CHAPTER 2 R PROGRAMMING

Syllabus

Reading and writing files, Conditions and Loops: standalone statements with illustrations in exercise, stacking statements, coding loops, Writing, and calling Functions, Exceptions, Timings, and Visibility, package in R.

File handling is one of the ways of storing data permanently in the form of a file on a computer hard disk drive(HDD) from where we can read it whenever required. Another option is storing data in the database which gives us much more options to search and retrieve the information using sql queries. So far the operations using the R programs are done on a prompt/terminal which is not stored anywhere. But in the software industry, most of the programs are written to store the information fetched from the program. One such way is to store and fetch information to/from a file. Two most common operations that can be performed on a file are:

- Importing or Reading Files in R
- Exporting or Writing Files in R

2.1 Reading and writing files

When a program is terminated, the entire data is lost, because it stores the data in variables which take place in primary memory, which is a volatile memory. Storing in a file will preserve our data even if the program terminates. If we have to enter a large amount of data, it will take a lot of time to enter them all. However, if we have a file containing all the data, we can easily access the contents of the file using a few commands in R. We can easily move our data from one computer to another without any changes. So those files can be stored in various formats. It may be stored in a .txt(tab-separated value) file, or in a tabular format that is a .csv(comma-separated value) file or it may be on the internet or cloud. R provides very easy methods to read these files.

In R programming, we can read and write files to perform tasks like importing data for analysis, saving results, or processing external data sources. R provides a variety of functions and packages to work with different types of files. Here are common methods for reading and writing files in R:

2.1.1 Reading Files in R:

Reading CSV Files:

To read data from a CSV (Comma-Separated Values) file, you can use the read.csv() function or its related functions. These functions allow you to import data into a data frame.

Example:
```
data <- read.csv("data.csv")
```

Reading Excel Files:

To read data from Excel files, you can use the readxl or openxlsx package to read Excel spreadsheets.

Example using readxl:
```
library(readxl)
data <- read_excel("data.xlsx")
```

Reading Text Files:

To read data from plain text files, you can use functions like read.table() or readLines(). The read.table() function is often used for reading tabular data in text files.

Example:
```
data <- read.table("data.txt", header = TRUE)
```

Reading Other Data Formats:

R supports various data import functions and packages to read other file formats, such as JSON, XML, SQLite databases, and more. For specific formats, you may need to install and load the corresponding packages.

Some more examples: One of the important formats to store a file is in a text file. R provides various methods that one can read data from a text file.

- **read.delim()**: This method is used for reading **.txt** files. By default, point (".") is used as a decimal point.

Syntax: read.delim(file, header = TRUE, sep = "\t", dec = ".", ...)

Here, **file** is the path to the file containing the data to be read into R. **header** is a logical value. If TRUE, **read.delim()** assumes that your file has a header row, so row 1 is the name of each column. If that's not the case, you can add the argument header = FALSE. **sep** is the field separator character. "\t" is used for a tab-delimited file. **dec** is the character used in the file for decimal points.

Example 1: Reading a text file using **read.delim()** function

```
myData = read.delim("datasetfile.txt", header = FALSE)
print(myData)
```

Example 2: Reading a text file using **file.choose(),** this code will ask the user to choose a file to read, note that it should be executed in RStudio.

```
myFile = read.delim(file.choose(), header = FALSE)
print(myFile)
```

Example 3: Reading file using **read_tsv( )** function using **readr** package.

```
# Importing the readr library
library(readr)
# Reading a text file using  read_tsv()
myData = read_tsv("datasetfile.txt", col_names = FALSE)
print(myData)
```

Example 4: We can also read a file line by line using the following code.

```
# Import the readr library
library(readr)
# Reading one line at a time using read_lines()
myData = read_lines("datasetfile.txt", n_max = 1)
print(myData)

# Reading two lines at a time using read_lines()
myData = read_lines("datasetfile.txt", n_max = 2)
print(myData)
```

Example 5: We can also read a whole file at a time using the **read_file( )** function.

```
myData = read_file("datasetfile.txt")
print(myData)
```

Example 6: We can also read a file and display the contents in a tabular format using read.table, read.csv, read.csv2 and read_csv functions in R programming language in the same ways as we have done in the above examples. For example,

```
myData = read.table("datasetfile.csv")
print(myData)
myData = read.csv("datasetfile.csv")
print(myData)
myData = read.csv2("datasetfile.csv")
print(myData)
```

2.1.2 Writing a file using R programming:

Reading and writing files is an important process of data analytics which is done in R with ease using built in functions and libraries. We can read and write data to txt, csv or excel files.

Writing CSV Files: To write data to a CSV file, you can use the write.csv() function. It allows you to save data frames to CSV format. CSV stands for Comma Separated Values. These files are used to handle a large amount of statistical data. Following is the syntax to write to a CSV file:

Examples:

```
write.csv(mydata, file="my_data.csv")

write.csv2(mydata, file="my_data.csv")

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

write.csv(data, "output.csv", row.names = FALSE)
```

Writing Data to Excel Files:

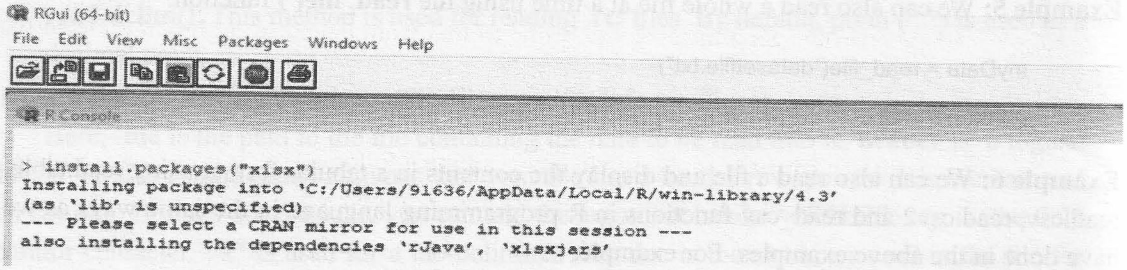
To write data to Excel files, you can use packages like writexl or openxlsx to create Excel spreadsheets.

Example using writexl:

```
library(writexl)

write_xlsx(data, "output.xlsx")
```

To write data to excel we need to install the package known as **"xlsx package"**, it is basically a java based solution for reading, writing, and committing changes to excel files. It can be installed as follows: **install.packages("xlsx")**



```
> install.packages("xlsx")
Installing package into 'C:/Users/91636/AppData/Local/R/win-library/4.3'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
also installing the dependencies 'rJava', 'xlsxjars'
```

Fig 2.1: installation of xlsx packages in R platform.

Following code can be written post installing the xlsx packages.

```
library("xlsx")
write.xlsx(mydata, file = "result.xlsx", sheetName = "my_data", append = FALSE)
```

After executing the above code mydata content will be written to the **result.xlsx** file with sheet name my_data.

Writing Data to a text file: Text files are commonly used in almost every application. Writing to .txt files is very similar to that of the CSV files. Following is the syntax to write to a text file:

```
write.table(my_data, file = "my_data.txt", sep = "")
```

Writing Other Data Formats: R supports various data export functions and packages to write data in formats such as JSON, XML, databases, and more. Depending on your needs, you may need to install and load the appropriate packages. Remember that file paths should be provided as strings with the correct file extension (e.g., ".csv", ".xlsx") or full file paths if the file is not in the working directory.

Before reading or writing files, make sure to install and load any required packages. The specific functions and packages you use will depend on the format of the data you are working with.

2.2 Programming with input and output statements:

Most GUI programs today use a dialog box for asking the user to provide some type of input. Like other programming languages, in R it is also possible to take input from the user using two methods: **readline()** method and **scan()** method.

In R, we can perform input and output operations to interact with the user, read data from external sources, and display results. Here are some commonly used input and output statements and functions in R:

2.2.1 Input Statements:

Reading from the Console: readline(): Reads a line of text from the console. It's often used for taking user input.

Example: `name <- readline("Enter your name: ")`

Reading Numeric Input from the Console: readline() followed by as.numeric(): You can use readline() to read a line of text from the console and then convert it to a numeric data type using as.numeric().

Example: `age <- as.numeric(readline("Enter your age: "))`

Reading from Files: Functions like read.table(), read.csv(), and package-specific functions (e.g., readxl::read_excel()) are used to read data from files. These functions allow you to import data from external sources, such as CSV, Excel, or text files.

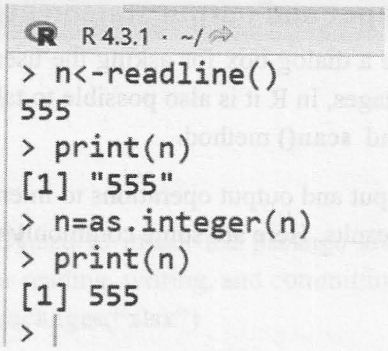
Example: `data <- read.csv("data.csv")`

Readline() method: In R language readline() method takes input in string format. If a user inputs an integer then it is taken as a string, lets say, one wants to input 255, then it will input as "255", like a string. So we need to convert that input value to the format that we need. In this case, string "255" is converted to integer 255. To convert the input value to the desired data type, there are some functions in R as given below:

```
n=readline();
as.integer(n);          # to convert to integer
as.numeric(n);   # to convert to numeric type (float, double etc)
as.complex(n);   #to convert to complex number (i.e 3+2i)
as.character(n)  # to convert to character
as.Date(n)              #to convert to date.
```

While taking input strings like name, address it does not require conversion but other than string data need to be converted to respective data type using above methods.

