Unit: II

Reading and writing files, Programming, Calling Functions, Conditions and Loops: stand- alone statement with illustrations in exercise 10.1, stacking statements, coding loops, Writing Functions, Exceptions, Timings, and Visibility

R-Ready Data-Sets

- ✓ In R, a **dataset** refers to a collection of data, typically organized in a structured form like a table or a matrix, where rows represent observations and columns represent variables.
- ✓ Datasets are a core component in data analysis and can be used for a wide range of tasks such as data manipulation, statistical analysis, and visualization
- ✓ R provides built-in data-sets.
- ✓ Data-sets are also present in user-contributed-packages.
- ✓ The datasets are useful for data analysis and statistical modelling.
- ✓ **data()** can be used to access a list of the data-sets.
- ✓ The list of available data-sets is organized
  - alphabetically by name and
  - grouped by package.
- ✓ The availability of the data-sets depends on the installed contributed-packages.

Built-in Data-Sets

- ✓ These datasets are included in the base R installation
- ✓ These data-sets are found in the package called "datasets."

For example,

R> library(help="datasets") **#To view summary of the data-sets within the package**

R> Help(“ChickWeight”) **#To get info about the "ChickWeight" data-set**

R> ChickWeight[1:5**,]** **# To display the first 5 records of ChickWeight**

Contributed Data-Sets

- ✓ Contributed datasets are created by the R-community.
- ✓ The datasets are not included in the base R installation.
- ✓ But the datasets are available through additional packages.
- ✓ You can install and load additional packages containing the required datasets.

For example,

R> install.packages("tseries")

R> library("tseries")

Reading in External Data Files

- ✓ Reading files in R is a fundamental operation for importing external data into R for analysis

Reading CSV Files

- ✓ CSV (Comma-Separated Values) is one of the most common file formats for datasets. R provides the read.csv () function to read CSV files

Syntax

```
read.csv(file, header = TRUE, sep = ",")
```

Where

- ✓ **file:** The name of the file from which data should be read.
- ✓ **header:** This indicates whether the first row of the file contains column names Default is FALSE.
- ✓ **sep:** This represents the field separator character.

Example

```
data <- read.csv("path/to/your/file.csv")
# Check the first few rows
head(data)
```

Reading Text Files

- ✓ For plain text files with custom delimiters (e.g., tab-separated, space-separated), you can use **the read.table() function**.

Syntax

```
read.table(file, header = FALSE, sep = "").
```

where,

- ✓ **file:** The name of the file from which data should be read.
- ✓ **header:** This indicates whether the first row of the file contains column names Default is FALSE.
- ✓ **sep:** This represents the field separator character

Example

```
# Read a text file with space-separated values
data <- read.table("path/to/your/file.txt", header = TRUE)
# Read a text file with a custom separator (e.g., tab-separated)
data <- read.table("file.txt", sep = "\t", header = TRUE)
```

Reading Excel Files

- ✓ To read Excel files (.xls or .xlsx), you can use the readxl package, which provides the read_excel() function.

Syntax

```
read_excel(path, sheet = NULL, range = NULL, col_names = TRUE,  col_types = NULL, )
```

Where

- ✓ **sheet:** Specifies the sheet to read.Default: NULL (the first sheet will be used).
- ✓ **range:**A range of cells to read, given in Excel-style notation (e.g., "A1:D10").
- ✓ **col_names:**Logical value indicating whether to use the first row as column names.
- ✓ col_types:A character vector specifying the type of each column (e.g., "numeric", "text", "date"). If NULL, column types are guessed automatically.

Example

```
# Read Excel file
data <- read_excel("path/to/your/file.xlsx")
# Read a specific sheet
data <- read_excel("file.xlsx", sheet = "Sheet1")
```

Reading Data from Databases

- ✓ To read data from a database (e.g., MySQL, SQLite, or PostgreSQL), you can use packages such as DBI and RMySQL, RPostgreSQL, or RSQLite for database-specific connections.

