Unit-3:
Introduction to R and working with Data

3.1 Overview of R and its applications in data analysis and statistics
3.2 Installing R and RStudio,
3.3 Basic R syntax, variables, and data types
3.4 Importing data into R from different file formats (CSV, Excel, etc.).
3.5 read, write and view data using data frames

• What Is R?

According to R-Project.org, "**R is a language and environment for statistical computing and graphics.**" It's an open-source programming language often used as a data analysis and statistical software tool.
R was developed in 1993 by Ross Ihaka and Robert Gentleman and includes linear regression, machine learning algorithms, statistical inference, time series, and more
R is a universal programming language compatible with the Windows, UNIX, and Linux platforms.
The environment features of R program is discussed below:

- A high-performance data storage and handling facility
- A vast, easily understandable, integrated assortment of intermediate tools dedicated to data analysis
- Graphical facilities for data analysis and display that work either for on-screen or hardcopy
- The well-developed, simple and effective programming language, featuring user-defined recursive functions, loops, conditionals, and input and output facilities.

The syntax of R consists of three items:

- Variables, which store data
- Comments, which are used to improve code readability
- Keywords, reserved words that have a special meaning for the compiler

Advantages of R programming

Here is a list of some of its major strong points:

- It’s open-source. No fees or licenses are needed, so it’s a low-risk venture if you’re developing a new program.
- It’s platform-independent. R runs on all operating systems, so developers only need to create one program that can work on competing systems. This independence is yet another reason why R is cost-effective!
- It’s great for statistics. Statistics are a big thing today, and R shines in this regard. As a result, programmers prefer it over other languages for statistical tool development.
- It’s well suited for Machine Learning. R is ideal for machine learning operations such as regression and classification. It even offers many features and packages for artificial neural network development.

Its applications in data analysis and statistics
R is widely recognized as one of the premier programming languages and environments for statistical computing and data analysis. Its applications span a wide range of domains within data analysis and statistics, including:

1. Exploratory Data Analysis (EDA):

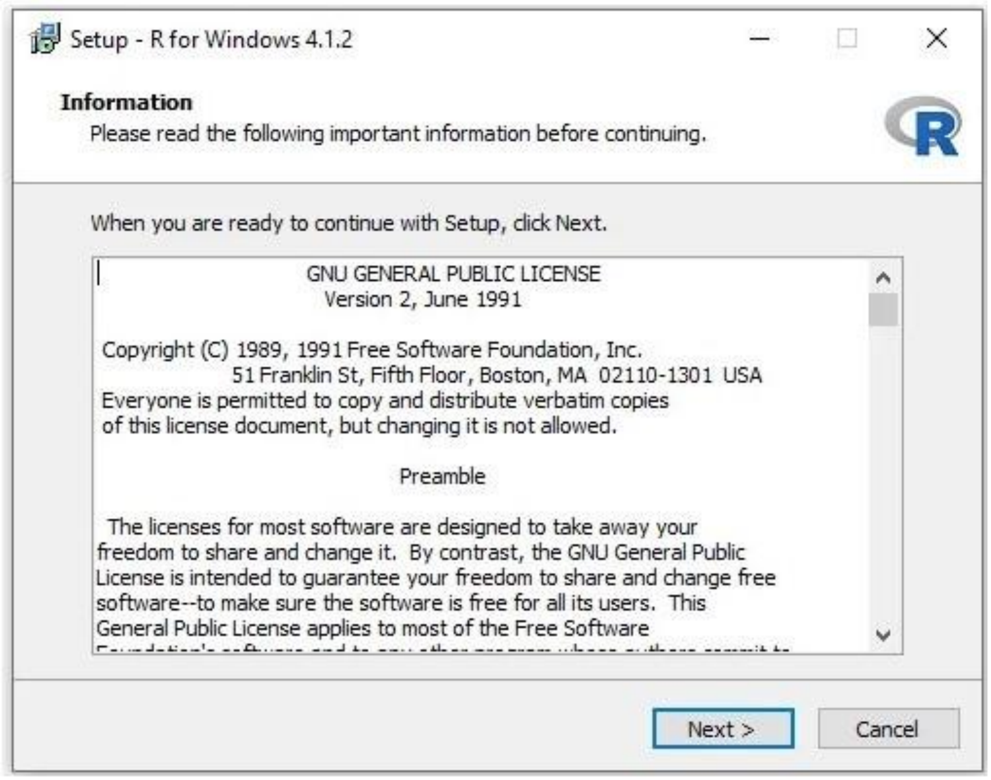
R provides powerful tools for summarizing data, visualizing distributions, detecting outliers, and exploring relationships between variables.
Packages like `ggplot2`, `plotly`, and `ggvis` are popular for creating detailed and customizable visualizations.

2. Statistical Modeling:
R supports a vast array of statistical models, ranging from simple linear regression to complex multilevel models and survival analysis.
Packages like `stats`, `lme4`, `survival`, and `brms` are used for fitting various types of models to data.

- ## <u>**Installing R and RStudio,**</u>

To install R on Windows OS:

1. Go to the CRAN website.
2. Click on **"Download R for Windows"**.
3. Click on **"install R for the first time"** link to download the R executable (.exe) file.
4. Run the R executable file to start installation, and allow the app to make changes to your device.
5. Select the installation language.
6. Follow the installation instructions.


Setup - R for Windows 4.1.2

Information

Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

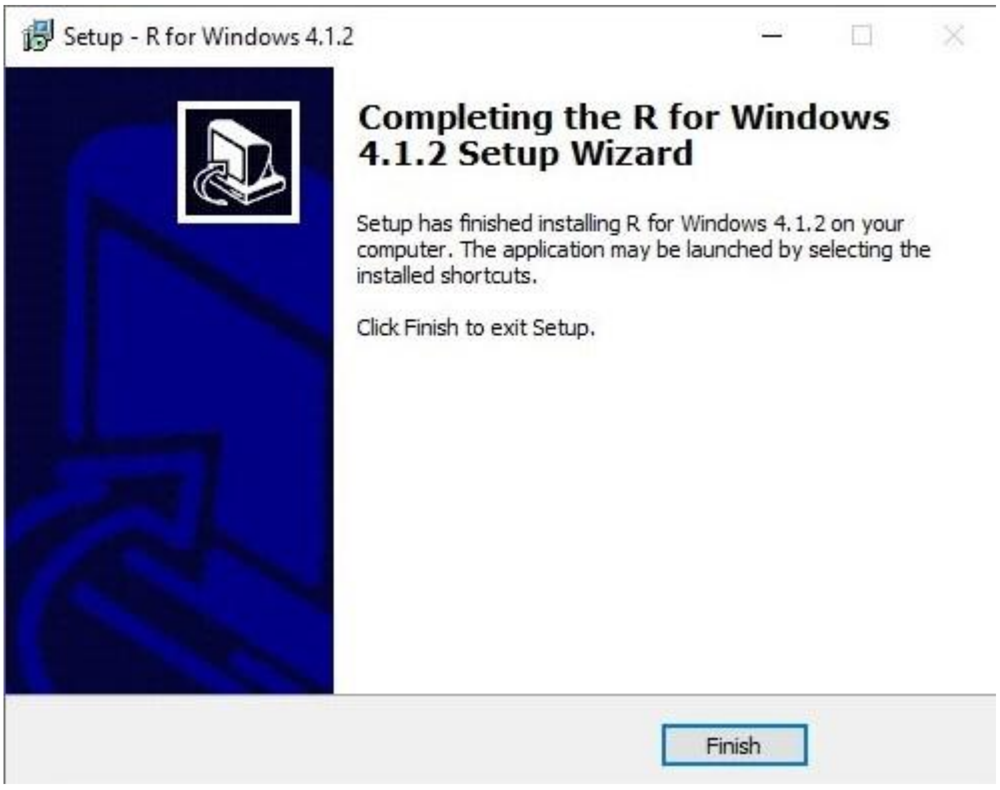
GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users. This
General Public License applies to most of the Free Software