Examples:

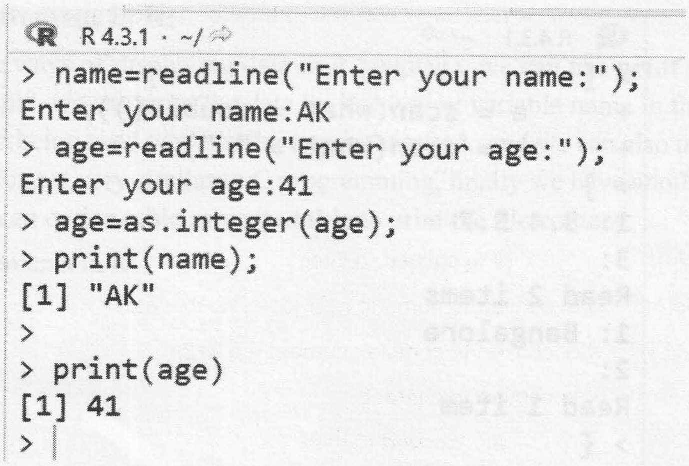

R 4.3.1 · ~/

```
> n<-readline()
555
> print(n)
[1] "555"
> n=as.integer(n)
> print(n)
[1] 555
>
```

We can even use the readline( ) method with a prompt argument to ask the user what to input in the program. Actually prompt argument facilitates other functions to construct files documenting. But prompt is not mandatory to use all the time. We can also use just a message in the readline( ) to specify the input data.

Examples:

```
name=readline("Enter your Name:");   OR
name=readline(prompt="Enter your name:");
age=readline("Enter your age:");
age=as.integer(age);
print(name);
print(age);
```


R 4.3.1 · ~/

```
> name=readline("Enter your name:");
Enter your name:AK
> age=readline("Enter your age:");
Enter your age:41
> age=as.integer(age);
> print(name);
[1] "AK"
>
> print(age)
[1] 41
>
```

We can use multiple readline( ) statements in { } open and close braces to take multiple input. For example:

```
{
        name=readline("Enter your name:");
        age=readline("Enter your age:");
        phone=readline("Enter your phone:");
}
```

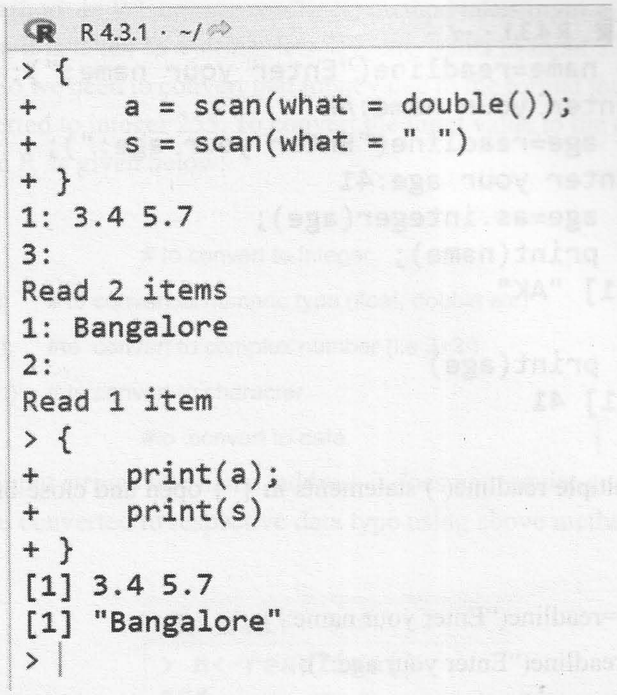
Scan method( ) : Scan( ) is another input method to accept data entry from the user at the run time in the program. This method is a very handy method while inputs are needed to be taken quickly for any mathematical calculation or for any dataset. It reads data in the form of a vector or list. This method also reads input from a file. While taking input using scan we must press Enter key 2 times to stop the entry.

Example:

```
# taking input using scan()
list = scan()
print(list)
```

We can also specify what data we are going to accept using **what** parameter in the scan.

```
a = scan(what = double());          #for double
s = scan(what = " ")                #for string
ch = scan(what = character())       #for character
```



```
R 4.3.1 · ~/
> {
+     a = scan(what = double());
+     s = scan(what = " ")
+ }
1: 3.4 5.7
3:
Read 2 items
1: Bangalore
2:
Read 1 item
> {
+     print(a);
+     print(s)
+ }
[1] 3.4 5.7
[1] "Bangalore"
> 
```

Reading file using scan: We can read a file using scan method by specifying the file name and data type to the scan() method as given below:

```
# taking input from the user as a string file
    sf = scan("fileString.txt", what = " ")

# double file input using scan()
    df = scan("fileDouble.txt", what = double())

# character file input using scan()
    cf = scan("fileChar.txt", what = character())

# print the inputted values
    print(sf)
    print(df)
    print(cf)
```

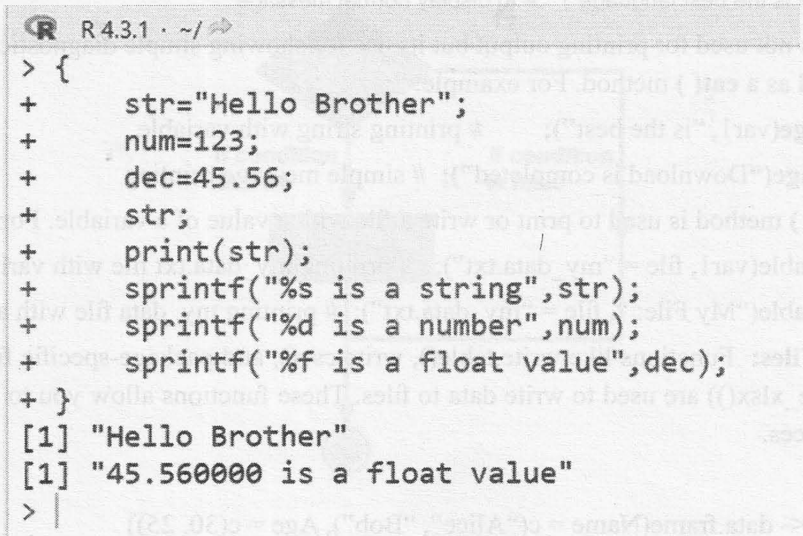
2.2.3 Output Statement in R:

There are multiple ways of displaying data in R language, we can use **print** statements as used in the above example, we can print the data by just typing variable name in the console, **cat** and **message( )** are also being used very similar to print method, and we can also use **sprintf( )** method with data type format very similar to C programming, finally we have another method write( ) which is used with an option table as **write.table** to print the file content.

List of output statements in R:

- print ( )
- cat ( )
- message( )
- sprintf( )
- write.table( )

Example:



```
> {
+     str="Hello Brother";
+     num=123;
+     dec=45.56;
+     str;
+     print(str);
+     sprintf("%s is a string",str);
+     sprintf("%d is a number",num);
+     sprintf("%f is a float value",dec);
+ }
[1] "Hello Brother"
[1] "45.560000 is a float value"
> 
```

Printing to the Console:

- print(): Prints values to the console.
- cat(): Concatenates and prints values to the console.

Example:

```
x <- 42
y <- "Hello, World!"
print(x)
cat("The value of x is", x, "and y is", y, "\n")
```

Formatting Output:

- sprintf(): Formats values as a string using C-style format specifications.
- paste(): Combines strings and variables.

Example:

```
x <- 42
y <- "Hello, World!"
formatted_str <- sprintf("The value of x is %d and y is %s", x, y)
paste("The value of x is", x, "and y is", y)
```

Cat( ) is the same as **print()** function. **cat()** converts its arguments to character strings. This is useful for printing output in user defined functions.

```
var1 = "Bangalore "
cat(var1, "is the best \n")  # to display variable's value with message
cat("R is the best language")    # to display normal message
```

Message( ) is not used for printing output but its use for showing simple diagnostic messages but it can be used as a **cat( )** method. For example:

message(var1, "is the best"**);**   # printing string with variable

message("Download is completed"**);** # simple message printing

Write.table( ) method is used to print or write a file with a value of a variable. For example:

write.table(var1, file = "my_data.txt"); # printing my_data.txt file with variable var1

write.table("My File: ", file = "my_data.txt"); # printing my_data file with a text.

Writing to Files: Functions like write.table(), write.csv(), and package-specific functions (e.g., writexl::write_xlsx()) are used to write data to files. These functions allow you to export data to external sources.

Example:

```
data <- data.frame(Name = c("Alice", "Bob"), Age = c(30, 25))
write.csv(data, "output.csv", row.names = FALSE)
```

2.3 Conditions statements:

Like other programming languages, R also has all the conditional statements like if, if-else and switch statements. These statements are used to verify the conditional statements and execute the task based on conditions.

- If statements
- If-else statements

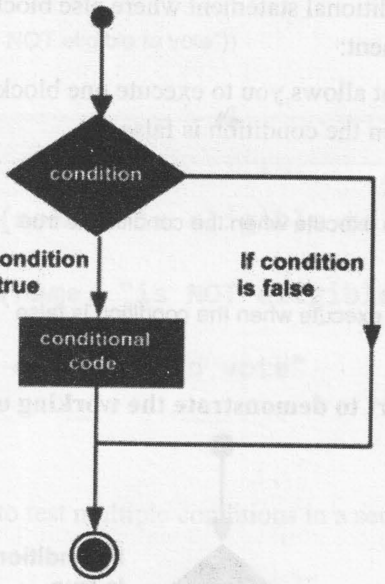
- if-else if-else Statement:
- Switch statement

If statement, is a conditional statement which takes an argument as a condition/s, if the condition is true then it executes the statement written in the if block.

The if statement is used to execute a block of code if a specified condition is true. The basic syntax is as follows:

```
if (condition) {
        # Code to execute when the condition is true
}
```

Flowchart to demonstrate the working of if statement.

