Module-02

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.

2 MARKS QUESTIONS.

1.What is standalone statement with example.

2.What are all the coding loops using in R

3.What is function with syntax.

4.What is global and local variable.

5.Explain types of reading files.

3MARKS QUESTIONS.

1.Explain difference between Conditional statements and coding loops in R

2.Explain coercion with example program.

3.Write a R program how to take the input from the user using readline() function includes NAME, AGE,CLASS,SECTION,COURSE,UUCMS NUMBER

4.What is Function. Explain function declaration and function call in program.

5.what is class and Explain # types of class.

5MARKS QUESTIONS.

1.Explain the condtional statements with example program.

2.What is recursive function? To write a R program take the input from the user To find factorial of given number using recursive function.

3.Expalin Control flow mechanism 1)Break2)next,3)repeate statements with example

4.What is exception. To implement R program using function with try, catch, finally exception handling.

5.Expalin Time and Visibility/progressbar with example.

10MARKS QUESTIONS.

1.What is function .Explain types of functions. To build one R program Largest of three numbers using function.

2.Explain Conditional statements in detail with example program.

3.What are all the Coding loops using in R with example program.

4.Explain Exception Handling with syntax and example program.

5.write a R Program that includes different operators, control structures, default values for arguments, returning complex objects.

UNIT-II

Reading and Writing files

R-Ready Data-Sets

• R provides built-in data-sets.

• Data-sets are also present in user-contributed-packages.

• The data-sets are useful for learning, practice and experimentation.

• 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

i) alphabetically by name and ii) 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") **# to install the package**

R> library("tseries") **# to load the package**

R> library(help="tseries") **# to explore the data-sets in " tseries " package**
R> help("ice.river") **# to get info about the "ice.river" data-set** R> data(ice.river) **# To access the data in your workspace**

R> ice.river[1:5,] **#display the first five records**

Reading in External Data Files

You can read data from external files using various functions.

Table Format

• Table-format files are plain-text files with three features:

1) Header: If present, the header is the first line of the file.
The header provides column names.

2) Delimiter: The delimiter is a character used to separate entries in each line.
3) Missing Value: A unique character string denoting missing values This is converted to `NA` when reading. • Table-format files typically have extensions like `.txt` or `.csv`.

Reading Table-Format Files

• **read.table()** is used for

i) reading data from a table-format file (typically plain text) and ii) creating a data frame from it.

• This function is commonly used for importing data into R for further analysis.

Syntax: read.table(file, header = FALSE, sep = "")
where, **file:** The name of the file from which data should be read. This can be a local file path or a URL.

header: This indicates whether the first row of the file contains column names Default is FALSE.

sep: This represents the field separator character.
Default is an empty string "".

Example: Suppose you have a file named data.txt with the following content:

Name Age City

Krishna 26 Mysore

Arjuna 31 Mandya

Karna 29 Maddur

• To read this data into R and create a data frame from it: #
Specify the file path
file_path <- "data.txt"

# Use read.table to read the data into a data frame my_data
<- read.table(file_path, header = TRUE, sep = "\t")

# View the resulting data frame print(my_data)

Output:

```
Name     Age    City
Krishna  26    Mysore
Arjuna    31    Mandya
Karna      29   Maddur
```

Explanation of above program:

• file_path is set to the location of the data.txt file.

• We use read.table to read the data from the file into the my_data data frame.

• The header argument is set to TRUE because the first row of the file contains column names. • The sep argument is set to "\t" because the file uses tab as the field separator.

Web-Based Files

• **read.table()** can be used for reading tabular data from web-based files.

• We can import data directly from the internet.

Example: To read tabular data from a web-based file located at the following URL: https://example.com/data.txt.

# Specify the URL of the web-based file

url <- "https://example.com/data.txt"

# Use read.table to read the data from the web-based file my_data <- read.table(url, header = FALSE, sep = "\t")

# View the resulting data frame
print(my_data)

Explanation of above program:

• url is set to the URL of the web-based file.

• We use read.table to read the data from the specified URL into the my_data data frame.

• We set header to FALSE since the data file doesn't have a header row.

• We specify sep as "\t" because the data is tab-separated. If the data were commaseparated, you would use sep = ",".

Spreadsheet Workbooks

• R often deals with spreadsheet software file formats, such as Microsoft Office Excel's `.xls` or `.xlsx`.

• Exporting spreadsheet files to a table format, like CSV, is generally preferable before working with R.

Reading CSV Files

• **read.csv()** is used for reading comma-separated values (CSV) files.

• It simplifies the process of importing data stored in CSV format into R.

Syntax: read.csv(file, header = TRUE, sep = ",",)
where **file:** The name of the file from which data should be read. This can be a local file path or a URL.

header: This indicates whether the first row of the file contains column names Default is FALSE.

sep: This represents the field separator character.
Default is an empty string "".
Example: Suppose you have a CSV file named data.csv with the following content:

```
Name    Age   City
Krishna  26   Mysore
Arjuna    31   Mandya
Karna     29   Maddur
```

• To read this data into R and create a data frame from it: #
Specify the file path
file_path <- "data.csv"

# Use read.csv to read the CSV data into a data frame my_data
<- read.csv(file_path)

# View the resulting data frame
print(my_data) **Output:**

```
Name    Age   City
Krishna  26   Mysore
Arjuna    31   Mandya
Karna     29   Maddur
```

Explanation of above program:

• file_path is set to the location of the data.csv file.
• We use read.csv to read the data from the CSV file into the my_data data frame. • Since the CSV file has a header row with column names, we don't need to specify the header argument explicitly; it defaults to TRUE.
• The default sep argument is ",", which is suitable for CSV files.