7. Click on **"Finish"** to exit the installation setup.

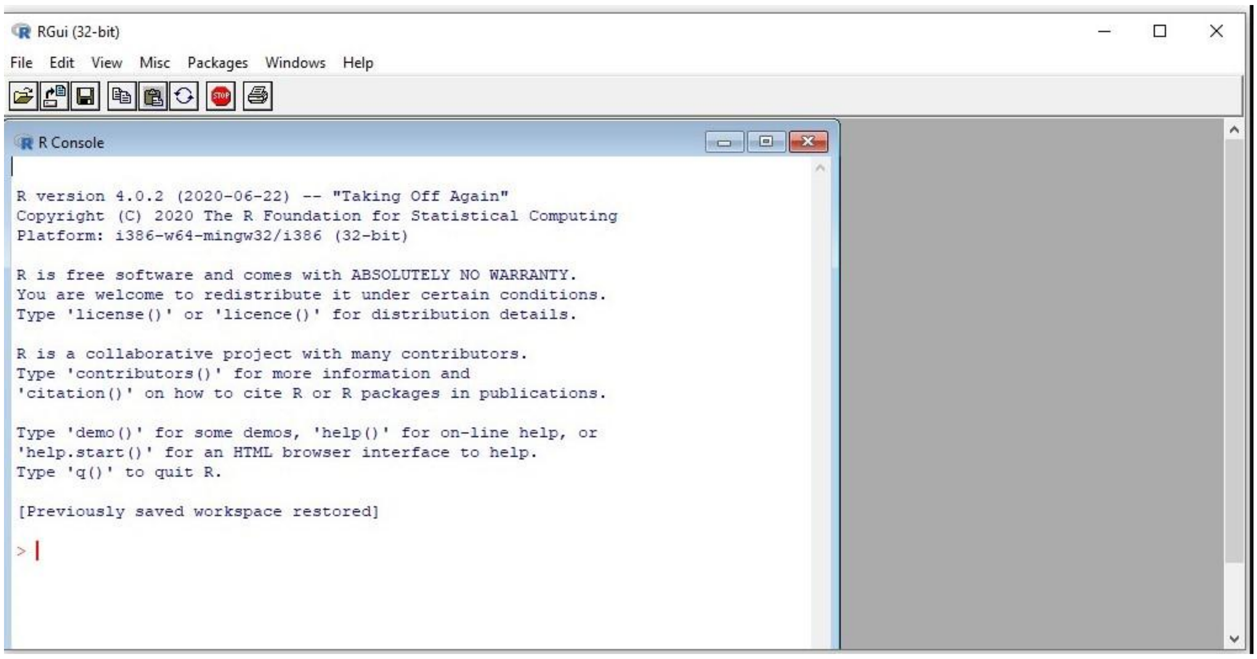

Setup - R for Windows 4.1.2

Completing the R for Windows 4.1.2 Setup Wizard

Setup has finished installing R for Windows 4.1.2 on your computer. The application may be launched by selecting the installed shortcuts.

Click Finish to exit Setup.

R has now been sucessfully installed on your Windows OS. Open the R GUI to start writing R codes.

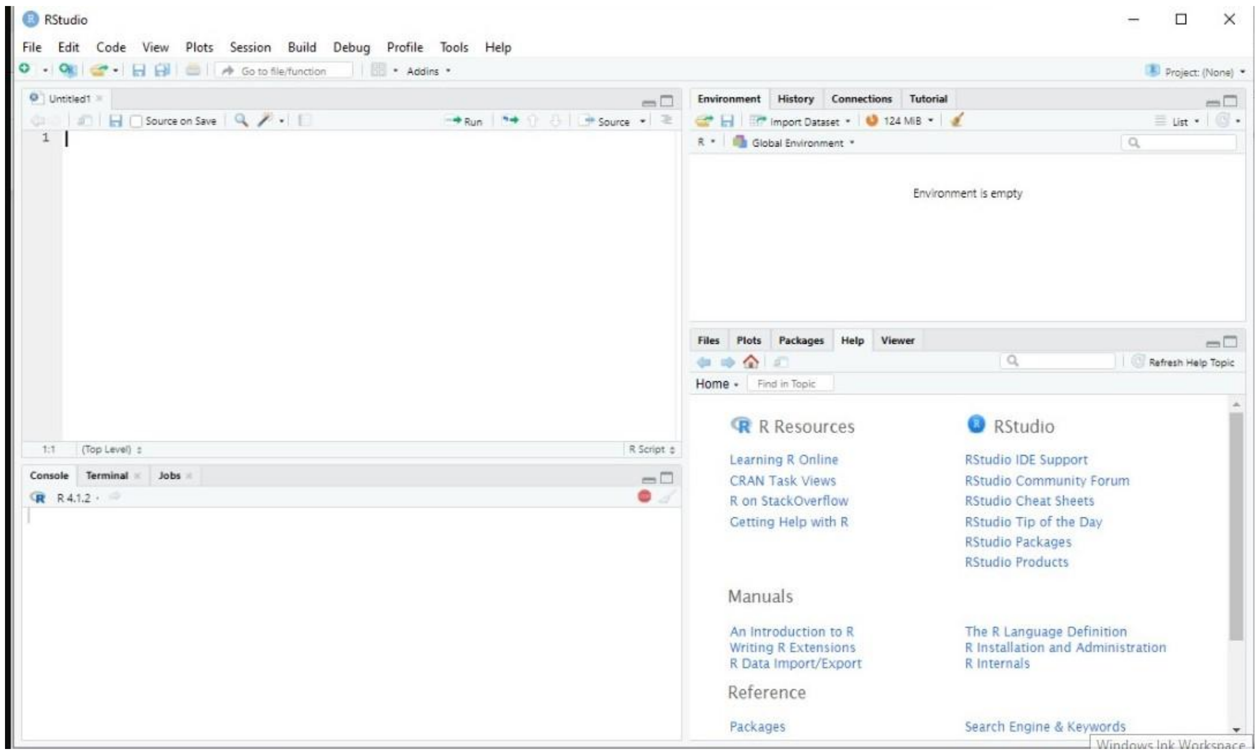


```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

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.

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.

[Previously saved workspace restored]

>
```

• Installing RStudio Desktop

To install RStudio Desktop on your computer, do the following:

1. Go to the RStudio website.
2. Click on **"DOWNLOAD"** in the top-right corner.
3. Click on **"DOWNLOAD"** under the **"RStudio Open Source License"**.
4. Download RStudio Desktop recommended for your computer.
5. Run the RStudio Executable file (.exe) for Windows OS.
6. Follow the installation instructions to complete RStudio Desktop installation.
7. RStudio is now successfully installed on your computer. The RStudio Desktop IDE interface is shown in the figure below:



- ## <u>Data Types</u>

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

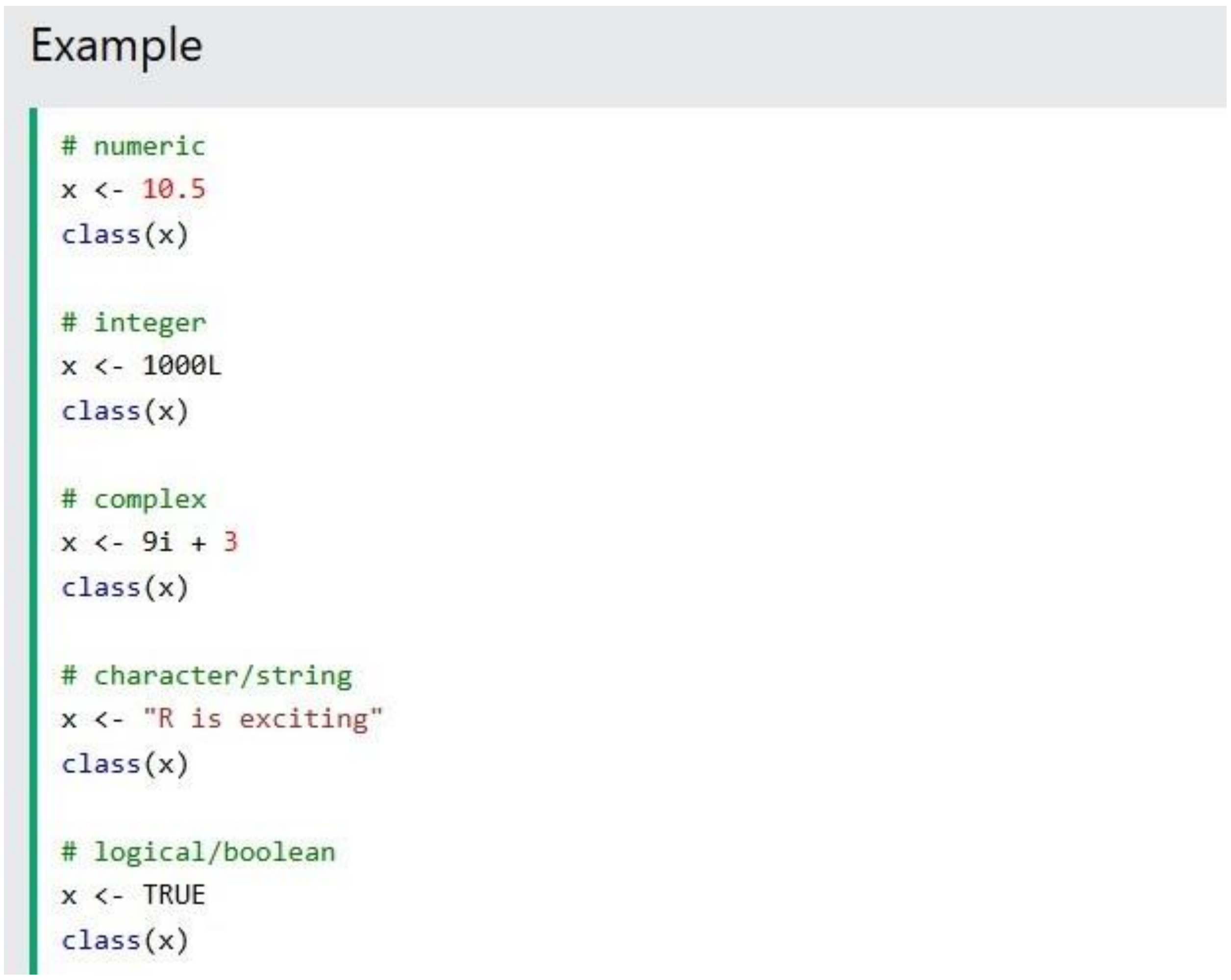
In R, variables do not need to be declared with any particular type, and can even change type after they have been set