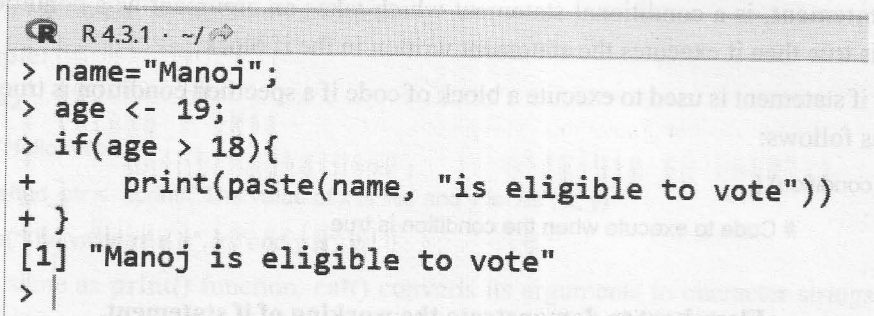


Example 1 :

```
x <- 10
if (x > 5) {
        cat("x is greater than 5.\n")
}
```

Example 2 :

```
name="Manoj";
age <- 19;
if(age > 18){
        print(paste(name, “is eligible to vote”))
    }
```



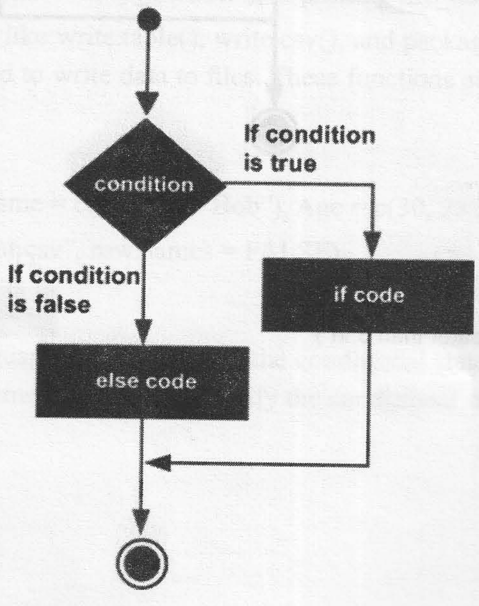
```
R 4.3.1 · ~/
> name="Manoj";
> age <- 19;
> if(age > 18){
+     print(paste(name, "is eligible to vote"))
+ }
[1] "Manoj is eligible to vote"
> 
```

If-else statement is a conditional statement where else block code gets executed when the condition is false. if-else Statement:

The if-else statement allows you to execute one block of code when a condition is true and another block of code when the condition is false.

```
if (condition) {
        # Code to execute when the condition is true
} else {
        # Code to execute when the condition is false
}
```

Flowchart to demonstrate the working of if-else statements.

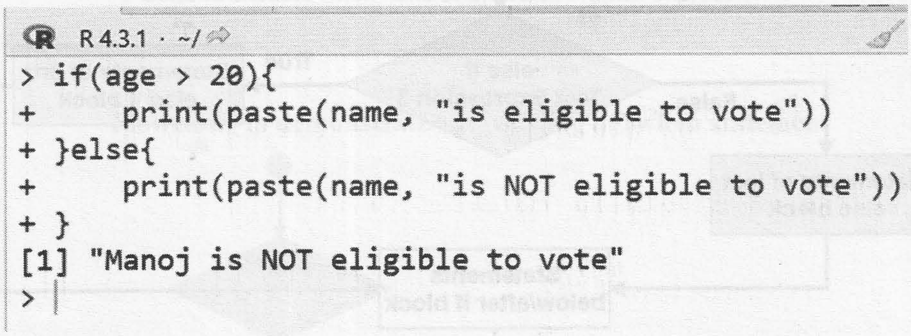


Example 1:

```
x <- 3
if (x > 5) {
        cat("x is greater than 5.\n")
} else {
        cat("x is not greater than 5.\n")
}
```

Example 2:

```
if(age > 20){
        print(paste(name, "is eligible to vote"))
}else{
        print(paste(name, "is NOT eligible to vote"))
}
```



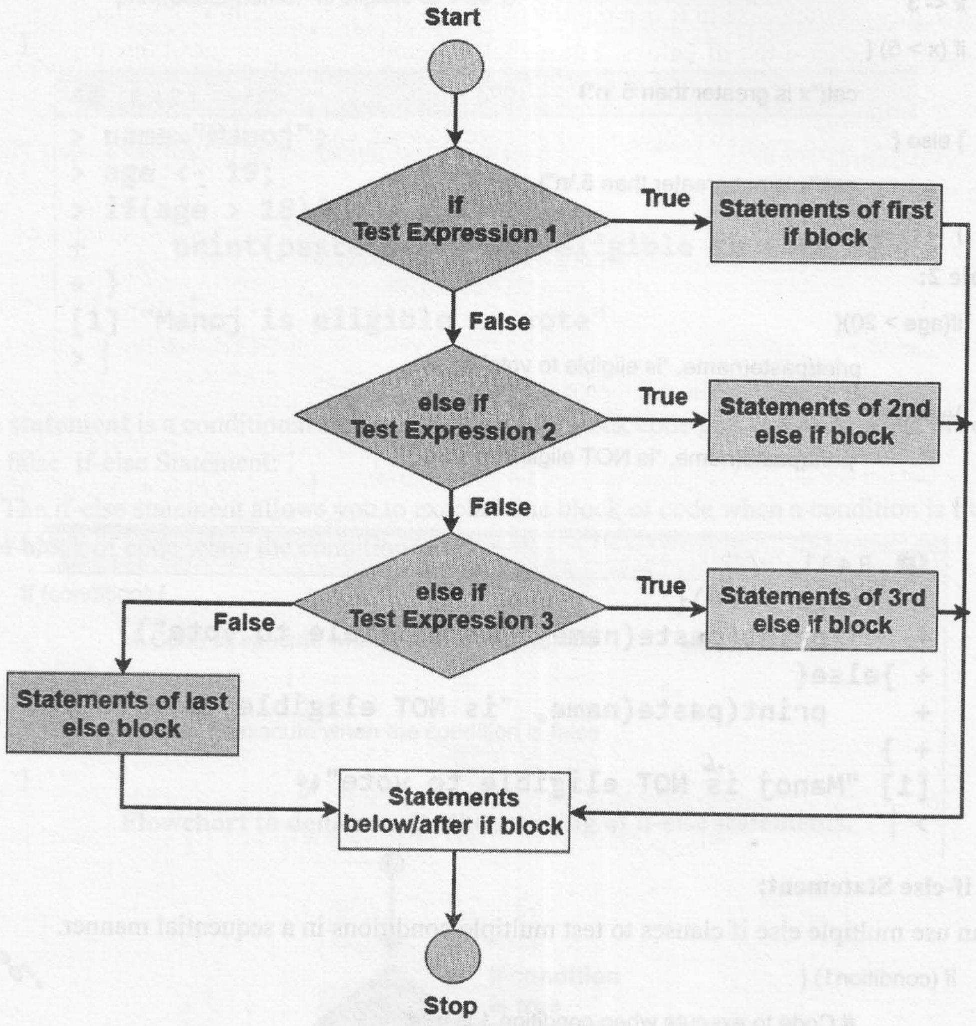
```
R 4.3.1 · ~/
> if(age > 20){
+     print(paste(name, "is eligible to vote"))
+ }else{
+     print(paste(name, "is NOT eligible to vote"))
+ }
[1] "Manoj is NOT eligible to vote"
>
```

if-else if-else Statement:

You can use multiple else if clauses to test multiple conditions in a sequential manner.

```
if (condition1) {
        # Code to execute when condition 1 is true
} else if (condition2) {
        # Code to execute when condition 2 is true
} else {
        # Code to execute when none of the conditions are true
}
```

Flowchart to demonstrate the working of if-else-if statements.


Start
if Test Expression 1
True
Statements of first if block
False
else if Test Expression 2
True
Statements of 2nd else if block
False
else if Test Expression 3
True
Statements of 3rd else if block
False
Statements of last else block
Statements below/after if block
Stop

Example:

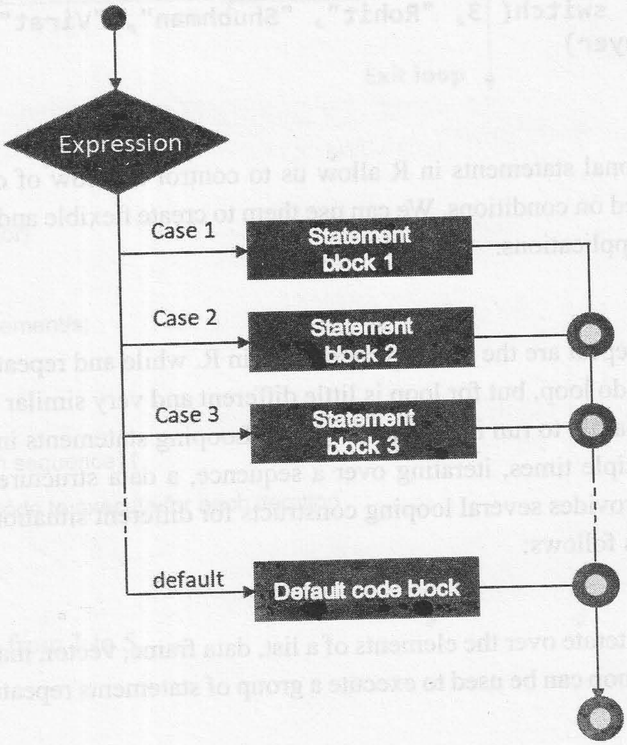
```
x <- 8
if (x > 10) {
        cat("x is greater than 10.\n")
} else if (x > 5) {
        cat("x is greater than 5 but not greater than 10.\n")
} else {
        cat("x is not greater than 5.\n")
}
```

Switch Statement in R is a substitute for long if statements that compare a variable to several integral values. Switch case in R is a multiway branch statement. It allows a variable to be tested for **equality against a list of values.** The switch statement is used to select one of several code blocks to execute based on a specified condition.