Example

```
# Establish a connection
con <- dbConnect(RSQLite::SQLite(), "path/to/your/database.sqlite")
# List tables
dbListTables(con)
# Read a table into a data frame
data <- dbReadTable(con, "your_table_name")
# Close the connection
dbDisconnect(con)
```

Reading Data from URLs

- ✓ R can also read data directly from a URL, which is useful when working with data hosted online (e.g., from a remote server or a file sharing service).

Example

```
data <- read.csv("https://example.com/data.csv")

data <- read.table("https://example.com/data.txt", sep = "\t", header = TRUE)
```

writing files

- ✓ You can write data to external files using various functions.

Writing Data to CSV Files

- ✓ The most common method for exporting data is to write it to a CSV file using the write.csv() function.
- ✓ **Syntax of write.csv():**

```
write.csv(x, file = "", row.names = TRUE)
```

Where

- ✓ **x**: The data frame or matrix to write to a file.
- ✓ **file**: The file path (can be a string or URL).
- ✓ **row.names**: Logical value (TRUE or FALSE) indicating whether to include row names. Default is TRUE.

Example

```
data <- data.frame(Name = c("John", "Jane", "Paul"), Age = c(23, 25, 22), Score = c(85, 90, 88))
write.csv(data, "output.csv", row.names = FALSE)
```

Writing Data to a Text File

- ✓ If you need to write data to a text file with custom delimiters (e.g., tab-separated or space-separated), you can use the **write.table()** function.
- ✓ **Syntax of write.table():**

```
write.table(x, file = "", row.names = TRUE, col.names = TRUE, sep = " ")
```

Where

- ✓ **x**: The data frame or matrix to write to a file.
- ✓ **file**: The file path (can be a string or URL).
- ✓ **row.names**: Logical value indicating whether to write row names (default is TRUE).
- ✓ **col.names**: Logical value indicating whether to write column names (default is TRUE).
- ✓ **sep**: The separator between values in the file (e.g., comma ,, tab \t, space).

Writing Data to a Text File

- ✓ If you need to write data to a text file with custom delimiters (e.g., tab-separated or space-separated), you can use the write.table() function.
- ✓ Syntax of write.table():

```
write.table(x, file = "", row.names = TRUE, col.names = TRUE, sep = " ")
```

- ✓ **x**: The data frame or matrix to write to a file.
- ✓ **file**: The file path (can be a string or URL).
- ✓ **row.names**: Logical value indicating whether to write row names (default is TRUE).
- ✓ **col.names**: Logical value indicating whether to write column names (default is TRUE).
- ✓ **sep**: The separator between values in the file (e.g., comma ,, tab \t, space).

Writing Data to Excel Files

- ✓ You can write data to Excel files using the write.xlsx() function from the openxlsx package or writexl package. Here, we'll show how to use openxlsx to write data to Excel.
- ✓ Install and Load the openxlsx Package:

```
write.xlsx(x, file, sheetName = "Sheet1", rowNames = FALSE, colNames = TRUE, ...)
```

- ✓ The data frame or matrix to write to the Excel file.
- ✓ **`file`**: The file path (e.g., `"output.xlsx"`).
- ✓ **`sheetName`**: The name of the sheet (default is `"Sheet1"`).
- ✓ **`rowNames`**: Logical value indicating whether to write row names (default is `FALSE`).
- ✓ **`colNames`**: Logical value indicating whether to write column names (default is `TRUE`).

Writing Data to a Database ()

- ✓ You can also write data to a database (e.g., SQLite) using the `DBI` and `RSQLite` packages.

Example

```
# Create a SQLite database connection
conn <- dbConnect(RSQLite::SQLite(), "database.sqlite")
# Write data to a table in the SQLite database
dbWriteTable(conn, "my_table", data, overwrite = TRUE)
# Close the database connection
dbDisconnect(conn)
```

Calling Functions

Scoping

- ✓ Scoping in R refers to the rules that determine how R searches for and evaluates variables in different environments (such as the global environment, function environments, etc.).
- ✓ It dictates how R identifies, variables and which value to assign to them when they are referenced in different contexts.