- ## <u>Basic Data Types</u>

Basic data types in R can be divided into the following types:

- numeric - (10.5, 55, 787)
- integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)
- complex - (9 + 3i, where "i" is the imaginary part)
- character (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")
- logical (a.k.a. boolean) - (TRUE or FALSE)

We can use the class() function to check the data type of a variable:

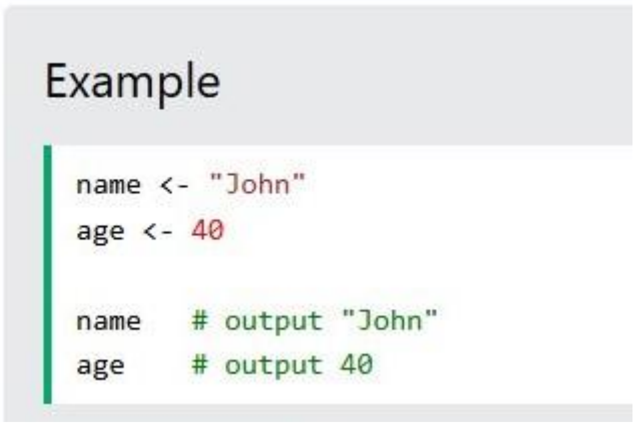

Example

```r
# numeric
x <- 10.5
class(x)

# integer
x <- 1000L
class(x)

# complex
x <- 9i + 3
class(x)

# character/string
x <- "R is exciting"
class(x)

# logical/boolean
x <- TRUE
class(x)
```

- **Creating Variables in R**

Variables are containers for storing data values.

R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the <-sign. To output (or print) the variable value, just type the variable name:


Example

```r
name <- "John"
age <- 40

name   # output "John"
age    # output 40
```

In R we must use . and _(under score) in variable name other symbole is not allowed in variable name like *,-,&,%,# etc.

In starting of variable we may use . and any character only we don't use any digit or any symbole in starting of variable name.

We can also assign variable like:

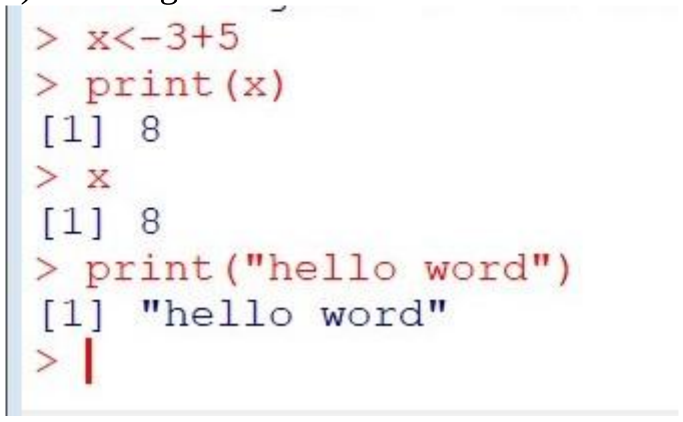
Var1<-10

Var2=10

10->var3

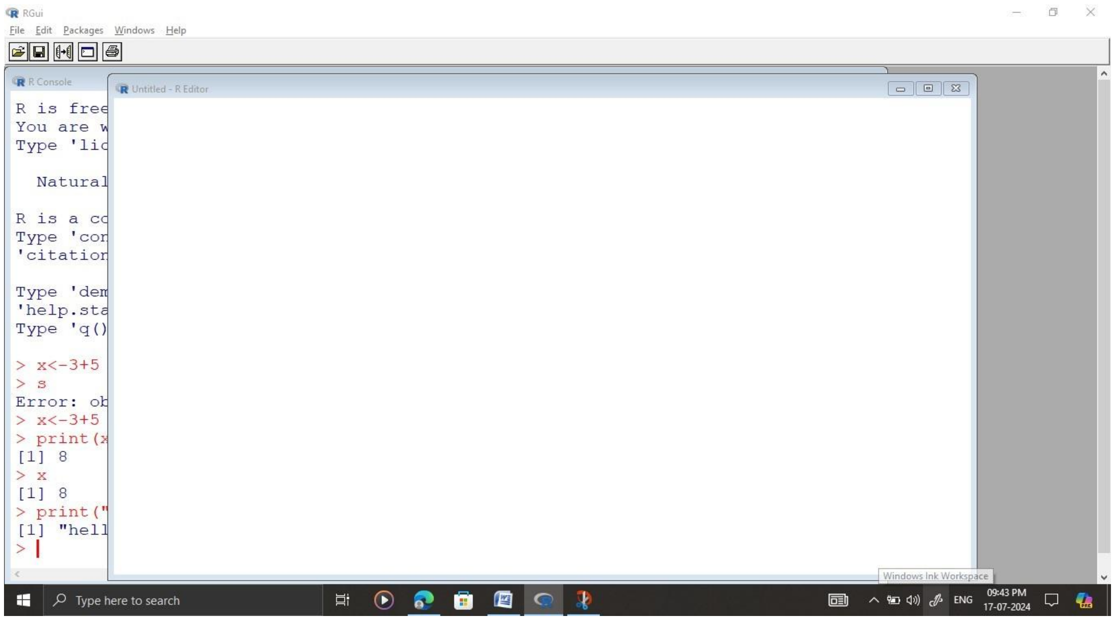
- ## Some important features in R

1) For Increase in size : edit > gui preferences > increase size.
2) For print a data: print("hello word")
3) For assign variable: <-



```
> x<-3+5
> print(x)
[1] 8
> x
[1] 8
> print("hello word")
[1] "hello word"
> 
```

4) for open R editor: File > New script



```
R is free
You are w
Type 'lic

  Natural

R is a co
Type 'con
'citation

Type 'dem
'help.sta
Type 'q()

> x<-3+5
> s
Error: ob
> x<-3+5
> print(x
[1] 8
> x
[1] 8
> print("
[1] "hell
> 
```

In R editor we can write a R code and execute it in R consol by using ctrl+r by line by line.

And if we have to run all the program then we have to use : edit > run all option.

5) R is case sensitive language.

6) We print the variable by the use of print function .

Ex. Print(“hello”)

7) we can use the function cat for print multiple variables.

Ex. cat(x," ",x1)

8) We can define comment by the use of #.

Ex. # Wel come to the first session of R programming.

9) We can quite the R console by the useing of q().

10) If we have to convert other the data type into numeric then, we use the function   as.numeric()

Ex. W<-as.numeric(25L)
W

25

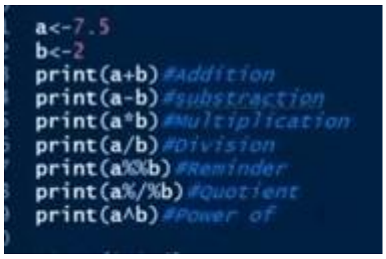
10) If we have to convert other the data type into integer then, we use the function as.integer()

Ex. W<-as.numeric(25.75)
W
25

- **<u>operation in R programming.</u>**

Arithmetic operation (+, -, *, /, %%, %/%, ^)

Ex.



```
a<-7.5
b<-2
print(a+b) #Addition
print(a-b) #substraction
print(a*b) #Multiplication
print(a/b) #Division
print(a%%b) #Reminder
print(a%/%b) #Quotient
print(a^b) #Power of
```

Relational operation

<	Less than
>	Greater than
==	Equal to
<=	Less than equal to
>=	Greater than equal to
!=	Not equal to

Logical operator

&	And
/	Or
!	Not

Assignment operator

<- , = , -> , <<- , ->>

- **<u>Conditional statement in R</u>**

1) if statement

# Example 1: Basic if statement

```
x <- 10

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

2) if else statement

```
# Example 2: if-else statement
x <- 3

if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

3) if else-if else statement

```
# Example 3: if else-if  else statement
x <- 7

if (x > 10) {
  print("x is greater than 10")
} else if (x > 5) {
  print("x is greater than 5 but not greater than 10")
} else {
  print("x is 5 or less")
}
```

4) nested if statement

```
# Example 4: Nested if statements
x <- 12

if (x > 5) {
  if (x < 10) {
    print("x is between 5 and 10")
  } else {
    print("x is greater than or equal to 10")
  }
} else {
  print("x is 5 or less")
}
```

- <u>**looping statement**</u>

1) for Loop

A for loop is used when you know exactly how many times you want to execute a block of code. It iterates over a sequence of values, such as a sequence of numbers or elements in a vector.

