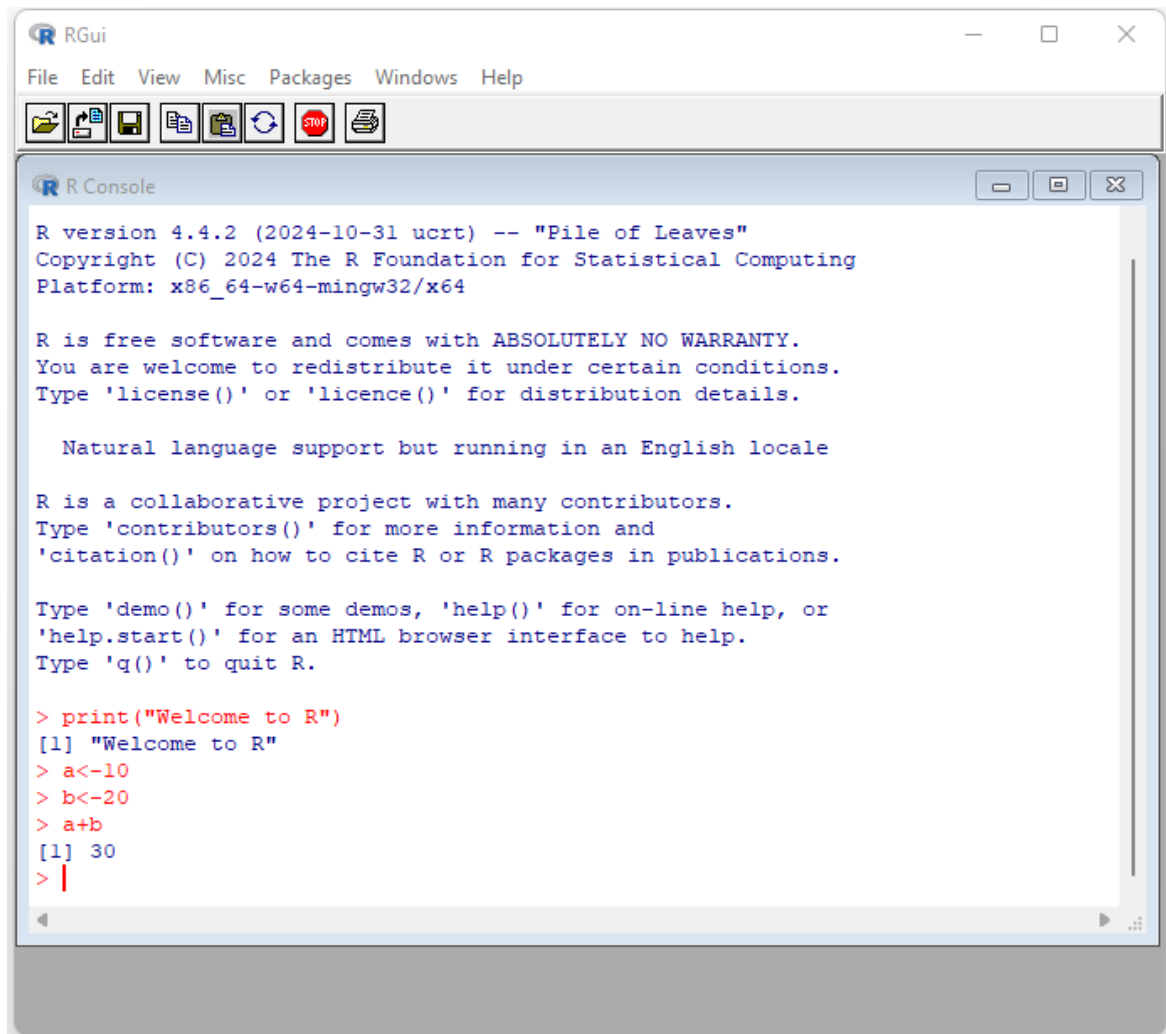
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("Welocme to R")
[1] "Welocme to R"
> a<-10
> b<-20
> a+b
[1] 30
> |
```

- **Using R Gui app**

To start working with R, open the R 4.4.2 app from the Start Menu. Your screen will then appear as shown below.



## **R Script File:**

The R script file is another method to write our programs (set of instructions).

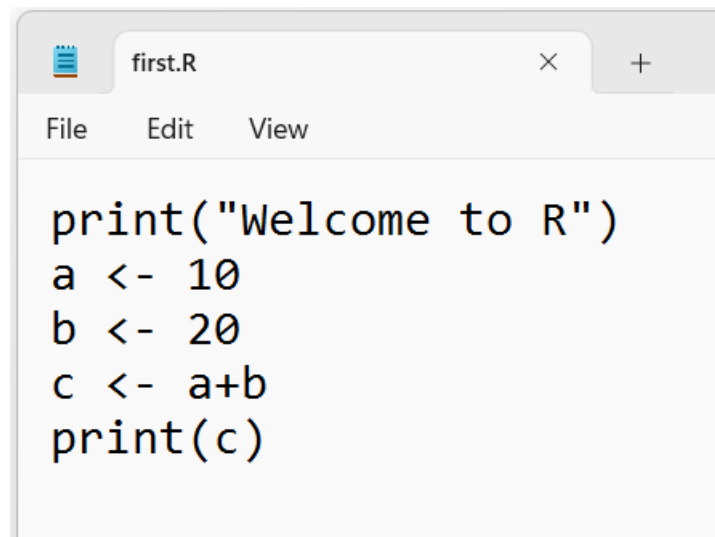
To create an R Script file, just open a text editor and type in your code. After that, save the file with a **.R** extension.

We can run these scripts from the command prompt using the R interpreter called **Rscript**.

Syntax: **Rscript filename. R**

## **Example:**

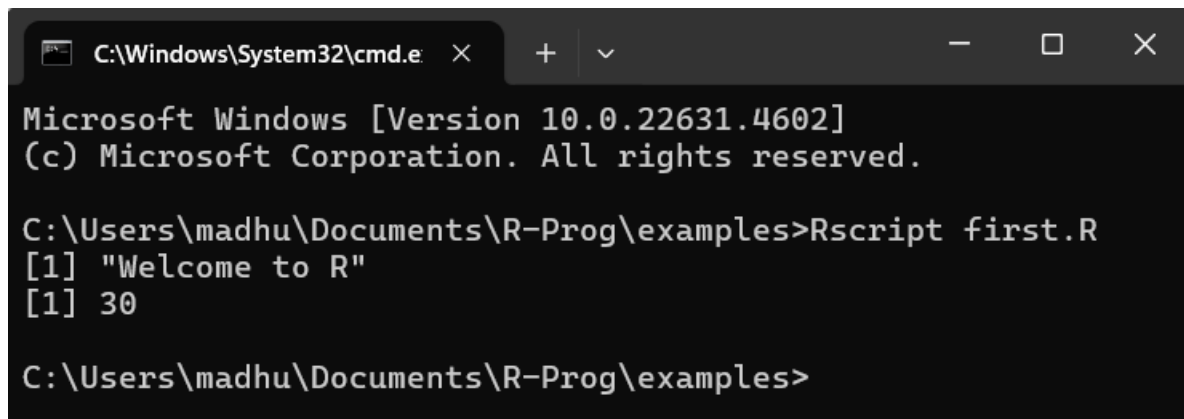
To start, Open Notepad, type the code, and save the file as **first.R**, as shown below.