Environment

- ✓ In R, an **environment** is a collection of variables and their associated values.
- ✓ Environments are like separate compartments where data structures and functions are stored.
- ✓ Environments are fundamental to how R manages and searches for variables in the context of scoping rules.

Types of Environment

- ✓ Global Environment
- ✓ Package Environments and Namespaces

- ✓ Local Environments

Global Environment

- ✓ It is the space where all user-defined objects exist by default.
- ✓ When objects are created outside of any function, they are stored in global environment.
- ✓ Use: Objects in the global environment are accessible from anywhere within the session. Thus they are globally available.
- ✓ `ls()` lists objects in the current global environment.

```
Example:
R> v1 <- 9
R> v2 <- "victory"
R> ls()
 [1]
 "v1" "v2"
```

Local Environment •

- ✓ Local environment is created when a function is called.
- ✓ Objects defined within a function are typically stored in its local environment.
- ✓ When a function completes, its local environment is automatically removed.
- ✓ This allows identical argument-names in functions and the global workspace.
- ✓ Use: Local environments protect objects from accidental modification by other functions.

Example

```
Define a function with a local environment
my_function <- function()
{ local_var <- 42 return(local_var)
 }
```

Package Environment and Namespace

- ✓ It is the space where the package's functions and objects are stored.
- ✓ Packages have multiple environments, including namespaces.
- ✓ Namespaces define the visibility of package functions.
- ✓ Use: Package environments and namespaces allow you to use functions from different packages without conflicts. Syntax to list items in a package environment: `ls("package:package_name")`.
- ✓ Example: R> ls("package:graphics") #lists objects environment "abline" "arrows" "assocplot" "axis"

Search path()

- ✓ The search() function in R is used to display the **search path** of environments that R uses to look for objects (such as variables, functions, etc.). The search path is essentially the order of environments where R looks for objects when you reference them by name.

Environment()

- ✓ in R, the environment () function is used to get the environment associated with a function or an expression.
- ✓ It is often used to understand where an object is stored and can be particularly useful for debugging or when working with custom environments and functions.

Reserved and Protected Names Reserved Names

- ✓ These names are used for control structures, logical values, and basic operations.
- ✓ These names are predefined and have specific functionalities.
- ✓ These names are strictly prohibited from being used as object-names.
- ✓ Examples: if, else, for, while, function, TRUE, FALSE, NULL

Protected Names:

- ✓ These names are associated with built-in functions and objects.
- ✓ These names are predefined and have specific functionalities.
- ✓ These names should not be directly modified or reassigned by users.
- ✓ Examples: Functions like c(), data.frame(), mean() Objects like pi and letters.

Conditions and Loops

Conditions:

- ✓ In R, condition statements allow you to execute code based on whether certain conditions are true or false.

There are 5 types of decision statements:

1) if statement
2) if else statement
3) nested if statement
4) else if ladder (stacking if Statement)
5) switch statement

if statement:

- ✓ The if statement is the simplest decision-making statement which helps us to take decision on the basis of the condition.
- ✓ The block of code inside the if statement will be executed only when the Boolean expression evaluates to be true. If the statement evaluates false, then the code which is mentioned after the condition will run.

Syntax:

```
if (Boolean expression){
  // If the Boolean expression is true, then statement(s) will be executed.
}
```

Example:

```
x <-20
y<-24
if(x<y){
  paste (x,"is a smaller number\n")
}
```

If-else statement

- ✓ If the condition is true, the if block code is executed and if the condition is false, the else block code is executed.

Syntax:

```
if (Boolean expression) {
  // statement(s) will be executed if the Boolean expression is true.
} else {
  // statement(s) will be executed if the Boolean expression is false.
}
```

Example:

```
a<- 100
if(a<20){
  cat("a is less than 20\n")
}else {
  cat("a is not less than 20\n")
}
cat("The value of a is", a)
```

else if Ladder Statement (Stacking `if` Statements)

- ✓ The if-else ladder is made up of multiple if and else statements arranged in a ladder-like fashion. The conditional expressions are evaluated from top to bottom.
- ✓ If a condition is true, the associated statement is executed and the rest of the ladder is skipped. If all conditions are false, the else statement is executed.