Syntax: switch(expression, case1, case2, case3,....) OR

```
switch(expression,
 value1 = {
            # Code to execute when expression equals value1
  },
  value2 = {
            # Code to execute when expression equals value2
  },
  default = {
            # Code to execute when none of the values match expression
   }
)
```

Flowchart to demonstrate the working of switch statement



Example 1:

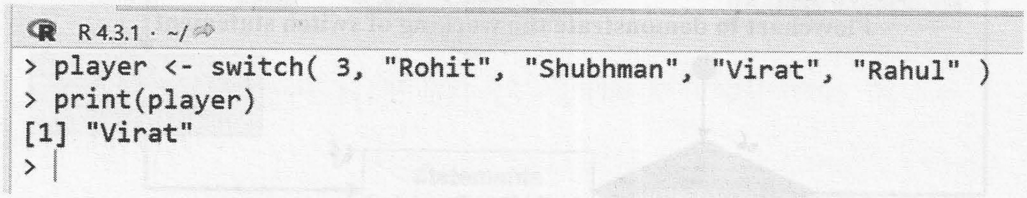
```
day <- "Sunday"
switch(day,
        "Monday" = cat("It's the start of the week.\n"),
        "Saturday" = cat("It's the weekend.\n"),
        default = cat("It's a regular day.\n")
)
```

Some key points to remember about the switch statement:

- If the expression type is a character string, the string is matched to the listed cases.
- If there is more than one match, the first match element is used.
- No default case is available.
- If no case is matched, an unnamed case is used.

Example 2:

```
player <- switch( 3, "Rohit", "Shubhman", "Virat",   "Rahul" )
print(player)
```



```
R 4.3.1 · ~/
> player <- switch( 3, "Rohit", "Shubhman", "Virat", "Rahul" )
> print(player)
[1] "Virat"
> |
```

These conditional statements in R allow us to control the flow of our code and perform different actions based on conditions. We can use them to create flexible and responsive programs for a wide range of applications.

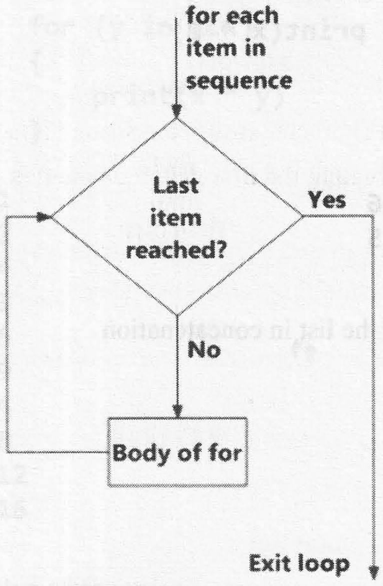
2.4 Loops:

For loop, while and repeat are the looping statements in R. while and repeat are similar to C programming while and do loop, but for loop is little different and very similar to for-in statement of javascript which is mainly to run in the list or object. Looping statements in R allow us to repeat a block of code multiple times, iterating over a sequence, a data structure, or until a specified condition is met. R provides several looping constructs for different situations. The main looping statements in R are as follows:

2.4.1 For Loop

For Loop is useful to iterate over the elements of a list, data frame, vector, matrix, or any other object. It means the for loop can be used to execute a group of statements repeatedly depending upon the number of elements in the object. It is an entry-controlled loop, in this loop, the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false. The for loop is used to iterate over a sequence of values, such as a vector, list, or numeric range. It repeatedly executes a block of code for each value in the sequence.

Flowchart to demonstrate the working of for-in loop



Syntax:

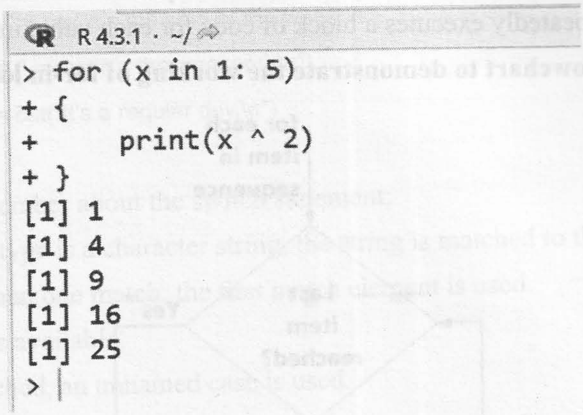
Syntax:

```
for (var in vector)
 {
            statement/s;
  }
OR
for (variable in sequence) {
            # Code to execute for each iteration
}
```

Example1: for loop from 1 to 5

```
for (x in 1: 5)
```

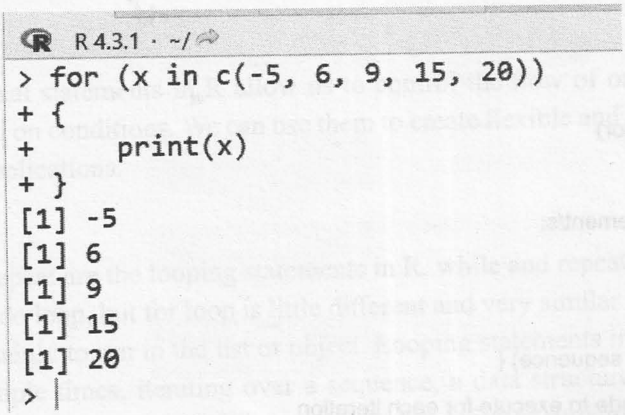
```
{
        print(x ^ 2)
}
```



```
R 4.3.1 · ~/
> for (x in 1: 5)
+ {
+       print(x ^ 2)
+ }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
>
```

Example2: for loop running in the list in concatenation

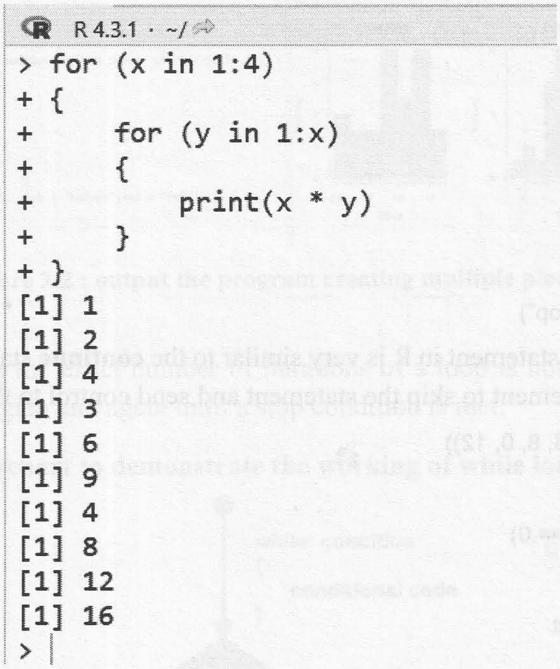
```
for (x in c(-5, 6, 9, 15, 20))
{
        print(x)
}
```



```
R 4.3.1 · ~/
> for (x in c(-5, 6, 9, 15, 20))
+ {
+       print(x)
+ }
[1] -5
[1] 6
[1] 9
[1] 15
[1] 20
>
```

Example3: nested for loop

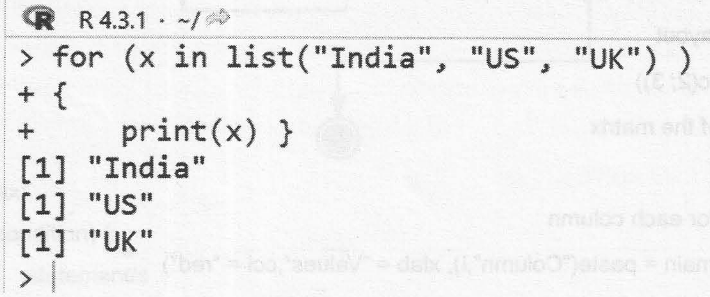
```
for (x  in 1:4)
{
    for (y  in 1:x)
    {
        print(x * y)
    }
}
```



```
> for (x in 1:4)
+ {
+     for (y in 1:x)
+     {
+         print(x * y)
+     }
+ }
[1] 1
[1] 2
[1] 4
[1] 3
[1] 6
[1] 9
[1] 4
[1] 8
[1] 12
[1] 16
>
```

Example3: running for loop in a list of countries.

```
for (x  in  list("India", "US", "UK") )
{
        print(x)
                                        }
```



```
> for (x in list("India", "US", "UK") )
+ {
+     print(x) }
[1] "India"
[1] "US"
[1] "UK"
>
```

Example4: for loop with break statement to break the loop with a condition.