Statement

```
for (variable in sequence) {
  # Code block to be executed
    Print("")
}
# Example 1: Looping over a sequence of numbers
for (i in 1:5) {
  print(i)
}
Output
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

2) while Loop

A while loop is used when you want to execute a block of code repeatedly as long as a condition is TRUE.

```
while (condition) {
  # Code block to be executed
}
```

condition: A logical expression that is evaluated before each iteration. If TRUE, the loop continues; if FALSE, the loop terminates.

Example

```
# Example: while looP
count <- 1
```

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

Output

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

3) Control Statements (break and next)

In both for and while loops, you can use break to exit the loop prematurely and next to skip the current iteration and proceed to the next one.

Example:

```
# Example: using break and next
for (i in 1:10) {
  if (i == 3) {
    next  # Skip iteration when i equals 3
  }
  if (i == 8) {
    break  # Exit loop when i equals 8
  }
  print(i)
}
```

Output

```
[1] 1
[1] 2
[1] 4
[1] 5
[1] 6
[1] 7
```

4) Nested Loops

You can nest loops inside one another to perform more complex tasks, such as iterating over multiple dimensions or processing nested data structures.

Example: nested loops

```
for (i in 1:3) {
  for (j in 1:2) {
   print(paste("i:", i, "j:", j))
  }}
```

Output

```
[1] "i: 1 j: 1"
[1] "i: 1 j: 2"
[1] "i: 2 j: 1"
[1] "i: 2 j: 2"
[1] "i: 3 j: 1"
[1] "i: 3 j: 2"
```

R functions

In R, a function is a block of code that performs a specific task and can be reused throughout your script or session. Functions in R are defined using the `function()` keyword. Here's a basic overview of how functions work in R:

Syntax:

```
function_name <- function(arg1, arg2, ...) {
  # Function body: code to execute
  statements
  return(value)  # Optional: specify the return value
}
```

1. Function Name:

This is the name you give to your function, which you use to call it later in your script.

2. Arguments (Parameters):

These are placeholders for values that you pass into the function. They are enclosed in parentheses `()` after the function name.

Arguments are optional. If your function doesn't require any inputs, you can leave them empty.

3. Function Body:

This is where you write the code that you want the function to execute. It can contain any valid R expressions, assignments, control structures (like `if`, `else`, `for`, `while`), and other function calls.

4. Return Value:

The `return()` statement specifies the value that the function should return to the caller. It's optional; if omitted, the function returns the value of the last expression evaluated.

Examples:

Example 1: Simple Function

```
# Define a function that adds two numbers
add_numbers <- function(x, y) {
 result <- x + y
 return(result)
}
# Call the function with arguments

sum_result <- add_numbers(3, 5)
print(sum_result)

# Output: 8
```

Example 2: Function without Arguments

```
# Define a function that prints a greeting
greet <- function() {
 print("Hello, world!")
}
# Call the function
greet()
# Output: Hello, world!
```

Example 3: Function with Default Argument

```
# Define a function with a default argument
power <- function(x, exp = 2) {
  result <- x ^ exp
  return(result)
}

# Call the function with different arguments
```

```
power(3)     # Output: 9 (default exponent 2)
power(2, 3)  # Output: 8 (2^3)
```

Notes:

- Functions in R are first-class objects, meaning they can be assigned to variables, passed as arguments to other functions, and returned as values from other functions.

- Arguments in R functions can have default values. If a default value is provided, the argument becomes optional when calling the function.

Using functions allows you to modularize your code, improve readability, and facilitate code reuse, which are fundamental principles in writing efficient and maintainable R scripts.

- **<u>Take input from user</u>**

In R, the readline() function is used to interactively prompt the user for input within a script or function. It reads a line of text from the console and returns it as a character string. Here's how you can use it:

```
# Example 1: Basic usage
user_input <- readline(prompt = "Enter your name: ")
cat("Hello, ", user_input, "! Nice to meet you.\n")

# Example 2: Using readline in a function
greet_user <- function() {
name <- readline(prompt = "Enter your name: ")
cat("Hello, ", name, "! Welcome.\n")
}
    greet_user()  # Call the function to prompt for input
```

 Basic Usage:

- readline(prompt = "Enter your name: "): This line prompts the user to enter their name with the specified message ("Enter your name: ").
- The user's input is captured in the variable user_input.
- cat("Hello, ", user_input, "! Nice to meet you.\n"): This line prints a greeting message using the value entered by the user.

 Function Example:

- greet_user() is a function that uses readline() to prompt for the user's name.
- Inside the function, name <- readline(prompt = "Enter your name: ") captures the user's input.
- cat("Hello, ", name, "! Welcome.\n") then prints a personalized greeting.

Built in function

R has a wealth of built-in functions that cover a wide range of tasks, from basic arithmetic to complex statistical analyses. Here are some common categories of built-in functions in R:

1. Mathematical Functions

- sqrt(x): Square root of x.
- log(x, base): Natural logarithm of x or logarithm to a specified base.
- exp(x): Exponential function.

2. Statistical Functions

- mean(x): Mean of x.
- median(x): Median of x.
- sd(x): Standard deviation of x.
- var(x): Variance of x.
- summary(x): Summary statistics of x.

3. Vector and Matrix Operations

- length(x): Length of a vector or list.
- dim(x): Dimensions of an object (e.g., matrix).
- t(x): Transpose of a matrix.

4. Character and String Functions

- nchar(x): Number of characters in each element of x.

- tolower(x): Convert to lowercase.
- toupper(x): Convert to uppercase.
- substr(x, start, stop): Extract or replace substrings.

5. Control Flow

- ifelse(test, yes, no): Vectorized conditional statement.
- for(i in 1:n) { }: Looping construct.
- while(condition) { }: Looping construct.

6. Graphics and Plotting

- plot(x, y): Basic scatterplot.
- hist(x): Histogram of x.
- boxplot(x): Boxplot of x.

7. Input and Output

- read.csv(file): Read a CSV file into a data frame.
- write.csv(x, file): Write a data frame to a CSV file.
- scan(): Read data from the console or a file.

8. Date and Time Functions

- Sys.Date(): Current date.
- Sys.time(): Current date and time.
- as.Date(x): Convert an object to a date.

These are just a few examples of the extensive set of built-in functions in R. For a comprehensive list and detailed documentation, you can use R’s help system with ?function_name or help(function_name).

Data structure in R

Data structure is a way to store a data in a memory.

(vector , matrix , array , list , data frames)

1. Vectors

- **Definition**: A vector is a one-dimensional array that holds elements of the same type (e.g., numeric, character, logical).
- elements of a vector are known as component
- **Creation**: Use the c() function.

- if we have to find the length of a vector then use length() function.
- For indexing we can use[] (in R indexing starts from 1).
- We use names function for give the names() of vector elements.

Example:1

```
numeric_vector <- c(1, 2, 3, 4)
char_vector <- c("apple", "banana", "cherry")
```

```
x<-1:10
```

by the use of seq() function

```
sq<-seq(1,3,length.out=5)
    seq(from=3.5,to=1.5,by=.5)
    seq(from=-2.7,to=1.5,length.out=.5)
```

2. Lists

- **Definition**: A list is a one-dimensional array that can hold elements of different types, including other lists.
- **Creation**: Use the list() function.

```
my_list <- list(name="Alice", age=25, scores=c(90, 85, 88))
```

example

```
my_list <- list(c("rer","ter","ttt"), c(55,65,70), c("b.com","b.b.a.","b.ca."))
my_list
#here we give name of each list in my_list.
names(my_list)=c("name","roll no.","department")
my_list
```

- For indexing the element of list we can use print(my_list[1]) or print(my_list$`roll no.`)

- V<-unlist(my_list) function convert list into vector.

3. Matrices

- **Definition**: A matrix is a two-dimensional array where all elements must be of the same type.
- **Creation**: Use the matrix() function.
- Matrix(data,nrow,ncolumn,byrow,dim_name)

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


my_matrix <- matrix(1:9, nrow=3, ncol=3,byrow=FALSE)
my_matrix
```

4. Arrays

- **Definition**: An array is a multi-dimensional extension of a matrix. It can have more than two dimensions.
- **Creation**: Use the array() function.

```
my_array <- array(1:24, dim=c(2, 3, 4))



v1<-c(12:17)
v2<-c(15,17,19,20)
row_name<-c("r1","r2","r3")
col_name<-c("c1","c2","c3")
mat_name<-c("mat1","mat2")
my<-
+array(c(v1,v2),dim=c(3,3,2),dimnames=list(row_name,col_name,mat_n
ame))#give name to row column and matrix
my
c<-print(my[3,2,2])#indexing the element of array
```

- We also use add or substract two array like v1+v2.