Syntax:

```
if(expression1){
  statement1;
  } else if(expression2) {
statement2;
  } else if(expression3) {
statement3
  } else if(expression4) {
statement4
  } else {
default statement5
}
```

```
score <- 85
if (score >= 90) {
  grade <- "A"
} else if (score >= 80) {
  grade <- "B"
} else if (score >= 70) {
  grade <- "C"
} else {
  grade <- "F"
}
print (paste ("Your grade is:", grade))
```

nested if Statement

- ✓ An if-else statement within another if-else statement is called nested if statement.
- ✓ Hence, it is called as multi-way decision

Syntax:

```
if(expr1){
        if(expr2){
                statement1
        } else {
                 statement2
        }
} else{
        statement3
}
```

Example

```
x <- 8
if (x > 5) {
  print("x is greater than 5")
  if (x > 10) {
    print("x is also greater than 10")
  } else {
    print("x is not greater than 10")
  }
}
```

Using `ifelse` for Element-wise Checks

ifelse()

- ✓ performs conditional operations on each element of a vector
- ✓ returns corresponding values based on whether condition is TRUE or FALSE. This is particularly useful when you need to perform element-wise conditional operation on data structures.

Syntax:

ifelse(test, yes, no)

where

- ✓ test: A logical vector or expression that specifies the condition to be tested.

- ✓ yes: The value to be returned when the condition is TRUE.
- ✓ no: The value to be returned when the condition is FALSE.

Example:

```
grades <- c(85, 92, 78, 60, 75)
pass_fail <- ifelse(grades >= 70, "Pass", "Fail")
pass_fail
```

switch Statement

- ✓ The switch statement in R is a control flow statement that allows you to select one of several options based on the value of an expression

Syntax:

switch (expression, case1, case2, case3, ... default)

where

- ✓ expression: This is evaluated and determines which case to execute.
- ✓ case1, case2, ...: These are the possible outcomes. You can specify them as named arguments or as a list.
- ✓ default: (Optional) Code block when none of the cases match.

Example

```
day <- "Tuesday"

result <- switch (day,
            Monday = "Start of the week",
            Tuesday = "Second day of the week",
            Wednesday = "Midweek",
            Thursday = "Almost there",
            Friday = "Last working day",
            Saturday = "Weekend!",
            Sunday = "Rest day")

print(result)
```

Coding Loops:

- ✓ Loops in R allow you to execute a block of code repeatedly based on certain conditions or for each element in a collection.
- ✓ The most common types of loops in R: for, while, and repeat loops.

There are 3 types of loops:

- ✓ while loop
- ✓ for loop
- ✓ repeat

for Loop

- ✓`for` loop is useful when iterating over elements in a vector, lists or data-frames.

Syntax:

```
for (variable in sequence) {
# Code to be executed in each iteration
```

```
}
```

where

- ✓ variable: The loop-variable that takes on values from the sequence in each iteration.
- ✓ sequence: The sequence of values over which the loop iterates.

Example

```
# Print numbers from 1 to 5
for (i in 1:5) {
  print(i)
}
```

Nesting for Loops

- ✓ Nesting for loops involves placing one for loop inside another.

Example

```
# Nested for loop to create a multiplication table
for (i in 1:3) {
  for (j in 1:3) {
    print(i * j)
  }
}
```

while Loop

- ✓ A while loop statement can be used to execute a set of statements repeatedly as long as a given condition is true.
- ✓ It is also called entry controlled looping statement

Syntax:

```
while (condition) {
  # Code to execute
}
```

- ✓ Firstly, the expression is evaluated to true or false.
- ✓ If the expression is evaluated to false, the control comes out of the loop without executing the body of the loop.
- ✓ If the expression is evaluated to true, the body of the loop (i.e. statement1) is executed.
- ✓ After executing the body of the loop, control goes back to the beginning of the while statement.

Example

```
# Print numbers from 1 to 5 using a while loop
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1  # Increment i
}
```

Repeat Loop

- ✓ The repeat loop repeatedly executes a block of code until a break statement is encountered.