```
print("Welcome to R")
a <- 10
b <- 20
c <- a+b
print(c)
```

Next, Open the command prompt from your current location and enter the following command to execute the R script file.

***Rscript first.R***



```
C:\Windows\System32\cmd.e
Microsoft Windows [Version 10.0.22631.4602]
(c) Microsoft Corporation. All rights reserved.

C:\Users\madhu\Documents\R-Prog\examples>Rscript first.R
[1] "Welcome to R"
[1] 30

C:\Users\madhu\Documents\R-Prog\examples>
```

---

## **Comments in R:**

In general, Comments are used in a programming language to describe the program or to hide some part of code from the interpreter.

R supports only ***single line comments***.

In case the user wants to specify a single line comment, then comment must start with ‘#’. R will ignore anything that starts with #.

### **Example:**

```
# This is single line comment
print ("Welcome to R")
```

In R, unlike other programming languages, like Java or python, there is ***no syntax*** in R for ***multiline comment***. Instead, you just need to put a # at the beginning of each line to make a multiline comment.

---

## Variables in R:

A variable is a named memory location where we can keep values for a specific program. In simpler terms, a variable is a name that points to a memory location.

A variable is also called an identifier and is used to store a value.

### **Creating Variables in R:**

In R, do not need to declare a variable explicitly. When a value is assigned to a variable, it is automatically declared.

To assign a value to a variable, use the “<-” symbol. To print the variable value, just type the variable name.

#### Syntax:

```
variable_name <- value
```

#### **Example:**

Filename: variables.R

```
# creating variables
sname <- "Naveen"
sage <- 20
# printing variables
sname
sage
```

#### Output:

```
Rscript variables.R
[1] "Naveen"
[1] 20
```

In the above example, *sname* and *sage* are **variables**, while "Naveen" and 20 are **values**.

In R, unlike other programming languages, you don't need to use a function to display or print variables. You can simply write the variable's name.

However, R does have a **print()** function to print / output variables. This might be useful if you are familiar with other programming languages, such as Python.

#### **Example:**

Filename: variables\_p.R

```
# creating variables
sname <- "Naveen"
sage <- 20
# printing variables
print(sname)
print(sage)
```

### Output:

```
Rscript variables_p.R  
[1] "Naveen"  
[1] 20
```

### Reminders:

- In many programming languages, the = sign is typically used for assigning values. But in R, you can use both = and <- for assignments. Most of the time, though, it's better to use <- since there are some cases in R where using = might not be allowed.
- You can choose whether or not to use the **print()** function to display output of your code. But if your code is within an R expression (like inside curly braces {}), you should use the print() function to display the result.

### **Rules for naming variables in R:**

In R, A variable can be given a simple name, such as a or b, or it can have a more detailed name like first\_name, age, or fullname.

There are some rules for naming variables in R, those are

- A variable name must begin with a letter or a period(.) and can be followed by more letters, numbers, a period (.), or an underscore (\_).
- If the variable starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with an underscore (\_) or a number.
- Variable names are case-sensitive (i.e., variables 'a' and 'A' are not equal).
- Reserved words are not allowed for use as variable names. (If, TRUE, NULL...).

### Examples for **valid** variable names:

```
firstname <- "Naveen"  
first_name <- "Naveen"  
firstName <- "Madhu"  
FIRSTNAME <- "Madhu"  
name1 <- "Durga"  
.fname <- "Durga"
```

### Examples for **Invalid** variable names:

```
first name <- "Naveen"  
first-name <- "Naveen"  
first@Name <- "Madhu"  
_FNAME <- "Madhu"  
1name <- "Durga"  
.1name <- "Durga"
```

## Multiple Variables:

R allows us to assign a single value to multiple variables in one line.

### Syntax:

```
var1 <- var2 <- var3 <- value
```

### **Example:**

Filename: variables\_m.R

```
# Assign a one value to multiple variables in single line
a <- b <- c <- 10
# Print variable values
print(a)
print(b)
print(c)
```

### Output:

```
Rscript variables_m.R
[1] 10
[1] 10
[1] 10
```

---

## **Datatypes in R:**

In general, Data types specify what type of data will be stored in variables. In other words, the variables can hold values of different data types.

In R, No need to specify the type of variable because the variable automatically changes its datatype based on assigned value.

R provides the ***class()*** function which enables us to check the data type of the variable.

R has several basic data types, which include:

- numeric
- integer
- complex
- character (a.k.a. string)
- logical (a.k.a. boolean)

### **Numeric Data Type:**

The ***numeric*** data type in R is used to represent all real numbers, whether they have decimal points or not. Like 12, 15.6, 456, -78, -56.3.

### **Example:**

Filename: numeric\_d.R

```
# without decimals
age <- 23
print(age)
```

```
# with decimals
weight <- 48.5
print(weight)
# print data type of variables
print(class(age))
print(class(weight))
```

Output:

```
Rscript numeric_d.R
[1] 23
[1] 48.5
[1] "numeric"
[1] "numeric"
```

### Integer Data Type:

The **integer** data type is used to represent real values without decimal points. And we use the suffix **L** to specify integer type of data. For example, 45L, 123L, 78L, -45L.

#### Example:

Filename: integer\_d.R

```
# without decimals
age <- 23L
print(age)
# print data type of variables
print(class(age))
```

Output:

```
Rscript integer_d.R
[1] 23
[1] "integer"
```

### Complex Data Type:

The complex data type is used to specify purely imaginary values in R. We use the suffix **i** to specify the imaginary part. For example, 3 + 2i, -2+5i.

#### Example:

Filename: complex\_d.R

```
val <- 5 + 6i
print(val)
# print data type of variables
print(class(val))
```

Output:

```
Rscript complex_d.R
[1] 5+6i
[1] "complex"
```

## Character Data Type:

The character data type is used to specify *character* or *string* values in a variable.

In programming, a string is a set of characters. For example, 'A' is a single character and "Apple" is a string.

You can use single quotes ' ' or double quotes " " to represent strings. In general, we use:

- Single quotes ( ' ' ) for character values
- Double quotes ( " " ) for string values

### Example:

Filename: character\_d.R

```
# create a string variable
fname <- "Apple"
print(class(fname))
# create a character variable
ch <- 'A'
print(class(ch))
```

Output:

```
Rscript character_d.R
[1] "character"
[1] "character"
```

## Logical Data Type:

The logical data type in R is also known as **boolean** data type. It can only have two values: TRUE and FALSE.

### Example:

Filename: logical\_d.R

```
b_val1 <- TRUE
print(b_val1)
print(class(b_val1))
b_val2 <- FALSE
print(b_val2)
print(class(b_val2))
```

Output:

```
Rscript logical_d.R
[1] TRUE
[1] "logical"
[1] FALSE
[1] "logical"
```

**Note:** R has some Special Values those are,

- *NA*: Missing data.
  - *NULL*: Represents an empty or undefined value.
  - *NaN*: Not a number (e.g., 0/0).
  - *Inf* and *-Inf*: Positive and negative infinity (e.g., division by zero).
-

## Type conversion in R:

In R, type conversion refers to converting variable from one data type to another. This is necessary when dealing with incompatible types or when preparing data for analysis. R provides various functions for type conversion.

- **as.numeric():** Converts a variable to numeric format.
- **as.integer():** Converts a variable to integer format.
- **as.character():** Converts a variable to character format.
- **as.logical():** Converts a variable to logical format (TRUE or FALSE).
- **as.factor():** Converts a variable to a factor, which is a special type of categorical variable.

### **Example:**

Filename: typeconversion.R