5. Data Frames

- **Definition**: A data frame is a two-dimensional table-like structure where each column can be of a different type. It is similar to a spreadsheet or SQL table.
- **Creation**: Use the data.frame() function.

```
my_df <- data.frame(
Name = c("ram", "shyam", "raj"),
Age = c(25, 30, 35),
Score = c(90, 85, 88)
)
```

- We can convert data frame into string by the use of str() function.
- We can indexing in data frame with the help of [] bracket and $ sign.

- Ex:
  print(my_df[1])
  print(my_df[1,2])
  print(my_df[1,])
  print(my_df([,3])
  my_df$name
  my_df$Name<-c("aa","bb","cc")
  my_df

- We can add the new column and row by the use of cbind() and rbind() function in data frame.
  Ex:
    cbind(my_df,assign=c(78,76,78))
    rbind(my_df,c("AA",40,88,77))
- Also we can combine two data frame with the help of cbind() and rbind() function.
  Ex: v<-rbind(my_df1,my_df2)
- We can check the dimention of data frame with the use of dim() function.
  Ex: dim(my_df) #give the number of row and column in data frame.

- we can get the sub data frame by the help of subset() function.

Ex: subset(my_df,Name!="aa")

- <u>Importing data into R from different file formats (CSV, Excel).</u>

1. CSV Files

CSV (Comma Separated Values) files are one of the most straightforward formats to import into R.

# Assuming your CSV file is named 'data.csv' and is located in your working directory
data <- read.csv("data.csv")

- Some function for csv file

getwd()=give current working directort

```
setwd("f:material/r") =set current working directort

aa<-read.csv("people.csv")
View(aa)
fix(aa)=fixong the csv file
str(aa)=show the structure of the data frame
summary(aa)=give statistical summary of the csv file
names(aa)=provide all the variable name.
nrow(aa)=provide number of row.
ncol(aa)=provide number of colume.
length(aa)=give length of file.
dim(aa)=show the dimention of the data frame.
colnames(aa)=return column names.
head(aa)=show first six row of file.
tail(aa)=show last six row of file.
bb<-aa[c(1:2,3,7)]=give 1,2.3.7 column of file
View(bb)
cc<-aa[c(1:3),c(1:3)]=provide first three column and three row of data
View(cc)
names(aa)
vv<-aa$User.Id[]=for indexing the data value.
Vv
ff<-subset(aa,Sex=="Male")
View(ff)
```

2. Excel Files

For Excel files, you typically need to use the readxl package, which provides functions to read Excel files into R.
First, make sure to install the readxl package if you haven't already:

```
install.packages("readxl")
```

Then, you can use the read_excel() function to import data from Excel:

```
library(readxl)

# Assuming your Excel file is named 'data.xlsx' and is located in your
working directory
data <- read_excel("data.xlsx")
```

UNIT: 4 DATA FILTERING AND CLEANING

- Sub setting and filtering data in R
- Adding removing and renaming variables in R
- Data cleaning and transformation
- Identifying and handling missing values
- Data type conversion and recording variables

• Subsetting and filtering data in R

In R, subsetting and filtering data are common operations used to extract specific subsets of data from a larger dataset based on certain conditions or criteria. Here's how you can approach subsetting and filtering:

Subsetting Data

1. **Subsetting by Row and Column Indices:**

   - To subset data by specific rows and columns, you can use square brackets `[ ]`.
   - Example: `subset_data <- data[1:10, c("column1", "column2")]`
   - This extracts the first 10 rows and columns "column1" and "column2" from `data`.

2. **Subsetting by Logical Conditions:**

   - You can subset data based on logical conditions using square brackets `[ ]` with logical expressions.
   - Example: `subset_data <- data[data$column1 > 50, ]`
   - This selects rows from `data` where values in `column1` are greater than 50.

3. **Subsetting by Column Name:**

   - You can subset data by column names using `$` or `[]`.
   - Example: `subset_column <- data$column1`
   - This extracts the column named "column1" from `data`.

Filtering Data

1. **Using the `subset()` Function:**

   - The `subset()` function in R allows for more complex filtering based on logical conditions.
   - Example: `subset_data <- subset(data, column1 > 50 & column2 == "value")`

- This creates `subset_data` by filtering `data` where `column1` values are greater than 50 and `column2` equals "value".

2. **Using the `dplyr` Package:**

- The `dplyr` package provides a more intuitive way to filter data using functions like `filter()` and `select()`.
- Example:

```
library(dplyr)
```

  - `filtered_data <- a %>%`
  - `    filter(Descriptive > 40) %>%`
  - `    select(Descriptive,Roll)`
  - `View(filtered_data)`
  - This filters `data` to include rows where `Descriptive` is greater than 40 and selects only `Descriptive` and `Roll no.`

Notes:

- **Square Brackets `[ ]`:** Used for basic subsetting and filtering operations.
- **`subset()` Function:** Convenient for straightforward filtering tasks.
- **`dplyr` Package:** Provides a more readable and chainable syntax for data manipulation tasks.

• Adding removing and renaming variables in R

n R, adding, removing, and renaming variables (columns) in data frames can be achieved using various functions and techniques. Here's how you can perform these operations:

Adding Variables

To add a new variable (column) to an existing data frame in R:

1. **Using `$` Operator:**

- Directly assign values to a new column using the `$` operator.

```
data$new_column <- c(1, 2, 3, 4, 5)  # Assuming `data` is
your existing data frame
```

- This creates a new column named `new_column` in `data` with values 1, 2, 3, 4, and 5.

2. **Using `cbind()` :**

- Combine the existing data frame with the new column using `cbind()`.

```
new_column <- c(1, 2, 3, 4, 5)
data <- cbind(data, new_column)
```

- This appends `new_column` to the existing `data` data frame.

3. **Using the `mutate()` Function from `dplyr`:**

- If you are working with the `dplyr` package, you can use `mutate()` to add a new column based on existing data.

```
library(dplyr)
data <- data %>%
mutate(new_column = c(1, 2, 3, 4, 5))
```

- This adds `new_column` with specified values to `data`.

Removing Variables

To remove (delete) variables from a data frame in R:

1. **Using `subset()` Function:**

- Create a new data frame excluding the variables to be removed.

```
data <- subset(data, select = -c(column_to_remove1,
column_to_remove2))
```

- This removes `column_to_remove1` and `column_to_remove2` from `data`.

2. **Using the `$` Operator with `NULL`:**

- Set a column to `NULL` to remove it from the data frame.

```
data$column_to_remove <- NULL
```

- This removes `column_to_remove` from `data`.

3. **Using the `select()` Function from `dplyr`:**

- Use `select()` to exclude columns from `data`.

```
library(dplyr)
data <- select(data, -column_to_remove)
```

- This removes `column_to_remove` from `data`.

Renaming Variables

To rename variables (columns) in a data frame in R:

1. **Using the `names()` Function:**

   If you have a data frame and you want to rename its columns, you can use the `names()` functions.

   ```
   df <- data.frame(old_name1 = 1:3, old_name2 = 4:6)
      # Rename columns
   names(df) <- c("new_name1", "new_name2")
   ```

- Assign new names directly to `names()` function.
  ```
  names(data)[which(names(data) == "old_name")] <-
  "new_name"
  ```

- This renames `old_name` to `new_name` in `data`.

2. **Using the `colnames()` Function:**

- Similarly, you can use `colnames()` for renaming columns.

  ```
  colnames(data)[which(colnames(data) == "old_name")] <-
  "new_name"
  ```

3. **Using the `rename()` Function from `dplyr`:**

- If using `dplyr`, you can use `rename()` to rename columns.

  ```
  library(dplyr)
  data <- data %>%
  rename(new_name = old_name)
  ```

- This renames `old_name` to `new_name` in `data`.

Notes:

- **Be cautious with data manipulation:** Always make sure you understand the structure of your data frame and how changes will affect your analysis.
- **Using packages:** Functions from the base R and `dplyr` package provide convenient ways to add, remove, and rename variables in data frames, depending on your preference and workflow.