Writing Out Data Files and Plots

• You can write data to external files using various functions.

Writing Files

• **write.table()** is used to write a data frame to a text file.
Syntax: write.table(x, file, sep = " ",row.names = TRUE, col.names = TRUE, quote

= TRUE) where **x:** The data frame or matrix to be written to the file. **file:** The name of the file where the data should be saved.

sep: This represents the field separator character (e.g., "\t" for tab-separated values, "," for comma-separated values).

row.names: A logical value indicating whether row names should be written to the file. Default is TRUE.

col.names: A logical value indicating whether column names should be written to the file. Default is TRUE

quote: A logical value indicating whether character and factor fields should be enclosed in quotes. Default is TRUE.

Example: Suppose you have a data frame my_data that you want to write to a text file named "my_data.txt".

```
# Sample data frame my_data
<- data.frame(
Name = c("Arjuna", "Bhima", "Krishna"),
Age = c(25, 30, 22),
Score = c(85, 92, 78)
)
# Specify the file name
file_name <- "my_data.txt"

# Use write.table to save the data frame to a tab-separated text file
write.table(my_data, file = file_name, sep = "\t", row.names = FALSE,
col.names
= TRUE, quote = TRUE)

# Confirmation message
cat(paste("Data saved to", file_name))
```

Explanation of above program:

- We have a sample data frame called my_data with columns "Name," "Age," and

"Score."

- We specify the file_name as "my_data.txt" to define the name of the output file.
- We use the write.table function to write the data frame to the specified file.
- We set sep to "\t" to indicate that the values should be tab-separated.
- We use row.names = FALSE to exclude row names from the output.
- We set col.names = TRUE to include column names in the output.
- We set quote = TRUE to enclose character and factor fields in quotes for proper formatting.

Plots and Graphics Files

Using `jpeg` Function

- **jpeg()** is used to create and save plots as JPEG image files.
- You can specify parameters like the filename, width, height, and quality of the JPEG image. **Syntax: jpeg(filename, width, height)** where

file: The name of the JPEG image file to which the graphics will be written
width and height: The width and height of the JPEG image in pixels.

Example:

# Create sample data

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

y <- c(1, 4, 9, 16, 25)

# Open a JPEG graphics device and save the plot to a file jpeg(filename = "scatter_plot.jpg", width = 800, height = 600)

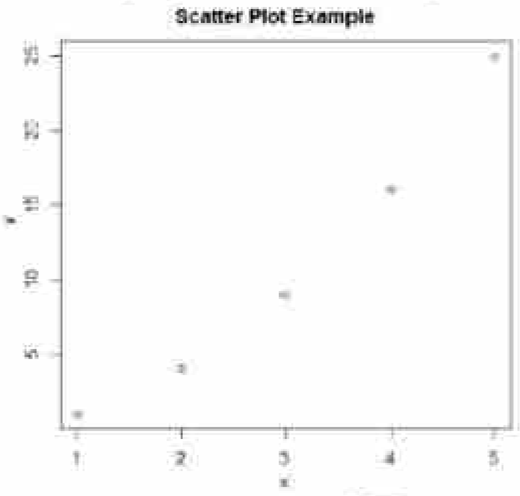
# Replot the same graph (this time it will be saved as a JPEG) plot(x, y, type = "p", main = "Scatter Plot Example")

dev.off()   **# Close the JPEG graphics device**

Explanation of above program:

- We create a simple scatter plot of x and y data points.
- We use the jpeg() function to specify the output as a JPEG image with the filename "scatter_plot.jpg."

- We set the dimensions of the output image using the width and height parameters (800x600 pixels).
- The quality parameter is set to 90, which controls the image compression quality (higher values result in better quality but larger file sizes).
- After opening the graphics device, we replot the same graph, which is now directed to the JPEG file.
- Finally, we close the JPEG graphics device using dev.off().



Using `pdf` Function

- pdf() can be used to create and save plots as PDF files.

Syntax: pdf(file, width, height) where **file:** The name of the PDF file to which the graphics will be written. **width and height:** The width and height of the PDF page in inches.