```
# converting numeric to character
x <- 12
x_ch <- as.character(x)
print(x_ch)
print(class(x_ch))
# converting character to numeric
y_ch = "23.1"
y = as.numeric(y_ch)
print(y)
print(class(y))
# converting numeric to logical
z = 1
z_lg = as.logical(z)
print(z_lg)
print(class(z_lg))
# converting logical to numeric
lg = FALSE
n = as.numeric(lg)
print(n)
print(class(n))
```

### Output:

```
Rscript typconversion.R
[1] "12"
[1] "character"
[1] 23.1
[1] "numeric"
[1] TRUE
[1] "logical"
[1] 0
[1] "numeric"
```

---

## Input & Output Functions in R: (or) Reading and Writing data:

R language provides various functions to read input and write output.

### Input functions:

In R language input from the user is taken from the console using two functions. They are

- ***readline()* function:** Takes input from user in string format.
- ***scan()* function:** Reads data from the user console or file.

### **readline() function:**

A *readline()* function takes or reads the input from a user console in string format.

#### Syntax:

```
var_name <- readline()
```

```
var_name <- readline(prompt = “ “)
```

The *readline()* takes input from the user in character or string form and assigns it to a variable. And the “***prompt***” argument is used to display some text to the user.

### **Example:**

Filename: readlinedemo.R

```
# Program to read input from user using readline()
# readline() reads input from user console
x <- readline()
# print the value
print(x)
y <- readline(prompt = "Type Something : ")
# print the value
print(y)
```

#### Output:

```
Rscript readlinedemo.R
Madhu
[1] "Madhu"
Type Something : Naveen
[1] "Naveen"
```

### **scan() function:**

In R , the *scan()* function is a built-in function that reads data or input from the user console.

By default this function reads numeric values and returns a numeric vector. And it allows us to read input as character or string by using “***what***” argument.

#### Syntax:

```
var_name <- scan()
```

```
var_name <- scan(what = “ “)
```

### Example:

Filename: scandemo.R

```
# reads input as numeric
x <- scan()
print(x)
print(class(x))
# reads input as character
y <- scan(what = " ")
print(y)
print(class(y))
```

Output:

```
Rscript scandemo.R
1: 12
2: 23
3: 12.5
4: 56
5: -45
6:
Read 5 items
[1] 12.0 23.0 12.5 56.0 -45.0
[1] "numeric"
1: "madhu"
2: "durga"
3: "naveen"
4:
Read 3 items
[1] "madhu" "durga" "naveen"
[1] "character"
```

### **Output functions:**

In R, You can print values by entering a variable or object on the console or by using the *print*, *cat* and *sprintf* functions.

#### **print() function:**

We can use the **print()** function to display the output to the terminal. This function is useful when you want the code to print some variable, message, or object.

Example:

```
# Print a message
print("This is a sample print")
```

Output:

```
[1] "This is a sample print"
```

- You can also **print variables** and **objects**.

Example:

```
str <- "R is Language"
# Print a variable
print(str)
```

Output:

```
[1] "R is Language"
```

- The **print** function allows you to customize the output. For instance, you can use the **digits** argument to set the desired number of digits.

Example:

```
x <- 4.56892
print(x, digits= 3)
```

Output:

```
[1] 4.57
```

- By default, if you print character values they will be shown with quotes. However, you can use the **quote=FALSE** argument to remove quotes from the output. This is equivalent to **noquote()** function.

Example:

```
print("hello", quote=FALSE)
```

Output:

```
[1] hello
```

### **cat() function:**

The **cat** function **concatenates and prints** objects without quotes. i.e., This function concatenates all the arguments and forms a single string which it then prints. This function also allows you to output values to a file.

Example:

```
cat("Hello \n")
cat("One", "Two", "Three")
```

Output:

```
Hello
One Two Three
```

- By default, the cat function separates the objects with an **empty space**, but you can customize this specifying a new separator with **sep** argument.

Example:

```
cat("One", "Two", "Three", sep = "-")
```

Output:

```
One-Two-Three
```

- If you want display the objects on a new line you can use "**n**" as separator.

Example:

```
cat("One", "Two", "Three", sep = "\n")
```

Output:

```
One
Two
Three
```

- Other interesting difference about **print** and **cat** is how each function handles a "**n**" inside the string to be printed. While cat adds a new line, the print function prints the result 'as is'.

Example:

```
cat("One\nTwo")
print("One\nTwo")
```

Output:

```
One
Two
[1] "One\nTwo"
```

- The cat() function used to concatenate output string and variable.

Example:

```
age <- 20
cat("age is",age)
```

Output:

```
age is 20
```

### **sprintf() function:**

The **sprintf** function **returns a formatted strings** based on format specifiers.

The following are format specifiers in R. Those are

- %d - Integer
- %s - Character string
- %g - Double precision value
- %f - Float value

Example:

```
name <- "Madhan"
age <- 23
sprintf("My name is %s and My age is %d",name,age)
weight <- 47.5
sprintf("My weight is %g",weight)
```

Output:

```
"My name is Madhan and My age is 23"
"My weight is 47.5"
```

---

## Operators in R:

In programming, an operator is a symbol which represents an action. In another words, an operator is a symbol which is used to perform operation on variables and values.

R supports the following operators:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Miscellaneous operators

### **Arithmetic operators:**

Arithmetic operators are the symbols which are used to perform arithmetical operations. R supports following arithmetic operators:

| Operator | Name                              | Example     |                   |
|----------|-----------------------------------|-------------|-------------------|
| +        | Addition                          | $x + y$     | 10 + 5 o/p: 15    |
| -        | Subtraction                       | $x - y$     | 10 - 5 o/p: 5     |
| *        | Multiplication                    | $x * y$     | 10 * 5 o/p: 50    |
| /        | Division                          | $x / y$     | 10 / 4 o/p: 2.5   |
| ^        | Exponent                          | $x ^ y$     | 2 ^ 3 o/p: 8      |
| %%       | Modulus (Remainder from division) | $x \% \% y$ | 15 \% \% 2 o/p: 1 |
| %/%      | Integer Division / Quotient       | $x \% \% y$ | 15 \% \% 2 o/p: 7 |

### **Example:**

Filename: arith\_op.R

```
# Define two numbers
a <- 15
b <- 2
add_result <- a + b # Addition
cat("Addition (a + b):", add_result, "\n")
sub_result <- a - b # Subtraction
cat("Subtraction (a - b):", sub_result, "\n")
mul_result <- a * b # Multiplication
cat("Multiplication (a * b):", mul_result, "\n")
div_result <- a / b # Division
cat("Division (a / b):", div_result, "\n")
exp_result <- a^b # Exponentiation
cat("Exponentiation (a ^ b):", exp_result, "\n")
mod_result <- a %% b # Modulus
cat("Modulus (a %% b):", mod_result, "\n")
int_div_result <- a %/% b # Integer Division
cat("Integer Division (a %/% b):", int_div_result, "\n")
```

### Output:

```
Rscript arith_op.R
Addition (a + b): 17
Subtraction (a - b): 13
```

Multiplication (a \* b): 30  
Division (a / b): 7.5  
Exponentiation (a ^ b): 225  
Modulus (a %% b): 1  
Integer Division (a %/% b): 7

### Assignment operators:

Assignment operators are used to assign values to variables.

| Operator | Name                      | Example |
|----------|---------------------------|---------|
| <-       | Left assignment operator  | x <- 10 |
| ->       | Right assignment operator | 10 -> x |

### Example:

Filename: assign\_op.R

```
x <- 30  
print(x)
```

```
40 -> y  
print(y)
```

Output:

```
Rscript assign_op.R  
30  
40
```

### Comparison / Relational operators:

Comparison / Relational operators are used to compare two values or variables.

| Operator | Name                     | Example |
|----------|--------------------------|---------|
| ==       | Equal                    | x == y  |
| !=       | Not equal                | x != y  |
| >        | Greater than             | x > y   |
| <        | Less than                | x < y   |
| >=       | Greater than or equal to | x >= y  |
| <=       | Less than or equal to    | x <= y  |

### Example:

Filename: relational\_op.R

```
a <- 10  
b <- 20  
cat("a = ", a, "\n")  
cat("b = ", b, "\n")  
cat("Is a equal to b? :", (a == b), "\n") # Equal to (==)  
cat("Is a not equal to b? :", (a != b), "\n") # Not equal to (!=)  
cat("Is a greater than b? :", (a > b), "\n") # Greater than (>)  
cat("Is a less than b? :", (a < b), "\n") # Less than (<)  
cat("Is a greater than or equal to b? :", (a >= b), "\n") # (>=)  
cat("Is a less than or equal to b? :", (a <= b), "\n") # (<=)
```

### Output:

```
Rscript relational_op.R
a = 10
b = 20
Is a equal to b? : FALSE
Is a not equal to b? : TRUE
Is a greater than b? : FALSE
Is a less than b? : TRUE
Is a greater than or equal to b? : FALSE
Is a less than or equal to b? : TRUE
```

### **Logical operators:**

The logical operators allow a program to decide based on multiple conditions. In simple words, Logical operators are used to combine two or more conditional statements.

| Operator | Name                                                                            | Example |
|----------|---------------------------------------------------------------------------------|---------|
| &        | Element-wise Logical AND operator. Returns TRUE if both elements are TRUE       | x & y   |
| &&       | Logical AND operator - Returns TRUE if both statements are TRUE                 | x && y  |
|          | Elementwise- Logical OR operator. Returns TRUE if one of the statements is TRUE | x   y   |
|          | Logical OR operator. Returns TRUE if one of the statements is TRUE              | x    y  |
| !        | Logical NOT - Returns FALSE if statement is TRUE                                | !x      |

### **Example:**

Filename: logical\_op.R

```
x <- TRUE
y <- FALSE
# Logical AND (&)
cat("Logical AND (x & y):", x & y, "\n")
# Logical OR (|)
cat("Logical OR (x | y):", x | y, "\n")
# Logical NOT (!)
cat("Logical NOT (!x):", !x, "\n")
# Logical AND with short-circuiting (&&)
cat("Short-circuit AND (x && y):", x && y, "\n")
# Logical OR with short-circuiting (||)
cat("Short-circuit OR (x || y):", x || y, "\n")
```

### Output:

```
Rscript logical_op.R
Logical AND (x & y): FALSE
Logical OR (x | y): TRUE
Logical NOT (!x): FALSE
Short-circuit AND (x && y): FALSE
Short-circuit OR (x || y): TRUE
```

## Miscellaneous operators:

Miscellaneous operators are used to manipulate data. In other words, Miscellaneous operators are used for a special and specific purpose.

| Operator | Name                                       | Example                  |
|----------|--------------------------------------------|--------------------------|
| :        | Creates a series of numbers in a sequence  | x <- 1:10                |
| %in%     | Find out if an element belongs to a vector | x %in% y                 |
| %*%      | Matrix Multiplication                      | x <- Matrix1 %*% Matrix2 |

### Example:

Filename: miscellaneous\_op.R

```
x <- 1:10
print(x)
print(3 %in% x)
print(12 %in% x)
```

### Output:

```
Rscript miscellaneous_op.R
1 2 3 4 5 6 7 8 9 10
TRUE
FALSE
```

---

## Objects in R:

An **object** in R is a **data structure that stores data, functions, or expressions** in memory.

In R, everything is considered an object. This includes:

- Variables
- Data Types
- Data Structures
- Functions

## Data Structures in R:

A **data structure** refers to the way of organizing and storing data in memory. R provides various built-in data structures, those are,

- Vector
- List
- Matrix
- Array
- Data Frame
- Factor

## Vector:

A **vector** is a collection of elements that are all the *same type*. In R, To create a list of elements as a vector, we use **c()** function and in this, the elements are separated by a comma(,).

It is a **one-dimensional** data structure and can hold numeric, character, logical, or other atomic types.

### Syntax:

```
vect_name <- c(e1,e2,e3,..)
```

### Example:

```
> vect <- c("Madhu","Naveen","Durga")
> print(vect)
[1] "Madhu" "Naveen" "Durga"
> print(class(vect))
[1] "character"

> vect_num <- c(45,56,78)
> print(vect_num)
[1] 45 56 78
> print(class(vect_num))
[1] "numeric"

> vect_num <- c(45,"Kiran",23)
> print(vect_num)
[1] "45" "Kiran" "23"
> print(class(vect_num))
[1] "character"
```

## List:

A **list** is a collection of elements of *different types*. In R, To create a list, we use **list()** function and in this, the elements are separated by a comma(,).

In other words, A **list** is a flexible data structure that can store **elements of different types**, including numbers, characters, vectors, matrices, other lists, and even functions.

### Syntax:

```
list_name <- list(e1,e2,e3,..)
```

### Example:

```
> ls_num <- list(45,56,78)
> print(ls_num)
[[1]]
[1] 45
[[2]]
[1] 56
[[3]]
[1] 78
> print(class(ls_num))
[1] "list"
```

```

> ls <- list("Madhu","Durga","Naveen")
> print(ls)
[[1]]
[1] "Madhu"
[[2]]
[1] "Durga"
[[3]]
[1] "Naveen"
> print(class(ls))
[1] "list"

> ls_m <- list(521,"Madhu",21)
> print(ls_m)
[[1]]
[1] 521
[[2]]
[1] "Madhu"
[[3]]
[1] 21
> print(class(ls_m))
[1] "list"

```

### Matrix:

A **matrix** in R is a **two-dimensional** data structure that stores elements of the **same type** (numeric, character, or logical) in **rows and columns**.

In R, To create a matrix, we use **matrix()** function along with **nrow** and **ncol** parameters. Where **nrow** represents number of rows and **ncol** represents number of columns.

#### Syntax:

```
matrix_name <- matrix(vector, nrow =n, ncol=n)
```

### Example:

```

> matrix_num <- matrix(c(10,20,30,40,50,60), nrow = 2, ncol = 3)
> print(matrix_num)
      [,1] [,2] [,3]
[1,]   10   30   50
[2,]   20   40   60

> matrix_ch <- matrix(c('A','B','C','D','E','F'), nrow = 2, ncol = 3)
> print(matrix_ch)
      [,1] [,2] [,3]
[1,] "A"  "C"  "E"
[2,] "B"  "D"  "F"

```

## Array:

In R, An **Array** is a data object which store the data in more than two dimensions.

Arrays are n-dimensional data structures. For example, if we create an array of dimensions (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. They are homogeneous data structures.

In R, an array is created with the help of **array()** function. This function takes a **vector** as an input and uses the value in the **dim** parameter to create an array.

### Syntax:

```
array_name <- array(vector, dim=c(nrow, ncol, ndim))
```

### Example:

```
> arr_name <- array(c(1,2,3,4,5,6,7,8), dim=c(2,2,2))
> print(arr_name)
, , 1
    [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2
    [,1] [,2]
[1,]    5    7
[2,]    6    8
```

## Data Frame:

In R, A **data frame** is a two-dimensional array-like structure, or we can say it is a **table** in which each column contains the value of one variable, and row contains the set of value from each column. Data Frames are two-dimensional, heterogeneous data structures.

Data Frames are the objects which are used to store the tabular data. Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.

In R, To create a data frame we use the **data.frame()** function.

### Syntax:

```
df <- data.frame(vector1, vector2, ..)
```

### Example:

```
> df <- data.frame(Number=c(501,502,503),
                    Name=c("Madhu","Durga","Naveen"),
                    Marks=c(67,77,87))
> print(df)
  Number  Name Marks
1   501 Madhu   67
2   502 Durga   77
3   503 Naveen  87
```

## Factor:

In R, A **Factor** is a data object, which is used to categorize the data.

Examples of factors are:

- Gender: Male , Female
- Music: Rock, Pop, Classic, Jazz
- Result : Pass , Fail
- Temperature: Low, Medium, High

To create a factor in R you need to use the function called ***factor()***. The argument to this factor() is the vector.

Syntax:

```
fact <- factor(vector)
```

## Example:

```
> fac = factor(c("Male", "Female", "Male", "Male", "Female", "Male"))
> print(fac)
[1] Male    Female Male    Male    Female Male
Levels: Female Male
```

## Functions in R:

In R, A Function is defined as a set of statements which are organized together to perform a specific task.

R provides a series of in-built functions, and it allows the user to create their own functions. Functions are used to perform tasks.

In R, the function is created using a keyword called ***function***.

Syntax:

```
func_name <- function(arg_1, arg_2, ...) {
    Function body
}
```

## Example:

Filename: functionDemo.R

```
# function to add 2 numbers
add_nums <- function(x, y) {
    result <- x + y
    print(result)
}
add_nums(25,36) # calling add_nums function
```

Output:

```
Rscript functionDemo.R
[1] 61
```

---

## Sub setting R Objects:

In R Programming Language, subsetting allows the user to access elements from an object.

In this, we will focus on how to extract subsets from R objects.

There are three operators available for this purpose., those are,

- [ ] Operator
- [[ ]] Operator
- \$ Operator

### **[ ] Operator:**

[ ] operator is used to extract element(s) from an object. This allows us to extract multiple elements of an object.

Vectors are basic objects in R and they can be subsetted using the [ ] operator.

‘-‘ is used to access all other indexes of vector.

#### **Example:**

```
> x <- c("a", "b", "c", "d", "e", "f")
> print(x[1]) # Extract the first element
[1] "a"
> print(x[2]) # Extract the second element
[1] "b"
> print(x[1:4]) # Extract the multiple elements
[1] "a" "b" "c" "d"
> print(x[c(1,2,3)])
[1] "a" "b" "c"
> print(x[-c(1,2,3)])
[1] "d" "e" "f"
```

### **[[ ]] Operator:**

[[ ]] operator is used for subsetting of list-objects. This operator is the same as [ ] operator but the only difference is that [[ ]] selects only one element whereas [ ] operator can select more than 1 element in a single command.

#### **Example:**

```
> ls <- list(501, "Madhu", 21)
> print(ls[[1]]) # To access First element
[1] 501
> print(ls[[2]]) # To access Second element
[1] "Madhu"
```

- [[ ]] operator is mainly used to extract nested element of a list.

### Example:

```
> std <- list(502, "Kiran", c(85, 89, 86))
> print(std[[3]]) # To access Third element
[1] 85 89 86
> print(std[[3]][[2]]) #To access second element of third element
[1] 89
```

### \$ Operator:

\$ operator can be used for lists and data frames in R. Unlike [ ] operator, it selects only a single element at a time. It can be used to access an element in named list or a column in data frame.

It is used for extracting **named elements** from a list or data frame. And it returns a single column (vector) when applied to a data frame.

### Example:

```
> std <- list(sno = 502, name = "Kiran", marks = c(85, 89, 86))
> print(std$sno) # To access sno
[1] 502
> print(std$name) # To access name
[1] "Kiran"
> print(std$marks) # To access marks
[1] 85 89 86
> print(std$marks[[2]]) # To access second element of marks
[1] 89
```

### Example:

```
> df <- data.frame(Number=c(501, 502, 503),
                    Name=c("Madhu", "Durga", "Naveen"),
                    Marks=c(67, 77, 87))
> print(df$Name) # To access Names
[1] "Madhu" "Durga" "Naveen"
> print(df$Number) # To access Numbers
[1] 501 502 503
> print(df$Number[2]) # To access Number of second student
[1] 502
> print(df$Name[2]) # To access Name of second student
[1] "Durga"
> print(df$Marks[2]) # To access Marks of second student
[1] 77
```

### Key Differences:

| Operator | Works on                    | Returns          | Example      |
|----------|-----------------------------|------------------|--------------|
| [ ]      | Vectors, Lists, Data Frames | Subset of object | x[1:2]       |
| [[ ]]    | Lists, Data Frames          | Single element   | my_list[[2]] |
| \$       | Lists, Data Frames          | Named element    | df\$age      |

---

## **Packages in R:**

In R, packages are collections of functions, data, and documentation that extend the capabilities of base R. They are essential for performing specialized tasks, such as data manipulation, visualization, statistical analysis, and machine learning.

They are stored in the **Comprehensive R Archive Network (CRAN)** and can be installed and loaded as needed.

- **Installing Packages:**

Before using a package, you need to install it using the ***install.packages()*** function.

Example:

```
install.packages("ggplot2") # Installs ggplot2 package
```

- Packages are downloaded from CRAN (Comprehensive R Archive Network) by default.
- Installed packages are stored in your R library.
- You need to install a package once per system.

- **Loading Packages**

After installation, you must load a package into the R session using ***library()*** or ***require()***.

Example:

```
library(ggplot2) # Loads ggplot2
```

```
require(ggplot2) # Alternative way to load
```

- **library()** stops execution if the package is not found.
- **require()** returns FALSE instead of stopping the script.
- You must load the package every time you start a new R session.
- If you try to load a package that isn't installed, R will throw an error.

- **Checking Installed Packages**

To see which packages are installed on your system, use:

Example:

```
installed.packages() # Lists all installed packages
```

To check if a specific package is installed, use:

Example:

```
"ggplot2" %in% rownames(installed.packages()) #TRUE or FALSE
```

- **Updating Packages**

To update all installed packages:

Example:

```
update.packages() # Update all packages
```

To update a specific package:

Example:

```
install.packages("ggplot2") # Reinstalls the latest version
```

- **Loading Data from Packages**

Many packages come with built-in datasets. To see the datasets available in a loaded package, use:

Example:

```
data() # Lists all available datasets
```

```
data("iris") # Loads the iris dataset
```

- **Removing Packages**

To remove an installed package:

Example:

```
remove.packages("ggplot2") # Uninstalls ggplot2
```

- **Popular R Packages**

Here are some commonly used R packages:

- Data Manipulation: *dplyr, tidyr, data.table*
- Visualization: *ggplot2, plotly, lattice*
- Statistical Analysis: *stats, car, lme4*
- Machine Learning: *caret, randomForest, xgboost*
- Time Series: *forecast, tseries, prophet*
- Text Mining: *tm, text2vec, tidytext*

- **Finding Help for a Package**

To check a package's documentation:

Example:

```
help(package = "ggplot2") # Opens the help page
```

---

## **Rounding in R:**

R provides several functions for rounding numbers, including *round()*, *ceiling()*, *floor()* and *trunc()*. Here is how they work:

**1. round()** – This function rounds the number to a specified number of decimal places.

Example:

```
> round(3.14159, 2) # Output: 3.14
> round(3.5)        # Output: 4 (rounds to the nearest integer)
> round(2.5)        # Output: 2 (R uses "round to even" method)
```

**Note:** R follows the "round to even" rule, meaning 2.5 rounds to 2 and 3.5 rounds to 4 to avoid bias in large datasets.

**2. ceiling()** - This function rounds the number to the nearest integer which is greater than given number.

Example:

```
> ceiling(3.1) # Output: 4
> ceiling(-2.9) # Output: -2
```

**3. floor()** - This function rounds the number to the nearest integer which is less than given number.

Example:

```
> floor(3.9) # Output: 3
> floor(-2.1) # Output: -3
```

**4. trunc()** - This function truncates the given number towards zero.

Example:

```
> trunc(3.9) # Output: 3
> trunc(-3.9) # Output: -3
```

**Note:** This function is similar to floor() but behaves differently for negative numbers (closer to zero).

## **Rounding Complex Numbers:**

When rounding complex numbers in R, the rounding functions apply only to the real and imaginary parts separately:

Example:

```
> z <- 3.14159 + 2.71828i
> round(z, 2) # Output: 3.14+2.72i
> ceiling(z)  # Output: 4+3i
> floor(z)    # Output: 3+2i
```

---

## Complex numbers in R:

In R, complex numbers are natively supported. You can create and perform operations on complex numbers using built-in functions.

- **Creating Complex Numbers**

You can define a complex number using ***complex()*** function.

Example:

```
> z <- complex(real = 3, imaginary = 4)
> print(z) # Output: 3+4i
```

(or)

```
> z <- 3 + 4i
> print(z) # Output: 3+4i
```

- **Basic Operations on Complex Numbers:**

You can perform various basic operations on complex numbers, including Addition, Subtraction, Multiplication and Division.

Example:

```
> z1 <- 3 + 4i
> z2 <- 1 - 2i
> add_res <- z1 + z2          # Addition
> print(add_res)             # 4+2i
> sub_res <- z1 - z2          # Subtraction
> print(sub_res)             # 2+6i
> mul_res <- z1 * z2          # Multiplication
> print(mul_res)             # 11-2i
> div_res <- z1 / z2          # Division
> print(div_res)             # -1+2i
```

- **Complex Number Functions**

R provides several built-in functions for complex numbers:

- **Re(z)** - It returns the real part of the complex number
- **Im(z)** - It returns the imaginary part of the complex number
- **Mod(z)** - It returns the modulus (absolute value)
- **Conj(z)** - It returns complex conjugate

Example:

```
> z <- 3 + 4i
> Re(z)      # 3
> Im(z)      # 4
> Mod(z)     # 5 (sqrt(3^2 + 4^2))
> Conj(z)    # 3 - 4i
```

---