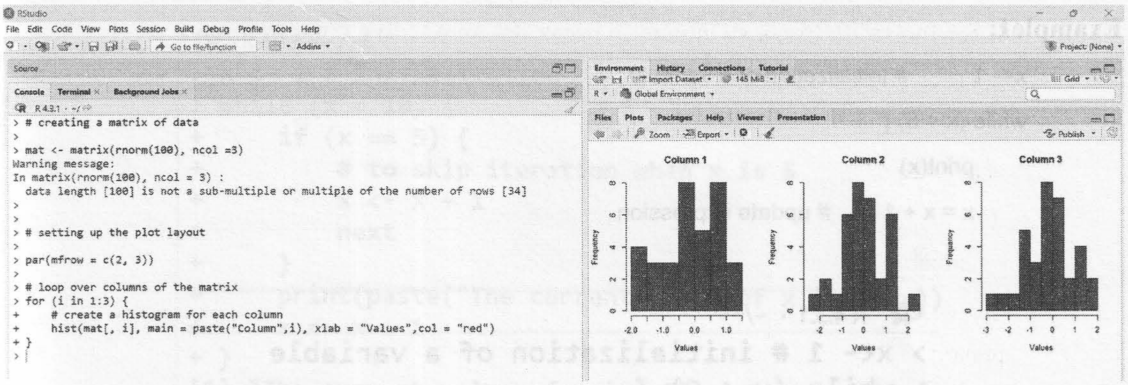
```
for (x in c(2, 4, 6, 8, 0, 12))
{
   if (x == 0)
   {
      break
   }
  print(x)
}
print("End of Loop")
```

Example 5: The next statement in R is very similar to the **continue** statement of C which is used with a conditional statement to skip the statement and send control to the loop condition.

```
for (x in c(2, 4, 6, 8, 0, 12))
{
           if (x == 0)
          {
            next
          }
          print(x)
}
print('end of loop')
```

Example 6: for loop to create histogram multiple plots

```
# creating a matrix of data
        mat <- matrix(rnorm(100), ncol =3)
# setting up the plot layout
        par(mfrow = c(2, 3))
# loop over columns of the matrix
        for (i in 1:3) {
# create a histogram for each column
        hist(mat[, i], main = paste("Column",i), xlab = "Values",col = "red")
        }
```



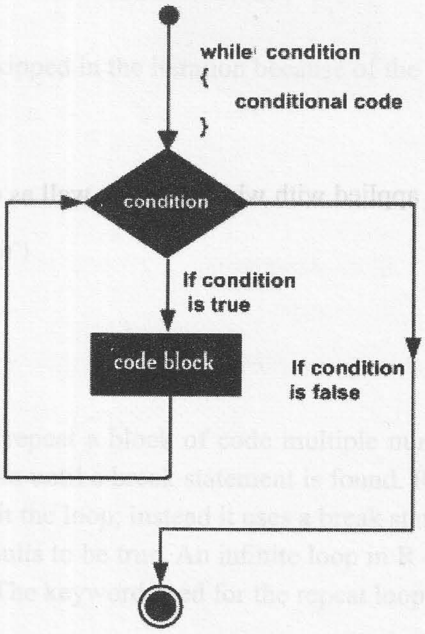
```
> # creating a matrix of data
>
> mat <- matrix(rnorm(100), ncol =3)
Warning message:
In matrix(rnorm(100), ncol = 3) :
  data length [100] is not a sub-multiple or multiple of the number of rows [34]
>
>
> # setting up the plot layout
>
> par(mfrow = c(2, 3))
>
> # loop over columns of the matrix
> for (i in 1:3) {
+     # create a histogram for each column
+     hist(mat[, i], main = paste("Column",i), xlab = "Values",col = "red")
+ }
>
```

Figure 2.2 : output the program creating multiple plots

2.4.2 :While loop

While loop is used when the exact number of iterations of a loop is not known beforehand. It executes the same code again and again until a stop condition is met.

Flowchart to demonstrate the working of while loop.



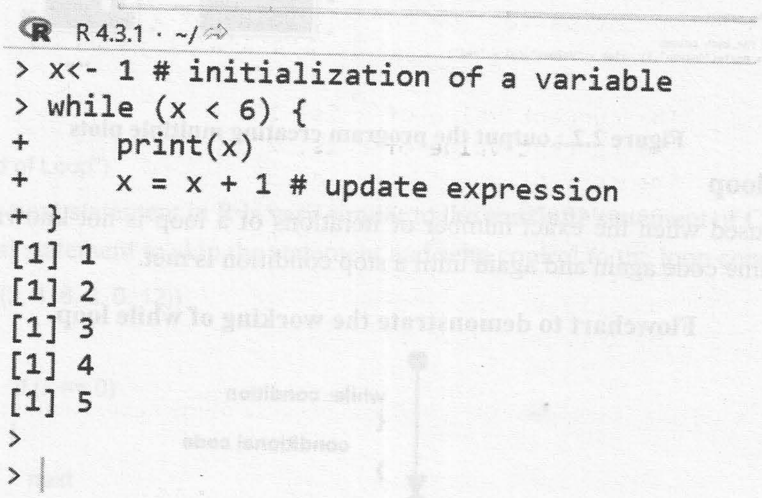
While loop Syntax:

```
        while (condition) {
                statement/s
                update_expression/increment or decrement of the variable

        }
```

Example1:

```
x<- 1          # initialization of a variable
while (x < 6) {
    print(x)
    x = x + 1      # update expression
}
```

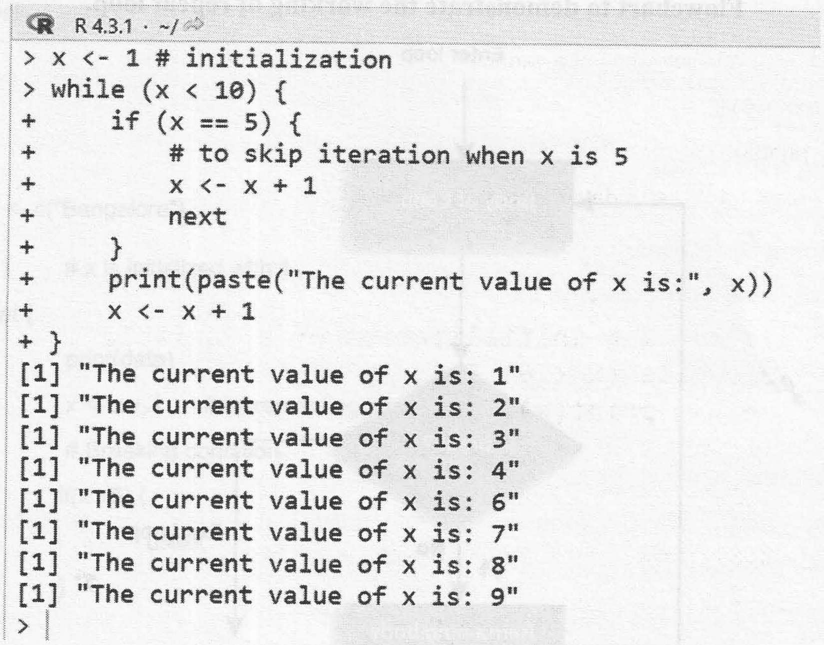


```
R 4.3.1 · ~/
> x<- 1 # initialization of a variable
> while (x < 6) {
+     print(x)
+     x = x + 1 # update expression
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
>
>
```

Break and **next** can be applied with while loops as well as demonstrated in the above for loop examples.

Example 2:

```
x <- 1      # initialization
while (x < 10) {
 if (x == 5) {
  # to skip iteration when x is 5
  x <- x + 1
  next
 }
 print(paste("The current value of x is:", x))
 x <- x + 1
}
```


R 4.3.1 · ~/

```
> x <- 1 # initialization
> while (x < 10) {
+     if (x == 5) {
+         # to skip iteration when x is 5
+         x <- x + 1
+         next
+     }
+     print(paste("The current value of x is:", x))
+     x <- x + 1
+ }
[1] "The current value of x is: 1"
[1] "The current value of x is: 2"
[1] "The current value of x is: 3"
[1] "The current value of x is: 4"
[1] "The current value of x is: 6"
[1] "The current value of x is: 7"
[1] "The current value of x is: 8"
[1] "The current value of x is: 9"
> 
```

the 5th value of x is skipped in the iteration because of the next statement.

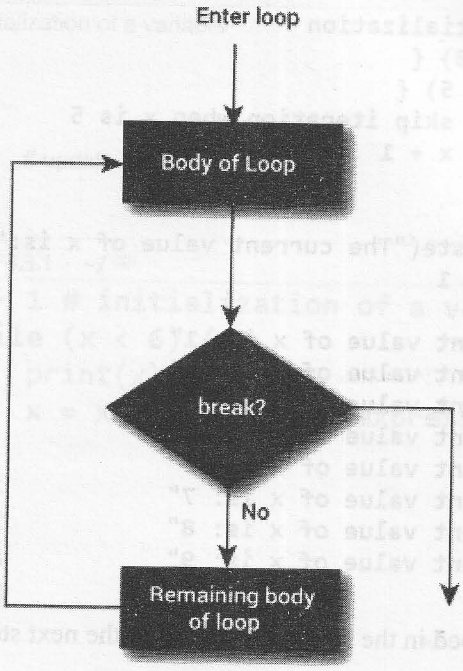
Example 3:

```
count <- 1
while (count <= 5) {
        cat("Count:", count, "\n")
       count <- count + 1
}
```

2.4.3: Repeat loop

Repeat loop in R is used to repeat a block of code multiple number of times. And it executes the same code again and again until a break statement is found. Repeat loop, unlike other loops, doesn't use a condition to exit the loop; instead it uses a break statement that executes if a condition within the loop body results to be true. An infinite loop in R can be created very easily with the help of the Repeat loop. The keyword used for the repeat loop is 'repeat'.

Flowchart to demonstrate the working of repeat loop.