Example:

# Create a simple scatter plot x
<- c(1, 2, 3, 4, 5)
y <- c(1, 4, 9, 16, 25)

# Open a PDF file for plotting
pdf("scatter_plot.pdf", width = 6, height = 4)

# Create a scatter plot
plot(x, y, type = "p", main = "Scatter Plot Example", xlab = "X-axis", ylab = "Yaxis")

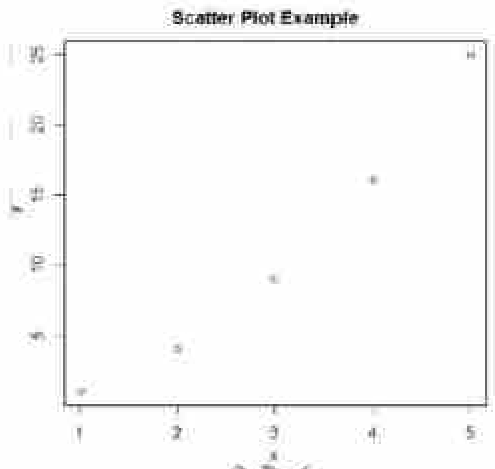
# Close the PDF file
dev.off()

Explanation of above program:

- We open a PDF graphics device using pdf() and specify the name of the output PDF

file, as well as the dimensions (width and height) of the PDF page.

- We create a scatter plot using the plot() function.
- The graphical output is written to the "scatter_plot.pdf" file in PDF format.
- We close the PDF device using dev.off() to complete the PDF file.



Ad Hoc Object Read/Write Operations

- Most common input/output operations involve data-sets and plot images.
- For handling objects like lists or arrays, you can use the `dput` and `dget` commands.

Using `dput` to Write Objects

- dput() is used to write objects into a plain-text file.
- It's often used to save complex objects like lists, data frames, or custom objects in a human-readable format.

Syntax: dput(object, file = "") where **object:** The R object you want to serialize into R code. **file:** The name of the file where the data should be saved.

Using `dget` to Read Objects

- dget() is used to read objects stored in a plain-text file created with `dput`.
  Syntax: dget(file) where **file:** The name of the file from which data should be read.

Example: Program to illustrate usage of `dput` and `dget`

# Create a sample list
my_list <- list(name =
"Rama", age = 30,
city = "Mysore",
)

# Use dput to serialize the list and save it to a text file dput(my_list,
file = "my_list.txt")

# Use dget to read and recreate the R object from the text file recreated_list
<- dget(file = "my_list.txt")

# Print the recreated R object print(recreated_list)

Explanation of above program:

- We start by creating a sample list named my_list. This list contains various elements, including a name, age, city, a vector of hobbies, and a Boolean value indicating whether the person is a student.
- We then use the dput() function to write the my_list object to a plain-text file.
- The first argument to dput() is the object my_list.
- The second argument, file, specifies the name of the file my_list.txt where
- We use the dget() function to read the object from the specified file.
- The file argument in dget() specifies the name of the file from which to read the object (my_list.txt in this case).

Calling Functions

Scoping

- Scoping-rules determine how the language accesses objects within a session. • These rules also dictate when duplicate object-names can coexist.

Environments

- Environments are like separate compartments where data structures and functions are stored.
- They help distinguish identical names associated with different scopes.

- Environments are dynamic and can be created, manipulated, or removed.
- Three important types of environments are:

1) Global Environment
2) Package Environments and Namespaces
3) 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.
- These environments are isolated from the Global Environment.
- This allows identical argument-names in functions and the global workspace.
- **Use:** Local environments protect objects from accidental modification by other functions.

```
# 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 contained in graphics package
environment
"abline" "arrows" "assocplot" "axis"
```

Search-path

- A search-path is used to access data structures and functions from different environments.
- The search-path is a list of environments available in the session.
- **search()** is used to view the search-path.
- Example:

```
R> search()
".GlobalEnv" "package:stats" "package:graphics" "package:base"
```

- The search-path
  i) starts at the global environment (.GlobalEnv) and ii) ends with the base package environment (package:base).
- When looking for an object, R searches environments in the specified order.
- If the object isn't found in one environment, R proceeds to the next in the searchpath.
- environment() can be used to determine function's environment.

```
R> environment(seq)
<environment: namespace:base>
R> environment(arrows)
<environment: namespace:graphics>
```

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.

<u>Argument Matching</u>

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

• Five ways to match function arguments are

1) Exact matching

2) Partial matching

3) Positional matching

4) Mixed matching 5) Ellipsis (...) argument

Exact

• Exact matching is the default argument matching method.

• In this, arguments are matched based on their exact parameter-names.

• Advantages:

1) Less prone to mis-specification of arguments. 2) The order of arguments doesn't matter.

• Disadvantages:

1) Can be cumbersome for simple operations.

2) Requires users to remember or look up full, case-sensitive tags.

• Example:

```
R> mat <- matrix(data=1:4, nrow=2, ncol=2, dimnames=list(c("A","B"),
c("C","D")))
```

```
R> mat
C D
A 1 3
B 2 4
```

Partial Matching

- Partial matching allows to specify only a part of the parameter-name as argument.
- The argument is matched to the parameter whose name starts with the provided partial name.
- Example:

```
R> mat <- matrix(nr=2, di=list(c("A","B"), c("C","D")), nc=2, dat=1:4)
R> mat
C D
A 1 3
B 2 4
```

- Advantages:
  1) Requires less code compared to exact matching.
  2) Argument tags are still visible, reducing the chance of mis-specification.
- Disadvantages:
  1) Can become tricky when multiple arguments share the same starting letters in their tags.
  2) Each tag must be uniquely identifiable, which can be challenging to remember.