Syntax:

```
repeat {
  # Code to execute
  if (condition) {
    break  # Exit the loop
  }
}
```

Example

```
# Print numbers from 1 to 5 using a repeat loop
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break  # Exit the loop when i is greater than 5
  }
}
```

Nested Loops

- ✓ You can nest loops inside one another. Here's an example using nested for loops:

Example

```
# Nested for loop to create a multiplication table
for (i in 1:3) {
  for (j in 1:3) {
    print(i * j)
  }
}
```

Jumping Statements in R

- ✓ **Break**
- ✓ **Next**

break:

- ✓ Exits the loop immediately.

Example:

```
# Print numbers from 1 to 5 using a while loop
i <- 1
while (i <= 5) {
  if(i==3){
    break
  }
  print(i)
  i <- i + 1
}
```

next:

- ✓ **S**kips to the next iteration of the loop.

Example

```
# Print numbers from 1 to 5 using a while loop
i <- 1
while (i <= 5) {
  if(i==3){
```

```
  i <- i + 1
  next
 }
 print(i)
 i <- i + 1
}
```

Implicit Looping with apply ()

- ✓ The apply function is used for applying a function to subsets of a data structure,such as rows or columns of a matrix.
- ✓ It allows you to avoid writing explicit loops and can simplify your code.

Syntax: apply (X, MARGIN, FUN)

Where

- ✓ X: The data structure (matrix, data-frame, or array) to apply the function to.
- ✓ MARGIN: Specifies whether the function should be applied to rows (1) or columns (2) of the data structure
- ✓ FUN: The function to apply to each subset

Example

```
# Create a matrix
matrix_data <- matrix(1:9, nrow = 3)
# Apply the sum function to each row
row_sums <- apply(matrix_data, 1, sum)
print(row_sums)
# Apply the sum function to each column
col_sums <- apply(matrix_data, 2, sum)
print(col_sums)
```

WRITING FUNCTIONS

Function

- ✓ A function is a block of code to perform a specific task. function is defined using the **`function`** keyword. This can take one or more arguments.
- ✓ This can also return values using the `return` statement. Function helps encapsulate code
- ✓ improves code readability and allows to reuse code-segments.

Syntax:

```
function_name <- function(arg1, arg2, ...)
 {
            # Function body
            # Perform some operations using arg1, arg2, and other arguments
            # Optionally, return a result using 'return' statement
 return(value)
 }
```

Where

- ✓ **my_function**: The name of your function.

- ✓ **arg1, arg2, ...**: The arguments the function accepts.
- ✓ **return(value)**: This is optional; if omitted, the last evaluated expression will be returned.

Example

```
add_numbers <- function(x, y) {
  sum <- x + y
  return(sum)
}
# Using the function
result <- add_numbers(5, 3)
print(result)  # Output: 8
```

Passing arguments

- ✓ You can pass arguments to a function by position or by name.
- ✓ **Positional arguments:** Arguments are matched based on their order.
- ✓ **Named arguments:** Arguments are specified using their names.

Argument Matching In R

- ✓ Argument matching refers to the process by which function-arguments are matched to their corresponding parameter-names within a function call

Types of Argument Matching in R

- ✓ Exact matching
- ✓ Partial matching
- ✓ Positional matching
- ✓ Mixed matching
- ✓ Ellipsis (...) argument or **Dot as Dot argument**
- ✓ Default argument

Exact Matching:

- ✓ When you pass arguments by name (i.e., explicitly specifying the argument names), R will match the argument to the parameter that has the exact same name.

```
my_function <- function(a, b, c) {
  return (a + b + c)
}
my_function(a = 1, b = 2, c = 3)  # Exact match
```

Partial Matching:

- ✓ R allows partial matching of argument names. If you provide a prefix of a parameter name, and this prefix is unique, R will automatically complete the match for you.

```
my_function <- function(a, b, c=10) {
cat("a:", a, "b:", b, "c:", c, "\n")
```

```
}
function(a = 1, b = 2)
```

Positional Matching:

- ✓ If you do not specify argument names, R matches the arguments to parameters based on their position in the function call. The first argument goes to the first parameter, the second argument to the second parameter, and so on

```
my_function <- function(a, b) {
  return (a + b )
}
my_function(5, 10)  # a = 5, b = 10 based on position
```

Default Values:

- ✓ If you do not specify a value for an argument with a default, R uses the default value defined in the function.

```
my_function <- function(a, b = 5) {
  return(a + b)
}
my_function(10)  # b will default to 5, returns 15
```

Using Ellipsis (...) (Dot-dot-dot):

- ✓ This allows you to pass additional arguments to functions that can accept them without explicitly defining them.

```
my_function <- function(a, ...) {
  print(list(...))
}
my_function(1, b = 2, c = 3)  # prints b and c as a list
```

Mixing Named and Positional Arguments

- ✓ R allows you to mix positional and named arguments, but named arguments must come after all positional arguments in the function call.

Example

```
my_function <- function(a, b, c = 10) {
  cat("a:", a, "b:", b, "c:", c, "\n")
}
# Mixing positional and named arguments
my_function(1, b = 2)
```

Using return

- ✓ return is used to specify what value should be returned as the result of the function
- ✓ Functions can return values using the return () keyword. If no return value is specified, the function returns the last evaluated expression.

Example:

```
square <- function(x) {
  return(x * x)
}
result <- square(4)
```

Arguments Lazy Evaluation

- ✓ Lazy evaluation means expressions are evaluated only when needed.
- ✓ The evaluation of function-arguments is deferred until they are actually needed.
- ✓ The arguments are not evaluated immediately when a function is called but are evaluated when they are accessed within the function.
- ✓ This can help optimize performance and save computational resources.

Example:

```
lazy_example<- function(a, b)
{
  cat("Inside    the      function\n")
  cat("a =", a, "\n")
  cat("b =", b, "\n")
  cat("Performing some operations...\n")
  result <- a + b
  cat("Operations completed\n")
  return(result)
}
x<-10
y<-20
r<-lazy_example(x, y)
r
```

Defaults arguments

- ✓ You can provide predefined values for some or all of the arguments in a function.
- ✓ In R, default arguments allow you to specify a value for a function parameter that will be used if no value is provided when the function is called. This makes your functions more flexible and easier to use.

Syntax:

```
function_name <- function (arg1 = default_value1, arg2 = default_value2, ...)
  {
                # Function body
                # Use arg1, arg2, and other arguments
  }
```

Where

- ✓ **`arg1`, `arg2`, etc.:** These are the function-arguments for which you want to set default values.
- ✓ **`default_value1`, `default_value2`, etc.:** These are the values you assign defaults for the respective arguments.

Example:

```
# Define a function with default arguments
greet <- function (name = "Guest", greeting = "Hello") {
  message <- paste (greeting, name)
  return(message)
}
# Using the function with default arguments
print (greet ())              # Output: "Hello Guest"
print(greet("Alice"))         # Output: "Hello Alice"
print (greet ("Bob", "Welcome")) # Output: "Welcome Bob"
```

Checking for Missing Arguments

- ✓ **missing ()** is used to check if an argument was provided when calling a function.
- ✓ It returns 'TRUE` if the argument is missing (not provided) and `FALSE` if the argument is provided.

Syntax:

missing (argument name)

Where

- ✓ `argument_name`: This is the name of the argument you want to check

Example:

Function to check if an argument is missing

```
check_argument <- function(x) {
  if (missing(x)) {
    cat("The argument 'x' is missing.\n")
  } else {
    cat("The argument 'x' is provided with a value of", x, "\n")
  }
}
check_argument()
```

```
# Call the function without providing 'x'
check_argument()
# Call the function with 'x'
check_argument(42)
```

Output:

The argument 'x' is missing

The argument 'x' is provided with a value of 42

Dealing with Ellipses or Dot as Dot

- ✓ The ellipsis allows you to pass a variable number of arguments to a function, which can be particularly useful in a variety of scenarios\

Example

```
# Define a function that takes variable arguments
my_function <- function(...) {
```

```
 args <- list(...)  # Capture all arguments as a list
 return(args)
}
result <- my_function(1, 2, 3, "hello", TRUE)
print(result)
```

Specialized Functions Helper Functions

- ✓ These functions are designed to assist another function in performing computations
- ✓ They enhance the readability of complex functions.
- ✓ They can be either defined internally or externally.