Syntax:

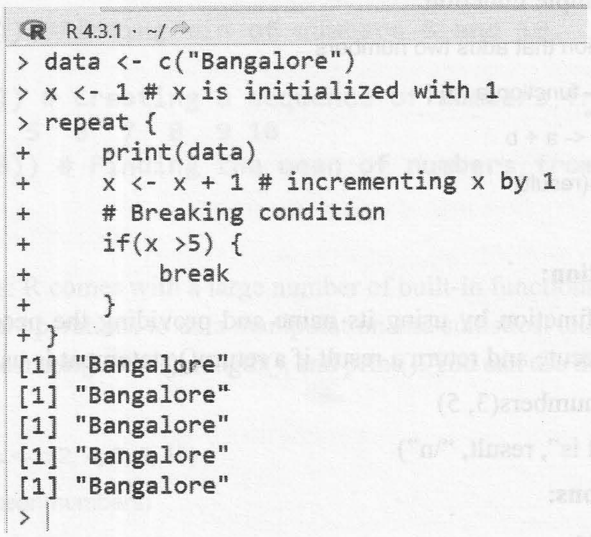
```
        repeat {
                Block statement/s
                if(condition) {
                        break
                }
        }
```

Example 1:

```
count <- 1
repeat {
    cat("Count:", count, "\n")
      count <- count + 1
      if (count > 5) {
        break
    }
}
```

Example 2:

```
    data <- c("Bangalore")
    x <- 1        # x is initialized with 1
    repeat {
            print(data)
            x <- x + 1      # incrementing x by 1
            # Breaking condition
            if(x >5) {
                break
            }
    }
```


R 4.3.1 · ~/

```
> data <- c("Bangalore")
> x <- 1 # x is initialized with 1
> repeat {
+     print(data)
+     x <- x + 1 # incrementing x by 1
+     # Breaking condition
+     if(x >5) {
+         break
+     }
+ }
[1] "Bangalore"
[1] "Bangalore"
[1] "Bangalore"
[1] "Bangalore"
[1] "Bangalore"
> 
```

2.5 Functions

Functions are useful when we want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In the R program we can create functions, the function name and the file in which we are creating the function need not be the same and we can have multiple functions in a program.

Functions are an essential concept in R programming. They are blocks of code that perform specific tasks or operations and can be called and reused multiple times. Functions in R can take arguments (input values), process them, and return results (output values). Here's how to define and use functions in R:

I. Defining a Function:

You can define a function in R using the function() keyword. The basic syntax of a function definition is as follows:

```
function_name <- function(arg1, arg2, ...) {
            # Function body: code to perform the task
            return(result)  # Optional: return a result
}
```

- function_name is the name you choose for your function.
- arg1, arg2, ... are the arguments (input values) that the function accepts.
- The function body contains the code to perform the desired task.
- return(result) is optional and used to return a result.

Example of a Simple Function:

```
# Define a function that adds two numbers
add_numbers <- function(a, b) {
            result <- a + b
            return(result)
}
```

II. Calling a Function:

You can call a function by using its name and providing the necessary arguments. The function will execute and return a result if a return() statement is used.

```
result <- add_numbers(3, 5)
cat("The result is", result, "\n")
```

III. Types of functions:

- Built-in functions
- User defined functions

2.5.1 Built-in functions

Built-in functions are available in R which we can use to perform any common tasks. While user defined functions are created by programmers i.e. by us to write common code in a defined block with or without parameters.

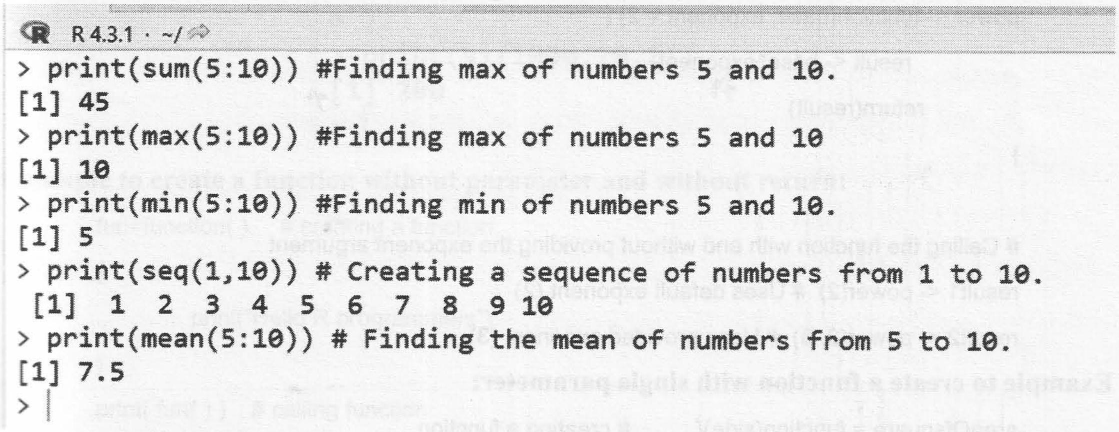
Some examples of built-in functions are: sum( ), max( ), min( ), seq( ), mean( ) etc.

sum( ) is used to perform addition on the given range in the sum( ) function. Max and min are to return maximum and minimum from the given range, seq( ) is used to generate the sequence numbers between the given range and mean is to compute the mean value for the given range.

Examples:

```
print(sum(5:10))          #Finding max of numbers 5 and 10.
print(max(5:10))          #Finding max of numbers 5 and 10.
print(min(5:10))          #Finding min of numbers 5 and 10.
print(seq(1,10))  # Creating a sequence of numbers from 1 to 10.
print(mean(5:10))         # Finding the mean of numbers from 5 to 10.
```

Execution:



```
R 4.3.1 · ~/
> print(sum(5:10)) #Finding max of numbers 5 and 10.
[1] 45
> print(max(5:10)) #Finding max of numbers 5 and 10
[1] 10
> print(min(5:10)) #Finding min of numbers 5 and 10.
[1] 5
> print(seq(1,10)) # Creating a sequence of numbers from 1 to 10.
 [1]  1  2  3  4  5  6  7  8  9 10
> print(mean(5:10)) # Finding the mean of numbers from 5 to 10.
[1] 7.5
> |
```

Built-in Functions: R comes with a large number of built-in functions that perform various tasks, from mathematical operations to data manipulation and statistical analysis. Some common built-in functions include mean(), sum(), length(), and print(). You can use these functions directly without defining them.

```
numbers <- c(2, 4, 6, 8, 10)
avg <- mean(numbers)
cat("Average:", avg, "\n")
```

2.5.2 User Defined functions:

We can create our own functions for any specific tasks which can be reused in the program. A function can take zero, single or multiple parameters and can return results as a single data or a list.

Function Arguments: Functions can have both named and unnamed arguments. Named arguments make it clear which value is being passed to which parameter, and you can use them in any order. Unnamed arguments are matched in the order they appear in the function definition.

```
# Using named arguments
result1 <- add_numbers(a = 5, b = 3)

# Using unnamed arguments
result2 <- add_numbers(5, 3)
```

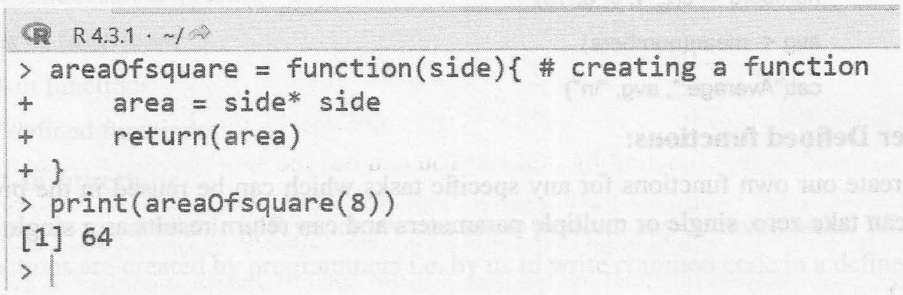
Default Values for Arguments: You can set default values for function arguments, which are used when the caller does not provide a value for that argument.

```
# Define a function with default values
power <- function(base, exponent = 2) {
        result <- base^exponent
       return(result)
}

# Calling the function with and without providing the exponent argument
result1 <- power(2)  # Uses default exponent (2)
result2 <- power(2, 3)  # Uses provided exponent (3)
```

Example to create a function with single parameter:

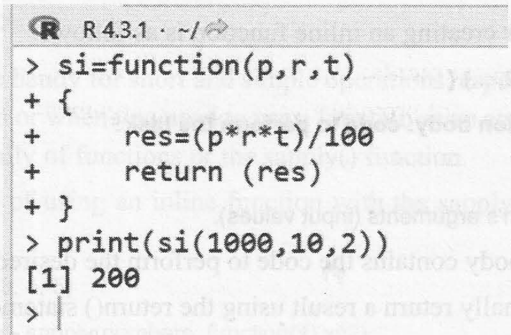
```
areaOfsquare = function(side){        # creating a function
        area = side* side
       return(area)
}
print(areaOfsquare(8))                # calling the function
```



```
> areaOfsquare = function(side){ # creating a function
+     area = side* side
+     return(area)
+ }
> print(areaOfsquare(8))
[1] 64
> 
```

Example to create a function with multiple parameter:

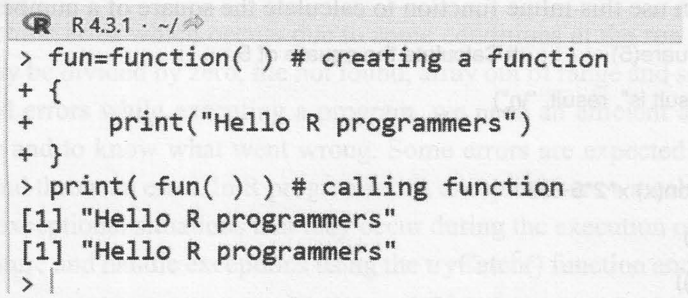
```
si=function(p,r,t)
{
          res=(p*r*t)/100
          return (res)
}
print(si(1000,10,2))
```



```
R 4.3.1 · ~/
> si=function(p,r,t)
+ {
+     res=(p*r*t)/100
+     return (res)
+ }
> print(si(1000,10,2))
[1] 200
>
```

Example to create a function without parameter and without return:

```
fun=function( )   # creating a function
{
          print("Hello R programmers")
}
print( fun( ) )   # calling function
```



```
R 4.3.1 · ~/
> fun=function( ) # creating a function
+ {
+     print("Hello R programmers")
+ }
> print( fun( ) ) # calling function
[1] "Hello R programmers"
[1] "Hello R programmers"
>
```

Variable Scoping: R uses lexical scoping, which means that variables defined within a function are local to that function by default. You can use the <<- operator to assign values to global variables from within a function. Functions are a fundamental building block in R and are used extensively for organizing code, improving code reusability, and encapsulating specific tasks. You can create custom functions tailored to your specific needs, making R a versatile and powerful programming language for data analysis and statistical computing.