Positional Matching

- Positional matching occurs when you specify arguments in the order in which the parameters are defined in the function's definition.
- Arguments are matched to parameters based on their position.
- **args()** can be used to find the order of arguments in the function.
- Example: R> args(matrix)

```
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
NULL
```

```
R> mat <- matrix(1:4, 2, 2, F, list(c("A","B"), c("C","D")))
R> mat
C D
A 1 3
B 2 4
```

• Advantages:

1) Results in shorter, cleaner code for routine tasks. 2) No need to remember specific argument tags.

• Disadvantages:

1) Requires users to know and match the defined order of arguments.
2) Reading code from others can be challenging, especially for unfamiliar functions.

Mixed Matching

• Mixed matching allows a combination of exact, partial, and positional matching in a single function call.

• Example:

```
R> mat <- matrix(1:4, 2, 2, dim=list(c("A","B"),c("C","D")))
R> mat
C D
A 1 3
B 2 4
```

Dot-Dot-Dot: Use of Ellipses

• Ellipsis argument allows you to pass a variable number of arguments to a function. • Example:

Functions like `c()`, `data.frame()`, and `list()`

• Example:

```
R> args(list)
function (...)
NULL
```

Conditions and Loops

Conditional constructs allow programs to respond differently depending on whether a condition is 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

Stand Alone Statements (if Statement)

The if statement is the simplest decision-making statement which helps us to take a 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)
{
  print(x,"is a smaller number\n")
}
```

Output: 20 is a smaller number

Stand Alone Statements (if Statement) with illustration in exercise 10.1 (Already solved in class and refer in class notes)

If-else statement

There is another type of decision-making statement known as the if-else statement. An if-else statement is the if statement followed by an else statement. An if-else statement, else statement will be executed when the boolean expression will false. **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)
```

Output: a is not less than 20

```
a is not less than 20
The value of a is 100
```

else if Statement (Stacking `if` Statements)

This is basically a "multi-way" decision statement. This is used when we must choose among many alternatives.

The expressions are evaluated in order (i.e. top to bottom). If an expression is evaluated to true, then

→ statement associated with the expression is executed &

→ control comes out of the entire else if ladder **Syntax:**

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

Example:

```
        marks=83;  if(marks>75){
        print("First class")  }else
        if(marks>65){
        print("Second class")
        }else if(marks>55){
        print("Third class")
        }else{
          print("Fail")
        }
```

Output: First class

nested if Statement

An if-else statement within another if-else statement is called nested if statement. This is used when an action has to be performed based on many decisions. Hence, it is called as multi-way decision **Syntax:** if(expr1) { if(expr2) statement1 else statement2

```
} else {
if(expr3)
statement3
else
statement4
}
```

Here, firstly expr1 is evaluated to true or false.

- If the expr1 is evaluated to true, then expr2 is evaluated to true or false.
  - If the expr2 is evaluated to true, then statement1 is executed.
    - If the expr2 is evaluated to false, then statement2 is executed.
- If the expr1 is evaluated to false, then expr3 is evaluated to true or false.
  - If the expr3 is evaluated to true, then statement3 is executed.
    - If the expr3 is evaluated to false, then statement4 is executed.

Example:

```
a <- 7
b <- 8
c <- 6 if (a > b) { if (a
  > c) { cat("largest =
  ", a, "\n")
} else { cat("largest
=", c, "\n")
}
} else { if
(b > c) {
cat("largest =", b, "\n")
} else { cat("largest
=", c, "\n")
}
}
```

Output:

```
Largest Value is: 8
```

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 operations 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:

# Create a numeric vector

grades <- c(85, 92, 78, 60, 75)

# Use ifelse to categorize grades as "Pass" or "Fail" pass_fail <- ifelse(grades >= 70, "Pass", "Fail")

# Display the result

pass_fail **Output:**

"Pass" "Pass" "Pass" "Fail" "Pass"

Explanation of the program

We have a vector grades containing numeric values representing exam scores. We use ifelse() to categorize each score as "Pass" if it's greater than or equal to 70, or "Fail" if it's less than 70.

The resulting pass_fail vector contains the categorization based on the condition.

switch Statement

This is basically a "multi-way" decision statement.

This is used when we must choose among many alternatives.

Syntax:

switch(expression,
case1, result1, case2,
result2,

. . . , . . . default) **where expression:** The expression whose value you want to match against the cases. **case1, case2**, ...: Values to compare against the expression.

result1, result2, ...: Code blocks when the expression matches the corresponding case. **default:** (Optional) Code block when none of the cases match. **Example:** grade <- "B"

# Check the grade and provide feedback switch(grade,

```
"A" = cat("Excellent!\n"),
"B" = cat("Well done\n"),
"C" = cat("You passed\n"),
"D" = cat("Better try again\n"),
cat("Invalid grade\n")
)
```

Output:

Well done

Coding Loops

• Loops are used to execute one or more statements repeatedly.

• There are types of loops:

1) while loop
2) for loop
3) Repeated loop

for Loop

• `for` loop is useful when iterating over elements in a vectors, lists or dataframes.