- **Data cleaning and transformation**

Data cleaning and transformation are crucial steps in preparing data for analysis in R. They involve handling missing values, correcting data types, transforming variables, and more. Here's a structured approach to perform data cleaning and transformation in R:

Data Cleaning

1. **Handling Missing Values:**

- Identify missing values (`NA`, `NaN`, empty strings, etc.) in your data using functions like `is.na()` or `complete.cases()`.
- Replace or impute missing values using functions like `na.omit()`, `complete()` (from `tidyr`), or `impute()` (from `imputeMissings` package).

```
# Example: Replace missing values with mean
data$column[is.na(data$column)]    <-    mean(data$column,
na.rm = TRUE)
```

2. **Removing Duplicates:**

   - Remove duplicate rows using `duplicated()` and `unique()` functions.

```
data <- unique(data)
```

3. **Handling Outliers:**

   - Identify and handle outliers using statistical methods like z-score, IQR (interquartile range), or domain-specific knowledge.

```
# Example: Remove outliers.
# Sample data
data <- c(1, 2, 3, 4, 5, 100)

# Identify outliers using IQR
Q1 <- quantile(data, 0.25)
Q3 <- quantile(data, 0.75)
IQR <- Q3 - Q1
outliers <- data[data < (Q1 - 1.5 * IQR) | data > (Q3 +
1.5 * IQR)]

# Remove outliers
data_clean <- data[!data %in% outliers]

# Output results
print(outliers)      # Print outliers
print(data_clean)    # Print cleaned data
```

Data Transformation

1. **Creating New Variables:**

   - Generate new variables based on existing data using arithmetic operations or functions.

   ```
   data$new_variable <- data$var1 + data$var2
   ```

2. **Reshaping Data:**

   - Reshape data using functions like `melt()` and `cast()` from the `reshape2` package or `pivot_longer()` and `pivot_wider()` from the `tidyr` package.

i) Wide to Long Format

To reshape data from a wide format to a long format (and vice versa), you typically use functions from the `tidyr` or `reshape2` packages.

Example:

```
# Load tidyr
library(tidyr)

# Example data in wide format
data_wide <- data.frame(
  id = 1:3,
  treatment_1 = c(2, 4, 5),
  treatment_2 = c(3, 6, 9)
)

# Convert to long format
data_long <- pivot_longer(data_wide,
                          cols = starts_with("treatment"),
                          names_to = "treatment",
                          values_to = "value")

print(data_long)
```

Explanation:

- `pivot_longer` is used to transform columns into rows.
- `cols` specifies which columns to pivot.
- `names_to` specifies the name of the new column that will contain the names of the old columns.
- `values_to` specifies the name of the new column that will contain the values.

Example:

```
# Load reshape2
library(reshape2)

# Example data in wide format
data_wide <- data.frame(
  id = 1:3,
  treatment_1 = c(2, 4, 5),
  treatment_2 = c(3, 6, 9)
)

# Convert to long format
data_long <- melt(data_wide, id.vars = "id",
                   variable.name = "treatment",
                   value.name = "value")

print(data_long)
```

Explanation:

- `melt` function converts wide data to long format.
- `id.vars` specifies columns to keep as identifier variables.
- `variable.name` and `value.name` specify the names of the new columns.

ii) Longer to Wide Format

To convert data from a long format to a wide format, you can use `tidyr` or `reshape2` functions.

Example:

```
# Load tidyr
library(tidyr)

# Example data in long format
data_long <- data.frame(
  id = rep(1:3, each = 2),
  treatment = c("treatment_1", "treatment_2", "treatment_1",
"treatment_2", "treatment_1", "treatment_2"),
  value = c(2, 3, 4, 6, 5, 9)
)

# Convert to wide format
data_wide <- pivot_wider(data_long,
                         names_from = treatment,
                         values_from = value)

print(data_wide)
```

Explanation:

- `pivot_wider` is used to transform rows into columns.
- `names_from` specifies the column that will become the new column names.
- `values_from` specifies the column that contains the values.

Example:

```
# Load reshape2
library(reshape2)

# Example data in long format
data_long <- data.frame(
  id = rep(1:3, each = 2),
  treatment = c("treatment_1", "treatment_2", "treatment_1",
"treatment_2", "treatment_1", "treatment_2"),
  value = c(2, 3, 4, 6, 5, 9)
)

# Convert to wide format
data_wide <- dcast(data_long, id ~ treatment, value.var =
"value")

print(data_wide)
```

Explanation:

- `dcast` converts long data to wide format.
- `id ~ treatment` specifies the formula where `id` remains as rows and `treatment` values become columns.
- `value.var` specifies the name of the column to use for the cell values.

Notes:

- **Documentation:** Document your data cleaning and transformation steps to maintain transparency and reproducibility.
- **Packages:** Utilize packages like `dplyr`, `tidyr`, `reshape2`, and others for efficient and readable data manipulation.
- **Iterative Process:** Data cleaning and transformation are often iterative processes, requiring exploration and adjustment based on insights gained during analysis.

By following these guidelines and using appropriate R functions and packages, you can effectively clean and transform your data to prepare it for further analysis or modeling tasks

- ## Identifying and handling missing values

Identifying and handling missing values is a critical part of data cleaning and preprocessing in R. Missing values can occur due to various reasons such as data entry errors, equipment malfunction, or non-response in surveys. Here's how you can identify and handle missing values effectively in R:

Identifying Missing Values

Check for Missing Values:

- Use functions like `is.na()` or `is.null()` to detect missing values in your data frame.

```
# Check for missing values in entire data frame
any(is.na(data))

# Check for missing values in specific columns
colSums(is.na(data))
```

The `any(is.na(data))` function checks if there are any missing values in the entire data frame `data`. The `colSums(is.na(data))` function calculates the number of missing values (`NA`) in each column of the data frame `data`.

Handling Missing Values

1. **Removing Missing Values:**

   - Remove rows or columns containing missing values using functions like `na.omit()` or `complete.cases()`.

```
# Remove rows with any missing values
data_clean <- na.omit(data)

# Remove rows with missing values in specific columns
data_clean      <-      data[complete.cases(data$column1,
data$column2), ]
```

The `na.omit(data)` function removes rows from `data` that contain any missing values (`NA`). The `complete.cases(data$column1, data$column2)` function removes rows with missing values in `column1` or `column2`.

2. **Imputing Missing Values:**

   - Replace missing values with estimated values such as mean, median, mode, or predictive models.

```
# Impute missing values with mean of the column
data$column[is.na(data$column)]    <-    mean(data$column,
na.rm = TRUE)
```

This example replaces missing values (`NA`) in `column` with the mean value of `column`, ignoring `NA` values (`na.rm = TRUE`).

Notes:

- **Data Context:** Understand the context and reasons behind missing values to choose appropriate handling techniques.
- **Documentation:** Document your approach to handling missing values to ensure transparency and reproducibility.
- **Validation:** Validate the impact of missing data handling on analysis results to ensure robustness.

By applying these techniques in R, you can effectively identify and manage missing values in your datasets, ensuring that your data is ready for further analysis or modeling tasks

• data type conversion and recording variables

In R, data type conversion and handling variables are essential tasks when preparing data for analysis or modeling. Here's how you can perform data type conversion, record variables, and manage them effectively:

Data Type Conversion

1. **Convert Numeric Data to Character and Vice Versa:**

- Use `as.character()` and `as.numeric()` functions to convert between numeric and character data types.

```
# Convert numeric to character
data$numeric_column <- as.character(data$numeric_column)

# Convert character to numeric
data$character_column                                  <-
as.numeric(data$character_column)
```

Ensure that the character data can be correctly converted to numeric (e.g., no non-numeric characters).

2. **Convert Factors to Character or Numeric:**

- Use `as.character()` or `as.numeric()` to convert factors to character or numeric data types, respectively.

```
# Convert factor to character
data$factor_column <- as.character(data$factor_column)
```

```
# Convert factor to numeric (factor levels should be
numeric)
data$factor_column <-
as.numeric(as.character(data$factor_column))
```

Be cautious when converting factors to numeric, as it converts the levels of the factor, not the underlying values.

3. **Convert Dates and Times:**

- Use `as.Date()` or `as.POSIXct()` functions to convert character or numeric data to date or time formats.

```
# Convert character to date
data$date_column <- as.Date(data$date_column, format =
"%Y-%m-%d")

# Convert numeric POSIX time to date-time format
data$datetime_column <-
as.POSIXct(data$numeric_time_column, origin = "1970-01-
01")
```

Specify the correct format (`format = "%Y-%m-%d"`) when converting character data to date.

Recording Variables

1. **Creating New Variables:**

- Generate new variables based on existing data or computations.

```
# Create a new variable based on existing columns
data$new_variable <- data$existing_column1 +
data$existing_column2
```

Use arithmetic operations or functions to create new variables from existing ones.

2. **Renaming Variables:**

- Use `names()` or `colnames()` to rename variables in a data frame.

```
# Rename a variable
names(data)[which(names(data) == "old_name")] <-
"new_name"]
```

Ensure the new names are meaningful and consistent with your data.