2.5.3 Inline Functions:

We write inline functions when it's small and other looping statements are not used in the function, it executes faster compared to other functions. To create an inline function we have to use the function command with the argument x and then the expression of the function.

In R, you can create inline or anonymous functions using the **function()** keyword, which allows you to define a small, unnamed function on the fly. These inline functions are often used for simple, one-time calculations or operations. They are also known as "lambda" functions or "anonymous" functions.

The basic syntax for creating an inline function is as follows:

```
function(arg1, arg2, ...) {
        # Function body: code to perform the task
    }
```

arg1, arg2, ... are the function's arguments (input values).

- The function body contains the code to perform the desired task.
- You can optionally return a result using the return() statement.

Example 1 of an inline function that squares a number:

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

We can also create an equivalent inline function using the function() keyword:

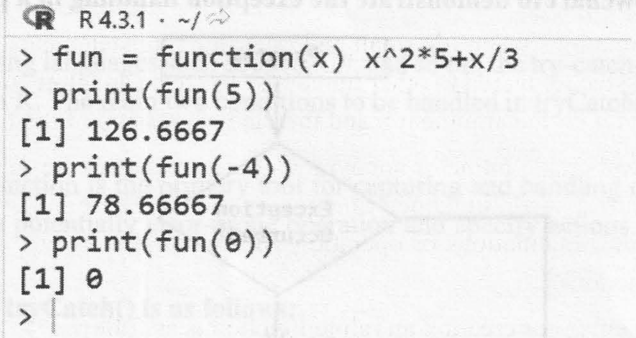
```
square <- function(x) x^2
```

You can then use this inline function to calculate the square of a number, as shown here:

```
result <- square(5)        # Calculate the square of 5
cat("The result is", result, "\n")
```

Example 2:

```
fun = function(x) x^2*5+x/3
print(fun(5))
print(fun(-4))
print(fun(0))
```



```
R 4.3.1 · ~/
> fun = function(x) x^2*5+x/3
> print(fun(5))
[1] 126.6667
> print(fun(-4))
[1] 78.66667
> print(fun(0))
[1] 0
>
```

Inline functions are handy for short and simple operations, especially when you don't want to create a named function or when you need to use a function as an argument to another function, such as in the apply() family of functions or the sapply() function.

Here's an example of using an inline function with the sapply() function to apply it to a vector of numbers:

```
numbers <- c(1, 2, 3, 4, 5)
squared_numbers <- sapply(numbers, function(x) x^2)
cat("Squared numbers:", squared_numbers, "\n")
```

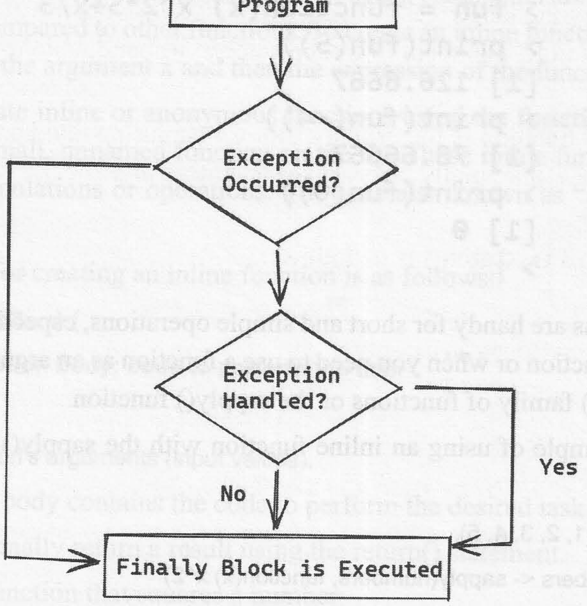
In this example, the inline function function(x) x^2 is applied to each element of the numbers vector, resulting in a new vector with the squared values.

Inline functions are a powerful and flexible feature in R that allow you to create and use functions quickly for specific tasks without the need to define a named function.

2.6 Exceptions:

Exception is a runtime error which occurs due to some conditions at the run time of a program. Runtime errors may be divided by zero, file not found, array out of range and so on. If we encounter any unexpected errors while executing a program, we need an efficient and interactive way to debug the error and to know what went wrong. Some errors are expected but sometimes the models fail to fit and throw an error. In R programming, exceptions are a mechanism for handling runtime errors or exceptional situations that may occur during the execution of a program. R provides a way to capture and handle exceptions using the tryCatch() function and related constructs.

Flowchart to demonstrate the exception handling in a program.



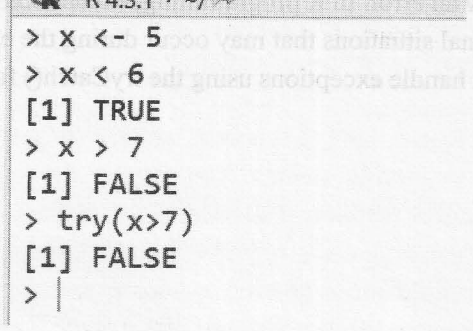
There are basically three methods to handle such conditions and errors in R programming :

- **try():** It helps us to continue with the execution of the program even when an error occurs.
- **tryCatch():** It helps to handle the conditions and control what happens based on the conditions.
- **withCallingHandlers():** It is an alternative to tryCatch() that takes care of the local handlers.

2.6.1 Try method:

An expression that may expect an exception can be written in try block,It helps us to continue with the execution of the program even when an error occurs.

Example:



```
R 4.3.1 · ~/
> x <- 5
> x < 6
[1] TRUE
> x > 7
[1] FALSE
> try(x>7)
[1] FALSE
> |
```

2.6.2 tryCatch():

Unlike other programming languages such as Java, C#, and so on, the try-catch-finally statements are used as a function in R. The main two conditions to be handled in tryCatch() are "errors" and "warnings".

The tryCatch() function is the primary tool for capturing and handling exceptions in R. It allows you to attempt a potentially error-prone operation and specify actions to take in case an exception occurs.

The basic structure of tryCatch() is as follows:

```
result <- tryCatch({
  # Code that may cause an exception
}, error = function(err) {
          # Code to handle the exception
}, finally = {
   # Code to execute after the try block, whether there's an exception or not
})
```

- The code inside the first block is the code that may generate an exception.
- The error argument is a function that specifies how to handle exceptions. This function receives the exception object err as an argument.
- The finally block is optional and contains code that executes after the try block, whether an exception occurs or not.

Examples 1:

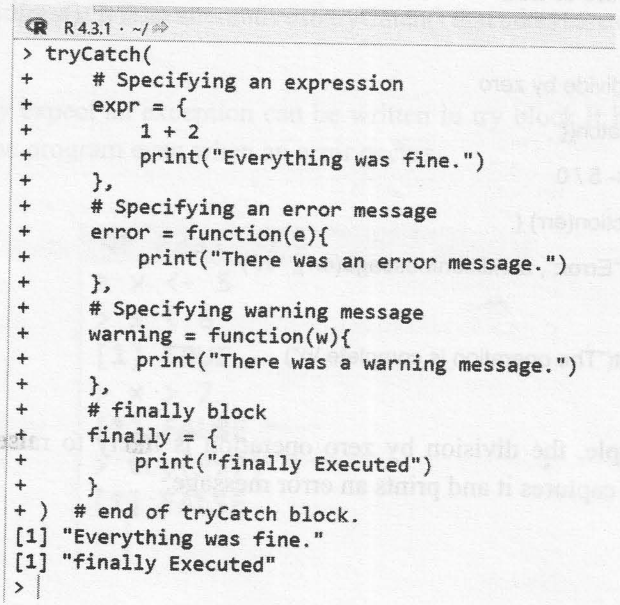
```
# Attempt to divide by zero
result <- tryCatch({
          x <- 5 / 0
}, error = function(err) {
         cat("Error:", conditionMessage(err), "\n")
}, finally = {
          cat("The operation is complete.\n")
})
```

In this example, the division by zero operation is likely to raise an exception, but the tryCatch() function captures it and prints an error message.