• **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:

```
numbers <- c(1, 2, 3, 4, 5) for
(i in numbers) { print(2*i)
}
```

Output: 2
4 6 8 10

Nesting for Loops

Nesting for loops involves placing one for loop inside another.

This allows you to create complex iteration patterns where you iterate over elements of multiple data structures.

Example:

```
for (i in 1:3)
{
for (j in 1:3)
```

```
{
product <- i * j cat(product,
"\t")
} cat("\n")
```

} **Output:**

```
1 2 3
2 4 6
3 6 9
```

Explanation of above program:

The outer loop iterates through i from 1 to 3.

For each value of i, the inner loop iterates through j from 1 to 3.
Within the inner loop, it calculates the product of i and j, which is i * j, and prints it to the console followed by a tab character ("\t").

After each row of products is printed (after the inner loop), a newline character ("\n") is printed to move to the next row.

while Loop

• A while loop statement can be used to execute a set of statements repeatedly as long as a given condition is true.

• **Syntax:** while(expression)

```
{
statement1;
```

} • 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:

```
i <- 1 # Initialize a variable
while (i <= 3) {
```

```
cat("Welcome to R \n") i
<- i + 1
}
```

Output:

```
Welcome to R
Welcome to R
Welcome to R
```

Example: Program to Create Identity Matrices #
Specify the size of the identity matrix (e.g., n = 4) n

```
<- 4 row <- 1
identity_matrix  <- matrix(0,  nrow = n,  ncol = n)
```

# Use a while loop to populate the matrix

```
while  (row <= n) {  identity_matrix[row,  row]
<- 1 row <- row + 1
}
```

# Print the identity matrix

print(identity_matrix) **Output:**

```
        [,1] [,2] [,3] [,4]
[1,]  1     0    0    0
[2,]      0     1    0    0
[3,]      0     0    1    0
[4,]      0     0    0    1
```

Explanation of above program:

- We specify the size of the identity matrix (in this case, n = 4).
- We initialize a variable row to 1 and create an empty matrix identity_matrix of size n x n filled with zeros.
- We use a while loop to populate the identity_matrix by setting the diagonal elements to 1. The loop runs until row exceeds the value of n.
- Finally, we print elements of the identity matrix to the console.

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: Program to illustrate usage of apply function

```
# Create a sample matrix of exam scores scores_matrix <- matrix(c(60, 70,
80, 90, 60, 70, 80, 90, 60, 70, 80, 90), nrow = 4 )
# Use apply to calculate the mean score for each student (across exams)
mean_scores <- apply(scores_matrix, 1, mean)
# Print the mean scores cat("Mean
Scores:")
print(mean_scores)
```

Output:

```
Mean Scores: 60 70 80 90
```

Explanation of above program:

• scores_matrix is a 4x3 matrix representing exam scores for four students (rows) across three exams (columns).

• We use the apply function with MARGIN = 1 to apply the mean function to each row
(i.e., across exams) of the scores_matrix.

• The result is stored in the mean_scores vector, which contains the mean score for each student.

WRITING FUNCTIONS

Function Creation

• 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
}
```

Where

`function_name`: This is the name of the function.

`arg1, arg2, ...`: These are the function-arguments.

`{ ... }` **:** This is the body of the function, enclosed in curly braces `{}`. - **`return(...)`:** Optionally, you can use the `return` statement to return values

Example:

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

result <- square(5)                # Call the function
cat("The square of 5 is:", result)
```

Output:

```
The square of 5 is: 25
```

Explanation of above program:

• We define a function called `square` that takes one argument `x`.

• Inside the function, we calculate the square of `x` by multiplying it by itself and store the result in the `result` variable.

• We use the `return` statement to specify that the result should be returned when the function is called.

- We call the `square` function with the argument `5` and store the result in the `result` variable.
- Finally, we print the result, which is "The square of 5 is: 25".

Function to print the Fibonacci sequence up to 150.

```
fibo1 <- function() {
fib_a <- 1 fib_b <- 1
cat(fib_a, ", ", fib_b, ", ", sep = "")
while (fib_b <= 150) { temp <-
fib_a + fib_b fib_a <- fib_b fib_b
<- temp if (fib_b <= 150) {
cat(fib_b, ", ", sep = "")
}
}
}

fibo1()    # Call the function to print the Fibonacci sequence up to 150
```

Output:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

Explanation of above program:

- We initialize fib_a and fib_b to 1 and print the first two Fibonacci numbers.
- We use a while loop to continue generating and printing Fibonacci numbers as long as fib_b is less than or equal to 150.
- Inside the loop, we calculate the next Fibonacci number, update fib_a and fib_b accordingly, and print the Fibonacci number if it's still less than or equal to 150. • When you call fibo1(), it will print the Fibonacci sequence up to 150.

Passing arguments

Example: Function to print area of circle

```
Circ.area function(r)
{
Area  pi*r^2
```

Return (area)
}

Circ.area(5)
Output: 78.539

Explanation of above program:

- We are passing radius as an argument to function
- Inside the function calculating the area of circle, then returning the value back where the function has been called.