3. **Removing Variables:**

- Use `subset()`, `select()` from `dplyr`, or indexing to remove variables from a data frame.

  ```
  # Remove a variable using subset
  data <- subset(data, select = -c(variable_to_remove))

  # Remove a variable using select from dplyr
  library(dplyr)
  data <- select(data, -variable_to_remove)
  ```

  Choose appropriate methods to remove variables based on your workflow and preferences.

Notes:

- **Data Integrity:** Ensure data type conversions maintain data integrity and correctness.
- **Documentation:** Document your data type conversions and variable management steps for clarity and reproducibility.
- **Validation:** Validate data type conversions and variable operations to avoid unintended consequences in your analysis or modeling process.

By following these guidelines and using appropriate functions in R, you can effectively handle data type conversions, create new variables, rename variables, and manage variables in your datasets to prepare them for further analysis or modeling tasks.

Unit: 5
Working With Data In R

1. Reordering and reshaping data frames
2. Merging and joining data frames.
3. Calculating summary statistics (mean, median, mode, standard deviation).
4. Generating frequency tables and cross-tabulations.
5. Commands to measures of central tendency and dispersion.
6. Concepts of normal distribution
7. Commands to explore view data distributions graphically (Bell curve).

• Reordering and reshaping data frames in R

Reordering and reshaping data frames in R is a common task that can be done using various functions and packages. Below are some common techniques and functions for these tasks:

Reordering Data Frames

1. Reordering Rows

To reorder rows based on a particular column, you can use the `order()` function.

```
# Sample data frame
df <- data.frame(
  ID = c(1, 2, 3, 4),
  Value = c(10, 30, 20, 40)
)

# Reorder rows based on the 'Value' column in ascending order
df_sorted <- df[order(df$Value), ]
print(df_sorted)

# For descending order
df_sorted_desc <- df[order(df$Value, decreasing = TRUE), ]
print(df_sorted_desc)
```

2. Reordering Columns

To reorder columns, you can use column indexing.

```
# Reorder columns
df_reordered <- df[, c("Value", "ID")]
print(df_reordered)
```

Reshaping Data Frames

1. Melting and Casting Data Frames

To reshape data frames from wide to long format and vice versa, the `reshape2` package (or its successor `data.table`) is useful.

Using `reshape2`:

```
library(reshape2)

# Sample data frame in wide format
df_wide <- data.frame(
  ID = c(1, 2),
  Q1 = c(10, 20),
  Q2 = c(30, 40)
)

# Melt the data frame to long format
df_long <- melt(df_wide, id.vars = "ID")
print(df_long)

# Cast back to wide format
df_wide_again <- dcast(df_long, ID ~ variable)
print(df_wide_again)
```

Using `data.table`:

```
library(data.table)

# Sample data frame
df <- data.table(
  ID = c(1, 2),
  Q1 = c(10, 20),
  Q2 = c(30, 40)
)

# Melt to long format
df_long <- melt(df, id.vars = "ID")
print(df_long)

# Cast back to wide format
df_wide <- dcast(df_long, ID ~ variable)
print(df_wide)
```

- ## Merging and joining data frames in R

Merging and joining data frames in R can be achieved through several functions, with the most commonly used being from base R and the `dplyr` package. Here's a brief guide on how to handle these operations:

Using Base R

1. `merge()`

The `merge()` function is versatile and allows you to join data frames by common columns or row names.

```
# Sample data frames
df1 <- data.frame(ID = c(1, 2, 3), Value1 = c("A", "B", "C"))
df2 <- data.frame(ID = c(2, 3, 4), Value2 = c("X", "Y", "Z"))

# Inner join (default)
```

```
merged_df <- merge(df1, df2, by = "ID")
print(merged_df)

# Left join
left_joined_df <- merge(df1, df2, by = "ID", all.x = TRUE)
print(left_joined_df)

# Right join
right_joined_df <- merge(df1, df2, by = "ID", all.y = TRUE)
print(right_joined_df)

# Full outer join
full_joined_df <- merge(df1, df2, by = "ID", all = TRUE)
print(full_joined_df)
```

- Calculating summary statistics (mean, median, mode, standard deviation).

Calculating summary statistics such as the mean, median, mode, and standard deviation is essential for data analysis. Here's how you can compute these statistics in R:

Basic Summary Statistics in R

1. Mean

The mean is the average of the numbers.

```
# Sample vector
data <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Calculate mean
mean_value <- mean(data)
print(mean_value)
```

2. Median

The median is the middle value when the numbers are sorted.

```
# Calculate median
median_value <- median(data)
print(median_value)
```

3. Mode

R does not have a built-in function for mode, but you can create one.

```
# Function to calculate mode
get_mode <- function(x) {
  uniq_x <- unique(x)
  uniq_x[which.max(tabulate(match(x, uniq_x)))]
}

# Calculate mode
mode_value <- get_mode(data)
print(mode_value)
```

Note: If the data is multimodal (more than one mode), this function will only return one mode. You can modify it to return all modes if needed.

4. Standard Deviation

The standard deviation measures the amount of variation or dispersion in the data.

```
# Calculate standard deviation
std_dev <- sd(data)
print(std_dev)
```

Summary Statistics for Data Frames

You can also calculate these statistics for columns in a data frame.

```
# Sample data frame
df <- data.frame(
  A = c(1, 2, 2, 3, 4, 4, 4, 5, 6),
  B = c(7, 8, 8, 8, 10, 11, 12, 12, 12)
)

# Calculate statistics for column A
mean_A <- mean(df$A)
median_A <- median(df$A)
mode_A <- get_mode(df$A)
std_dev_A <- sd(df$A)

print(mean_A)
print(median_A)
print(mode_A)
print(std_dev_A)

# Calculate statistics for column B
mean_B <- mean(df$B)
median_B <- median(df$B)
mode_B <- get_mode(df$B)
std_dev_B <- sd(df$B)

print(mean_B)
print(median_B)
print(mode_B)
print(std_dev_B)
```

Using `dplyr` for Summary Statistics

The `dplyr` package can simplify calculations, especially for data frames.

```
library(dplyr)

# Sample data frame
df <- tibble(
  A = c(1, 2, 2, 3, 4, 4, 4, 5, 6),
  B = c(7, 8, 8, 8, 10, 11, 12, 12, 12)
)

# Summary statistics with dplyr
summary_stats <- df %>%
  summarise(
    Mean_A = mean(A),
    Median_A = median(A),
    Mode_A = get_mode(A),
    SD_A = sd(A),
    Mean_B = mean(B),
    Median_B = median(B),
    Mode_B = get_mode(B),
    SD_B = sd(B)
  )

print(summary_stats)
```

Additional Considerations

- **Handling Missing Values:** Use the `na.rm = TRUE` parameter to exclude `NA` values in calculations.

  ```
  mean_value <- mean(data, na.rm = TRUE)
  ```

- **Customizing Mode Calculation:** For datasets with multiple modes, you might want to return all modes.

```
# Function to return all modes
  get_modes <- function(x) {
  uniq_x <- unique(x)
  freq <- table(x)
  modes <- names(freq[freq == max(freq)])
  as.numeric(modes)
}

# Calculate all modes
modes_value <- get_modes(data)
print(modes_value)
```

These methods will help you compute and analyze summary statistics efficiently in R.

- ## Generating frequency tables and cross-tabulations in R

Generating frequency tables and cross-tabulations is a crucial part of data analysis in R. Here's how you can do it using base R functions and `dplyr` for more complex tasks.

Frequency Tables

Using Base R

1. **Single Variable Frequency Table**

   Use the `table()` function to get the frequency of each unique value in a vector.

   ```
   # Sample data
   data <- c("apple", "banana", "apple", "orange", "banana", "banana",
   "apple")
   ```

```
# Frequency table
freq_table <- table(data)
print(freq_table)
```

2. **Frequency Table for Multiple Variables**

   For more than one variable, `table()` can handle this directly.

   ```
   # Sample data frame
   df <- data.frame(
     Fruit = c("apple", "banana", "apple", "orange", "banana", "banana",
   "apple"),
     Color = c("red", "yellow", "red", "orange", "yellow", "yellow",
   "red")
   )

   # Cross-tabulation
   cross_tab <- table(df$Fruit, df$Color)
   print(cross_tab)
   ```

Advanced Cross-Tabulations

Using `xtabs` in Base R

The `xtabs()` function creates contingency tables from a formula and data frame.

```
# Sample data frame
df <- data.frame(
  Fruit = c("apple", "banana", "apple", "orange", "banana", "banana",
"apple"),
  Color = c("red", "yellow", "red", "orange", "yellow", "yellow", "red")
)

# Cross-tabulation
cross_tab_xtabs <- xtabs(~ Fruit + Color, data = df)
print(cross_tab_xtabs)
```

Using `CrossTable` from the `gmodels` package

The `CrossTable` function provides detailed cross-tabulations.

```
library(gmodels)

# Sample data frame
df <- data.frame(
  Fruit = c("apple", "banana", "apple", "orange", "banana", "banana",
"apple"),
  Color = c("red", "yellow", "red", "orange", "yellow", "yellow", "red")
)

# Cross-tabulation with detailed output
CrossTable(df$Fruit, df$Color)
```

Summary

- **Frequency Table for Single Variable:** Use `table()` in base R or `count()` in `dplyr`.
- **Cross-Tabulation:** Use `table()` or `xtabs()` in base R, or `count()` in `dplyr`. For detailed cross-tabulations, consider `CrossTable()` from the `gmodels` package.