Externally Defined Helper Functions

- ✓ These functions are defined in external libraries or modules.
- ✓ They can be used in your code w/o defining them within your program.
- ✓ They are typically provided by programming language or third-party libraries
- ✓ They provide commonly used functionalities.

Example: Externally defined helper function 'mean' is used to find mean of 5 nos.

```
values <- c(10, 20, 30, 40, 50)
average <- mean(values)
cat("The average is:", average, "\n")
```

Internally Defined Helper Functions

- ✓ These functions are also known as user-defined functions.
- ✓ They are defined by programmer according to their requirement.
- ✓ They are used to perform a specific task or operation.
- ✓ They enhance code organization, reusability, and readability.

Example: Internally defined helper function to calculate the square of a number

```
square <- function(x) { result  <-
x  *  x return(result)
  }
num <- 5
squared_num <- square(num)
cat("The square of", num, "is:", squared_num, "\n")
```

Disposable Functions

- These functions are created and used for a specific, one-time task.
- They are not intended for reuse or long-term use.
- They are often employed to perform a single, temporary operation.
- They are discarded after use.

Example: A disposable function to calculate the area of a rectangle once

```
calculate_rectangle_area <- function(length, width)
```

```
  {
area <- length * width
cat("The area of the rectangle is:", area, "\n")
  }
```

Advantages of Disposable Functions

- ✓ Convenient for simple, one-off tasks.
- ✓ Avoids cluttering the global environment with unnecessary function objects.
- ✓ Provides a concise way to define and use functions inline.

Recursive Functions

- ✓ These functions call themselves within their own definition.
- ✓ They solve problems by breaking them down into smaller, similar sub-problems.
- ✓ The recursive case defines how the problem is divided into smaller subproblems and solved recursively.

Example

```
factorial <- function(n) {
  if (n == 0) {
    return (1)  # Base case
  } else {
    return (n * factorial(n - 1))  # Recursive case
  }
}
# Using the function
print(factorial(5))  # Output: 120
```

Exception

- ✓ When there's an unexpected problem during execution of a function, R will notify you with either a warning or an error.
- ✓ In R, you can issue warnings with the warning command, and you can throw errors with the stop command

Example for warning command:

```
warn_test<-function(x){
 if(x<=0){
  warning ("'x' is less than or equal to 0 but setting it to 1 and continuing")
  x <- 1
}
  return(5/x)
}
warn_test(0)
```

Example for stop command:

```
error test<-function(x){
if(x<=0){



op ("'x' is less than or equal to 0... TERMINATE")
```

```
  }
return(5/x)
  }
error_test(0)
```

Exception handling

- ✓ Exception handling in R is a mechanism that allows you to manage errors and warnings in your code gracefully, ensuring that your program can continue running or respond appropriately when an error occurs.
- ✓ R provides several functions for exception handling, including try(), tryCatch(), and withCallingHandlers (). Here's a detailed overview of each:

Using try()

- ✓ The try () function allows you to attempt to execute an expression and capture any errors that occur. If an error happens, try() will return an object of class "try-error" instead of stopping execution.

Syntax: try (expr, silent = FALSE)

- ✓ expr: The expression you want to evaluate. This can be any R expression that might generate an error.
- ✓ silent: A logical value (TRUE or FALSE). If set to TRUE, any warnings generated during the execution will not be printed to the console. This is useful if you want to suppress warning messages.
- ✓ If the expression is evaluated successfully, try() returns the result of the expression.
- ✓ If an error occurs, it returns an object of class "try-error", allowing you to check whether the operation was successful.

Example

```
result <- try({
  log(-1)
}, silent = TRUE)

# Check if an error occurred
if (inherits(result, "try-error")) {
  print("An error occurred while calculating the logarithm.")
} else {
  print(result)
}
```

Using tryCatch ()

- ✓ In R, tryCatch() is a powerful function for handling exceptions that allows you to manage errors, warnings, and messages more flexibly than try().