Using return

- return is used to specify what value should be returned as the result of the function
- This allows you to pass a value or an object back to the calling code.
- If there's no `return` statement inside a function:
  i) The function ends when the last line in the body code is executed.
  ii) It returns the most recently assigned or created object in the function. iii) If nothing is created, the function returns `NULL`.

Example: add_numbers <-
function(x, y)
{
result <- x + y return(result)
}
# Call the function and store the result in a variable sum_result
<- add_numbers(5, 3)
# Print the result cat("The sum
is:", sum_result, "\n") Output:
The sum is:8

Explanation of above program:

- We define a function called add_numbers that takes two arguments x and y.
- Inside the function, we calculate the sum of x and y and store it in the variable result.

- We use the return statement to specify that the result should be returned as the output of the function.
- When we call add_numbers(5, 3), it calculates the sum of 5 and 3 and returns the result, which is 8.
- We store the returned result in the variable sum_result and then print it.

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)
}
# Create two variables
x <- 10 y <- 20
# Call the function with the variables
lazy_example(x, y)
```

Output:

```
Inside the function
a = 10 b = 20
Performing some operations... Operations
completed
[1] 30
```

Explanation of above program:

- When we call lazy_example(x, y), we are passing the variables x and y as arguments to the function.
- Initially, the function prints "Inside the function" and then proceeds to print "a =

10" and "b = 20". This indicates that the values of a and b are evaluated at this point.

- Next, the function prints "Performing some operations..." and calculates result as a

+ b.

- However, it does not evaluate the actual values of x and y (i.e., 10 and 20) immediately. Instead, it holds off on the computation until the result is needed.
- Finally, the function prints "Operations completed" and returns the result, which

is

30.

Setting Defaults

- You can provide predefined values for some or all of the arguments in a function.
- Useful for providing a default behavior if user doesn't specify a value for arguments.
- **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 as defaults for the respective arguments.

- **Example:** Function to calculate the area of a rectangle
  calculate_rectangle_area <- function(width = 2, height = 3) { area <- width * height return(area)

}
# Call the function without specifying width and height
default_area <- calculate_rectangle_area() **# Call the function with custom width and height** custom_area <- calculate_rectangle_area(width = 5, height = 4) cat("Default Area:", default_area, "\n")
cat("Custom Area:", custom_area, "\n")

Output:

```
Default Area:6
Custom Area:20
```

Explanation of above program:

- We define a function `calculate_rectangle_area` that takes two arguments, `width` and `height`, with default values of 2 and 3, respectively.
- Inside the function, we calculate the area of a rectangle using the provided or default values.
- When we call `calculate_rectangle_area()` without specifying `width` and `height`, the function uses the default values (2 and 3) and returns the default area of 6, which is stored in the variable `default_area`.
- When we call `calculate_rectangle_area(width = 5, height = 4)` with custom values, the function uses these values and returns the calculated area of 20, which

is stored in the variable
`custom_area`.

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") }
}
```

# 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
```

Explanation of above program:

- We define a function called `check_argument` that takes one argument, `x`.
- Inside the function, we use the `missing` function to check if the argument `x` is missing (not provided). If it is missing, we print a message indicating that it is missing. Otherwise, we print the value of `x`.
- When we call `check_argument()` without providing `x`, the function uses `missing` to check if `x` is missing and prints "The argument 'x' is missing."
- When we call `check_argument(42)` with a value of 42 for `x`, the function uses `missing` to check that `x` is provided with a value and prints "The argument 'x' is provided with a value of 42."

Dealing with Ellipses

- Ellipsis allows passing extra arguments w/o predefining them in the argumentlist.
- Typically, ellipses are placed in the last position of a function-definition.
- Example: Function to generate and plot Fibonacci sequence up to 150.

```
myfibplot <- function(thresh, plotit = TRUE, ...) {
fibseq <- c(1, 1) counter <- 2
while (fibseq[counter] <= thresh) {
```

```
fibseq <- c(fibseq, fibseq[counter - 1] + fibseq[counter])
counter <- counter + 1 if (fibseq[counter] > thresh) {
break }
} if (plotit)
{
plot(1:length(fibseq), fibseq, ...)
} else { return(fibseq)
}
}
```

Explanation of above program:

- We use a while loop to generate Fibonacci numbers and add them to the fibseq vector until the condition fibseq[counter] <= thresh is no longer met.
- We check the condition fibseq[counter] > thresh within the loop, and if it's true, we break out of the loop.

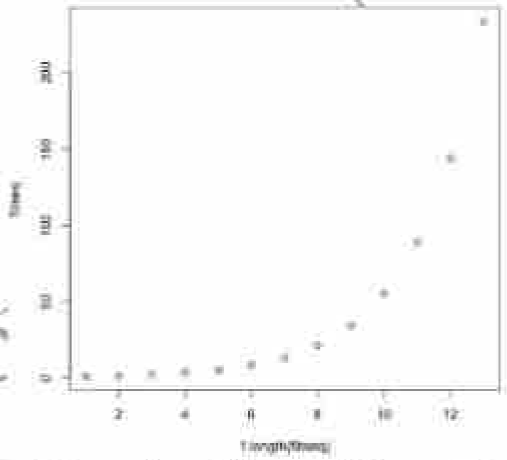


Figure 11-1: The default plot produced by a call to myfibplot, with thresh=150

<u>**Specialized Functions**</u>

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")

Output:

The average is: 30

Explanation of above program:

- We use the `mean` function, which is an externally defined helper function provided by the R programming language.
- mean() calculates the average of the numeric values in the `values` vector.
- We store the result in the `average` variable and then print it.

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")

Output:

The square of 5 is: 25

Explanation of above program:

- We define an internally defined helper function called `square`. This function calculates the square of a number `x`.
- Inside the function, we perform the calculation and store the result in the `result` variable.

• We use the `return` statement to specify that the `result` should be returned as the output of the function.

• We then call the `square` function with a value of `5` and store the result in the `squared_num` variable. • Finally, we print the squared value using the `cat` function.

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") }
```