These methods allow you to summarize and explore categorical data effectively.

- ## Commands to measures of central tendency and dispersion.

To compute measures of central tendency and dispersion in R, you can use several built-in functions. Here's a detailed guide on the commands for each measure:

Measures of Central Tendency

1. **Mean**

   The mean (average) is calculated using the `mean()` function.

   ```
   # Sample vector
   data <- c(10, 20, 30, 40, 50)

   # Calculate mean
   mean_value <- mean(data)
   print(mean_value)
   ```

2. **Median**

   The median is calculated using the `median()` function.

   ```
   # Calculate median
   median_value <- median(data)
   print(median_value)
   ```

3. **Mode**

   R does not have a built-in mode function, but you can define one.

   ```
   # Function to calculate mode
   get_mode <- function(x) {
     uniq_x <- unique(x)
     freq <- table(x)
     mode_value <- uniq_x[which.max(freq)]
     return(mode_value)
   }

   # Calculate mode
   mode_value <- get_mode(data)
   print(mode_value)
   ```

Measures of Dispersion

1. **Standard Deviation**

   The standard deviation measures the amount of variation in the data. Use `sd()`.

```
# Calculate standard deviation
std_dev <- sd(data)
print(std_dev)
```

2. **Variance**

   The variance measures the spread of the data points. Use `var()`.

   ```
   # Calculate variance
   variance <- var(data)
   print(variance)
   ```

3. **Range**

   The range is the difference between the maximum and minimum values. You can calculate it using `range()`.

   ```
   # Calculate range
   data_range <- range(data)
   range_value <- diff(data_range)
   print(range_value)
   ```

   Alternatively, you can use:

   ```
   # Direct calculation of range
   range_value <- max(data) - min(data)
   print(range_value)
   ```

Using `dplyr` for Summary Statistics

You can also compute these measures for data frames using the `dplyr` package.

1. **Install and Load `dplyr`**

   ```
   install.packages("dplyr")
   library(dplyr)
   ```

2. **Compute Summary Statistics**

   ```
   # Sample data frame
   df <- tibble(value = c(10, 20, 30, 40, 50))

   # Calculate summary statistics
   summary_stats <- df %>%
     summarise(
       Mean = mean(value),
       Median = median(value),
       Standard_Deviation = sd(value),
       Variance = var(value),
       Range = max(value) - min(value)
     )

   print(summary_stats)
   ```

Summary of Commands

- **Mean:** `mean()`
- **Median:** `median()`
- **Mode:** Custom function `get_mode()`
- **Standard Deviation:** `sd()`
- **Variance:** `var()`
- **Range:** `range()` with `diff()` or `max() - min()`

These functions will help you compute and analyze the central tendency and dispersion of your data effectively.

- ## Concepts of normal distribution in R

In R, you can work with normal distribution using various functions and packages. Here's a detailed overview of how to apply and understand the concepts of normal distribution in R:

1. Generating Normal Distribution Data

You can generate random data that follows a normal distribution using the `rnorm()` function.

```
# Generate 1000 random numbers from a normal distribution with mean 0 and
standard deviation 1
data <- rnorm(1000, mean = 0, sd = 1)
```

2. Visualizing Normal Distribution

You can visualize the distribution using a histogram or a density plot.

```
# Histogram
hist(data, breaks = 30, main = "Histogram of Normally Distributed Data",
xlab = "Value", col = "lightblue", border = "black")

# Density plot
plot(density(data), main = "Density Plot of Normally Distributed Data",
xlab = "Value", ylab = "Density")
```

3. Quantile Function

You can find the quantile for a given probability using `qnorm()`.

```
# Quantile for a probability of 0.95, mean = 0, sd = 1
quantile_value <- qnorm(0.95, mean = 0, sd = 1)
print(quantile_value)
```

4. Using `ggplot2` for Visualization

For more advanced visualization, you can use the `ggplot2` package.

```
library(ggplot2)
```

```
# Create a data frame
df <- data.frame(value = data)

# Plot with ggplot2
ggplot(df, aes(x = value)) +
  geom_histogram(aes(y = ..density..), bins = 30, fill = "lightblue", color
= "black") +
  geom_density(color = "red") +
  labs(title = "Histogram and Density Plot of Normally Distributed Data", x
= "Value", y = "Density")
```

Summary of Key Functions

- **`rnorm(n, mean, sd)`**: Generate `n` random numbers from a normal distribution with specified mean and standard deviation.
- **`dnorm(x, mean, sd)`**: Compute the density of the normal distribution at `x`.
- **`pnorm(q, mean, sd)`**: Compute the cumulative probability up to quantile `q`.
- **`qnorm(p, mean, sd)`**: Compute the quantile for a given probability `p`.
- **`hist()`**: Create histograms for visualizing the distribution.
- **`density()`**: Create density plots.
- **`ggplot2`**: For advanced and customizable visualizations.

These commands and functions will help you understand and work with normal distributions effectively in R.

- ## **Commands to explore view data distributions graphically (Bell curve) in R**

To explore and visualize data distributions graphically, especially if you want to visualize how closely your data follows a bell curve (normal distribution), you can use several commands and packages in R. Below are various methods and commands to help you view and interpret data distributions graphically.

1. Histogram with Density Plot

A histogram with a superimposed density plot is a common way to visualize the distribution of data.

```
# Sample data
data <- rnorm(1000, mean = 0, sd = 1)

# Basic histogram with density plot
hist(data, breaks = 30, probability = TRUE, main = "Histogram with Density
Plot", xlab = "Value", col = "lightblue", border = "black")
lines(density(data), col = "red", lwd = 2)
```

2. Q-Q Plot

A Q-Q (quantile-quantile) plot compares the quantiles of your data against the quantiles of a theoretical normal distribution.

```
# Q-Q plot
qqnorm(data, main = "Q-Q Plot")
qqline(data, col = "red", lwd = 2)
```

3. Density Plot

A density plot provides a smoothed estimate of the distribution of your data.

```
# Density plot
plot(density(data), main = "Density Plot", xlab = "Value", ylab =
"Density", col = "blue", lwd = 2)
```

4. Using `ggplot2` for Advanced Visualization

The `ggplot2` package allows for more customizable and advanced visualizations.

1. **Install and Load `ggplot2`**

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

2. **Histogram with Density Plot**

   ```
   # Create a data frame
   df <- data.frame(value = data)

   # Histogram with density plot
   ggplot(df, aes(x = value)) +
     geom_histogram(aes(y = ..density..), bins = 30, fill = "lightblue",
   color = "black") +
     geom_density(color = "red") +
     labs(title = "Histogram with Density Plot", x = "Value", y =
   "Density")
   ```

5. Normal Probability Plot

A normal probability plot is another way to visualize if the data follows a normal distribution.

```
# Normal probability plot
qqnorm(data, main = "Normal Probability Plot")
qqline(data, col = "red", lwd = 2)
```

6. Bell Curve Overlay

To overlay a normal distribution curve on your histogram:

```
# Generate a sequence of values from min to max of your data
x <- seq(min(data), max(data), length = 100)
# Compute the density of the normal distribution
y <- dnorm(x, mean = mean(data), sd = sd(data))

# Histogram with normal distribution curve
hist(data, breaks = 30, probability = TRUE, main = "Histogram with Normal
Distribution Curve", xlab = "Value", col = "lightblue", border = "black")
lines(x, y, col = "red", lwd = 2)
```

Summary

- **Histogram with Density Plot:** Use `hist()` and `lines(density())`.
- **Q-Q Plot:** Use `qqnorm()` and `qqline()`.
- **Density Plot:** Use `plot(density())`.
- **`ggplot2`:** For more customizable visualizations, use `ggplot2` functions like `geom_histogram()` and `geom_density()`.
- **Normal Probability Plot:** Similar to Q-Q plot, use `qqnorm()` and `qqline()`.
- **ECDF Plot:** Use `plot(ecdf())`.

These methods will help you graphically explore and understand the distribution of your data, including how closely it follows a normal (bell curve) distribution.