Syntax

```
tryCatch(
  expr,
  error = function(e) { /* Handle error */ },
  warning = function(w) { /* Handle warning */ },
  message = function(m) { /* Handle message */ },
  finally = { /* Code to run regardless of success or failure */ }
)
```

- ✓ **expr**: The expression you want to evaluate. This is where you put the code that might throw an error or warning.
- ✓ **error**: A function that is called if an error occurs. The error object is passed to this function.
- ✓ **warning**: A function that is called if a warning occurs. The warning object is passed to this function.
- ✓ **message**: A function that is called if a message is signaled.

Example

```
result <- tryCatch (
  {
    log(-1)
  },
  error = function(e) {
    paste("Caught an error: ")
  },
  warning = function(w) {
    paste ("Caught a warning:")
  }
)
print(result)
```

Finally ()

- ✓ function in R is used to specify a block of code that will be executed regardless of whether an error occurs or not.
- ✓ This is particularly useful for tasks that need to be performed regardless of the outcome of a `tryCatch()` block.

Example

```
result <- tryCatch (
  {
    log(-1)
  },
  error = function(e) {
    paste("Caught an error: ")
  },
  warning = function(w) {
    paste ("Caught a warning:")
  },
Finally={
Print("Executed")
}
)
print(result)
```

withCallingHandlers ().

- ✓ The withCallingHandlers () function in R is used for handling conditions (such as errors, warnings, and messages) that arise during the evaluation of an expression. Unlike tryCatch (), which stops the evaluation when a condition is encountered, withCallingHandlers () allows the evaluation to continue while providing a mechanism to respond to conditions as they are signaled.

Syntax

```
withCallingHandlers(
  expr,
  error = function(e) { /* Handle error */ },
  warning = function(w) { /* Handle warning */ },
  message = function(m) { /* Handle message */ }
)
```

Example

```
result <- withCallingHandlers (
  {
    log(-1)
  },
  error = function(e) {
    paste ("Caught an error: ")
  },
  warning = function(w) {
    paste ("Caught a warning:")
  }
)
print(result)
```

Timing

- ✓ In R, **timing** refers to measuring the execution time of code to understand its performance and efficiency. R provides several functions to measure how long a specific piece of code takes to run, helping you to optimize or debug your code.
- ✓ R provides several built-in functions and packages to measure the execution time of your code. Here are some common methods:
- ✓ The Sys.sleep() command makes R pause for a specified amount of time, in seconds, before continuing.

Using system.time()

- ✓ The system.time() function measures the amount of time taken to evaluate an expression.

```
Starttime ← Sys.time(){
Func ()
  }
Endtime ← Sys.time()
```

Example

```
Sleep_func ← function(){
Sys.sleep(5)
  }
Starttime ← Sys.time(){
Sleep_func()
  }
Endtime←Sys.time()
Print (Endtime-Starttime)
```

Visibility

- ✓ The location where we can find a variable and also access it if required is called the scope of a variable.
- ✓ There are mainly two types of variable scopes:

Global Variables: As the name suggests, Global Variables can be accessed from any part of the program.

- ✓ They are available throughout the lifetime of a program.
- ✓ They are declared anywhere in the program outside all of the functions or blocks.
- ✓ Declaring global variables: Global variables are usually declared outside of all of the functions and blocks. They can be accessed from any portion of the program.

Example:

```
#globalvariable
global = 5
# global variable accessed from within a function
display = function()
{
  print(global)
}
display()
# changing value of global variable
global = 10
display()
```

Local Variables:

- ✓ Variables defined within a function or block are said to be local to those functions.
- ✓ Local variables do not exist outside the block in which they are declared, i.e. they cannot be accessed or used outside that block.
- ✓ Declaring local variables: Local variables are declared inside a block.

```
Func<-function(){
  # this variable is local to the
  # function func() and cannot be # accessed outside this function
  age = 18
  print(age)
}
cat("Age         is:\n")
func()
```