# Use the disposable function to calculate the area of a specific rectangle

calculate_rectangle_area(5, 3) **Output:**

The area of the rectangle is: 15

Explanation of above program:

• We define a function called `calculate_rectangle_area` that calculates the area of a rectangle based on its length and width.

• We use this function once to calculate the area of a specific rectangle with a length of 5 units and a width of 3 units.

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 subproblems.

• They consist of two parts: a base case and a recursive case.

• The base case defines the condition under which the recursion stops.
The recursive case defines how the problem is divided into smaller subproblems and solved recursively.

• Example: Recursive function to calculate the nth Fibonacci number myfibrec <- function(n) { if (n == 1 || n == 2) {

```
return(1) # Base cases: Fibonacci of 1st and 2nd numbers are both 1
} else {
return(myfibrec(n - 1) + myfibrec(n - 2)) # Recursive case }
}
# Example: Calculate the 5th Fibonacci number
n <- 5 result <- myfibrec(n)
cat("The", n, "th Fibonacci number is:", result, "\n")
```

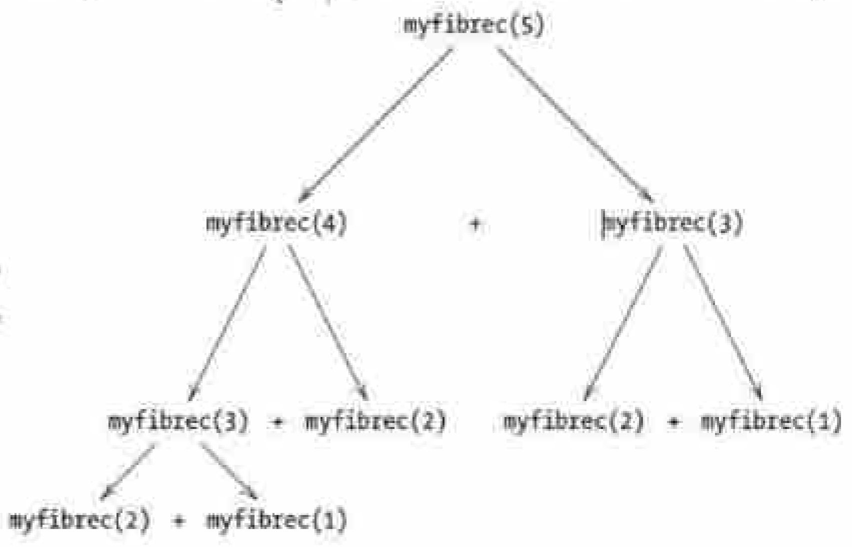
Output:

```
The 5th Fibonacci number is: 4
```

Explanation of above program:

• We handle the base cases where n is 1 or 2, returning 1 because the first two Fibonacci numbers are both 1.

• In the recursive case, for n greater than 2, we calculate the nth Fibonacci number by adding the (n-1)th and (n-2)nd Fibonacci numbers. We do this by calling the myfibrec function recursively with n - 1 and n - 2 as arguments. • Finally, we call the f myfibrec function with n = 5 and print the result.


myfibrec(5)
myfibrec(4) + myfibrec(3)
myfibrec(3) + myfibrec(2)
myfibrec(2) + myfibrec(1)
myfibrec(2) + myfibrec(1)

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) **Output:**

```
5
Warning message:
In warn_test(0) :
'x' is less than or equal to 0 but setting it to 1 and continuing
```

Explanation:

In warn_test, if x is nonpositive, the function issues a warning, and x is overwritten to be 1. warn_test has continued to execute and returned the value 5

Example for stop command:

```
error_test <- function(x){ if(x<=0){ stop("'x' is less
than or equal to 0... TERMINATE")
} return(5/x)
} error_test(0)
```

Output:

```
Error in error_test(0) : 'x' is less than or equal to 0... TERMINATE
```

Explanation:

In error_test, on the other hand, if x is nonpositive, the function throws an error and terminates immediately.

The call to error_test did not return anything because R exited the function at the stop command.

Catching Errors with try Statements

When a function terminates from an error, it also terminates any parent functions. For example, if function A calls function B and function B halts because of an error, this halts execution of A at the same point. To avoid
this severe consequence, you can use a try statement to attempt a function call and check whether it produces an error.

Example:

```
v<-c(1,2,4,'0',5) for
(i in v)
{
try(print(5/i))
}
```

Output:

```
    5
    2.5
    1.25
    Error in 5/I : non numeric argument to binary operator
    1
```

Explanation:
In the example given above we have code which has non-numeric value in the vector and we are trying to divide 5 with every element of the vector.
Using the try block we can see the code ran for all the other cases even after the error in one of the iteration.

Using tryCatch

The try block prevents your code from stopping but cannot provide a way to handle exceptions. Trycatch helps to handle the conditions and control what happens based on the conditions.

Syntax:

```
check = tryCatch({
expression
}, warning = function(w){
   code that handles the warnings
}, error = function(e){     code
that handles the errors }, finally
= function(f){     clean-up code
})
```

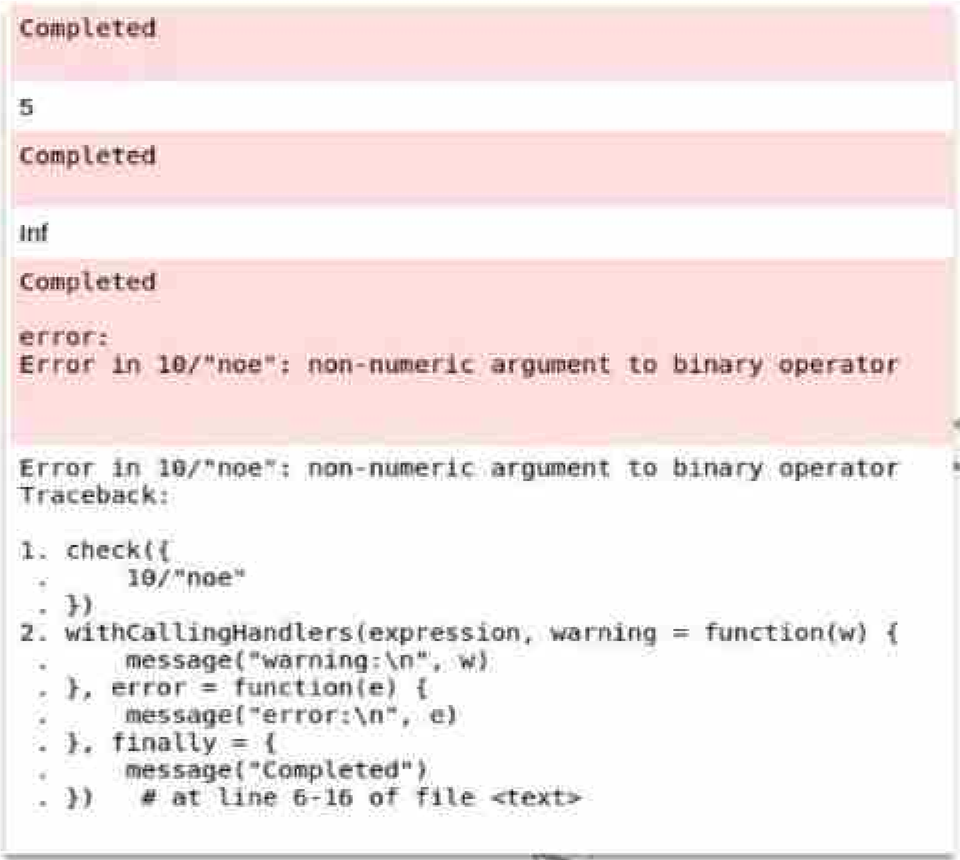
Example: check <-
function(expression){

```
withCallingHandlers(expression,

              warning = function(w){
              message("warning:\n", w)
              },
              error = function(e){
message("error:\n", e)
              },
              finally = {
              message("Completed")
              })
}

check({10/2}) check({10/0})
check({10/'noe'})
```

Output:



```
Completed

5
Completed

Inf
Completed

error:
Error in 10/"noe": non-numeric argument to binary operator


Error in 10/"noe": non-numeric argument to binary operator
Traceback:

1. check({
 .     10/"noe"
 . })
2. withCallingHandlers(expression, warning = function(w) {
 .     message("warning:\n", w)
 . }, error = function(e) {
 .     message("error:\n", e)
 . }, finally = {
 .     message("Completed")
 . })   # at line 6-16 of file <text>
```

Timing

it's often useful to keep track of progress or see how long a certain task took to complete.

If you want to know how long a computation takes to complete, you can use the **Sys.time** command.

This command outputs an object that details current date and time information based on your system.

Sys.time()

Output: "2016-03-06 16:39:27 NZDT"

The **Sys.sleep** command makes R pause for a specified amount of time, in seconds, before continuing.

Syntax:

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)
```

Output:

```
5.008 sec
```

Explanation:

Store/Record the time before the execution in the variable Starttime, then after the execution of the function, store the time in Endtime variable.

The difference between Endtime and Starttime gives the running time of the function.

Visibility /Progress Bar

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:

```
# global variable
global = 5

# global variable accessed from
# within a function display
= function()
{
print(global)
} display()

# changing value of global variable global
= 10
display()
```

Output:

```
5
10
```

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 can not be accessed or used outside that block. • Declaring local variables: Local variables are declared inside a block.

Example:

```
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()
```

Output:

```
Age is :18
```