Examples 2:

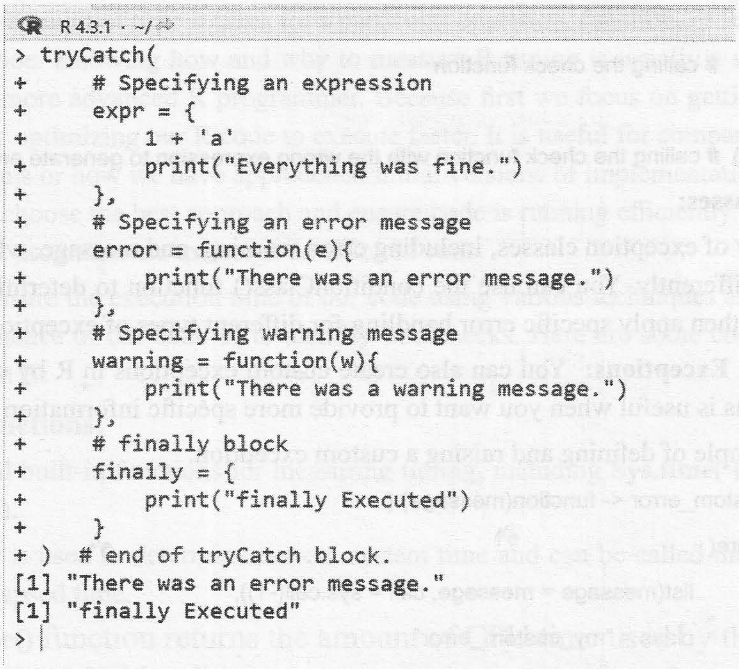
```
tryCatch(
        # Specifying an expression
        expr = {
                1 + 'a'
                print("Everything was fine.")
                },
        # Specifying an error message
        error = function(e){
                print("There was an error message.")
        },
        # Specifying warning message
        warning = function(w){
                print("There was a warning message.")
        },
        # finally block
        finally = {
                print("finally Executed")
        }
    ) # end of tryCatch block.
```

# Output after executing the code



```
R 4.3.1 · ~/
> tryCatch(
+     # Specifying an expression
+     expr = {
+         1 + 2
+         print("Everything was fine.")
+     },
+     # Specifying an error message
+     error = function(e){
+         print("There was an error message.")
+     },
+     # Specifying warning message
+     warning = function(w){
+         print("There was a warning message.")
+     },
+     # finally block
+     finally = {
+         print("finally Executed")
+     }
+ )  # end of tryCatch block.
[1] "Everything was fine."
[1] "finally Executed"
> 
```

# After executing the code with wrong expression


R 4.3.1 · ~/

```
> tryCatch(
+     # Specifying an expression
+     expr = {
+         1 + 'a'
+         print("Everything was fine.")
+     },
+     # Specifying an error message
+     error= function(e){
+         print("There was an error message.")
+     },
+     # Specifying warning message
+     warning = function(w){
+         print("There was a warning message.")
+     },
+     # finally block
+     finally = {
+         print("finally Executed")
+     }
+ ) # end of tryCatch block.
[1] "There was an error message."
[1] "finally Executed"
>
```

2.6.3 withCallingHandlers():

It is an alternative to the **tryCatch()** method that takes care of the local handlers.The only difference is that **tryCatch()** deals with existing handlers while the **withCallingHandlers()** deals with local handlers.

Example: handler is written within a function to handle local errors of the function body.

```
check <- function(expression){
            withCallingHandlers(expression,
                    warning = function(w){
                            message("warning :\n", w)
                    },
                    error = function(e){
                            message("error :\n", e)
                    },
                    finally = {
                            message("Completed Execution")
            })
    }
check({10/2})      # calling the check function
check({10/0})
check({10/'abc'}) # calling the check function with the wrong expression to generate error.
```

Exception Classes:

R has a variety of exception classes, including error, warning, and message, which can be caught and handled differently. You can use the conditionClass() function to determine the class of an exception and then apply specific error handling for different types of exceptions.

Custom Exceptions: You can also create custom exceptions in R by extending the base error class. This is useful when you want to provide more specific information about the error.

Here's an example of defining and raising a custom exception:

```
my_custom_error <- function(message) {
 structure(
          list(message = message, call = sys.call(-1)),
          class = "my_custom_error"
          )
 }

tryCatch({
       stop(my_custom_error("This is a custom error message."))
}, error = function(err) {
       cat("Custom Error Message:", conditionMessage(err), "\n")
})
```

This code defines a custom exception using the my_custom_error function and raises it using stop(). The tryCatch() block then handles the custom error.

Handling exceptions in R is essential for writing robust and error-tolerant code, especially in data analysis and scientific computing, where unexpected issues can arise. The tryCatch() function and related techniques provide the means to capture, inspect, and respond to exceptions appropriately in your R code.

2.7 Timings:

Timing in R is the amount of time it takes for a particular operation, function, or set of operations to execute in R code. Knowing how and why to measure R timing is usually a sign of moving from beginner to more advanced R programmer. Because first we focus on getting our code to work, followed by optimizing our R code to execute faster. It is useful for comparing the performance of algorithms or how we have approached initial versions of implementation. Comparing versions can help choose the best approach and ensure code is running efficiently. It's an important option in R for programmers to design an efficient code.

We can measure the execution time of our code using various techniques and functions to assess the performance of our code or to identify bottlenecks. Here are some common ways to measure timings in R:

2.7.1 Timing functions:

R provides several built-in functions for measuring timing, including **Sys.time( )**, **system.time( )**, and **proc.time( ).**

- **Sys.time( )** is used to determine current system time and can be called multiple times to calculate elapsed time.
- **system.time()** function returns the amount of CPU time used by the R process. The system.time() function allows you to measure the execution time of a specific expression or block of code. It returns an object containing information about user time, system time, and elapsed time.

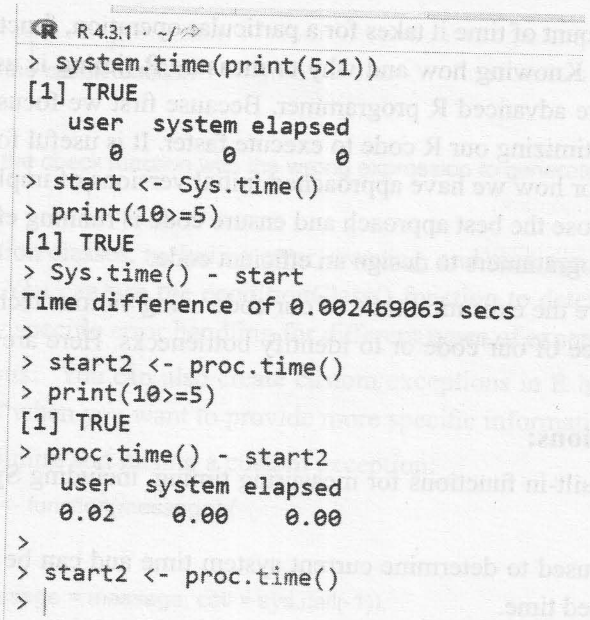
```
result <- system.time({

  # Code to measure execution time
          for (i in 1:1000000) {
           sqrt(i)
          }
})


cat("User time:", result[1], "\n")
cat("System time:", result[2], "\n")
cat("Elapsed time:", result[3], "\n")
```

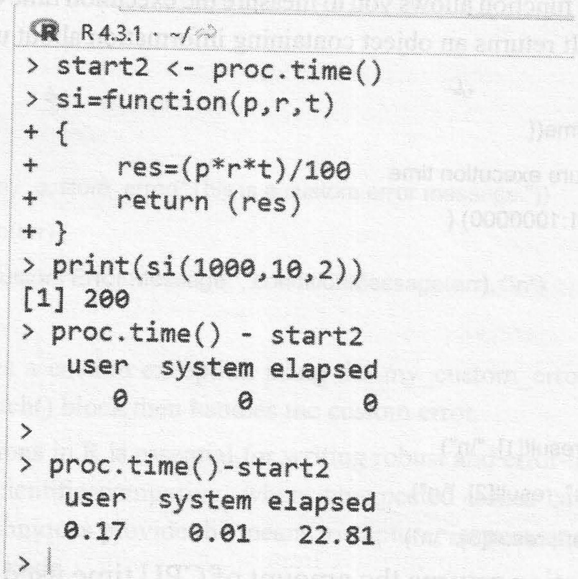
- **proc.time()** function returns the amount of CPU time used. It can be useful for measuring the time taken by a sequence of expressions or function calls.

Examples for measuring timing with all these three functions:



```
> system.time(print(5>1))
[1] TRUE
   user  system elapsed 
      0       0       0 
> start <- Sys.time()
> print(10>=5)
[1] TRUE
> Sys.time() - start
Time difference of 0.002469063 secs
> 
> start2 <- proc.time()
> print(10>=5)
[1] TRUE
> proc.time() - start2
   user  system elapsed 
   0.02    0.00    0.00 
> 
> start2 <- proc.time()
> |
```

Computing the process time for simple interest function:



```
> start2 <- proc.time()
> si=function(p,r,t)
+ {
+     res=(p*r*t)/100
+     return (res)
+ }
> print(si(1000,10,2))
[1] 200
> proc.time() - start2
   user  system elapsed 
      0       0       0 
> 
> proc.time()-start2
   user  system elapsed 
   0.17    0.01   22.81 
> |
```

As demonstrated in this example we can start the process time at the beginning of a program and subtract the start time at the end of the program.

microbenchmark Package:

The microbenchmark package is a powerful tool for measuring the execution time of small code snippets with high precision. It allows you to compare the performance of different approaches to solving a problem.

Example:

```
library(microbenchmark)

result <- microbenchmark(

 method1 = {

        # Code for method 1

 },

 method2 = {

        # Code for method 2

 },

 times = 100
)
print(result)
```

profvis Package:

The profvis package is used for profiling R code to identify performance bottlenecks and visualize code execution. It helps you understand where your code is spending the most time.

Example:

```
library(profvis)

profvis({

       # Code to profile

        for (i in 1:1000000) {
          sqrt(i)
        }
})
```

Benchmarking with the bench Package:

The bench package provides a comprehensive framework for benchmarking R code and provides detailed performance information, including memory usage.

Example:

```
library(bench)

result <- mark({
        # Code to benchmark
    for (i in 1:1000000) {
      sqrt(i)
    }
  })
  summary(result)
```

Benchmarking with the rbenchmark Package:

The rbenchmark package allows you to compare the execution time of multiple functions or code snippets.

Example:

```
library(rbenchmark)
result <- benchmark(
method1 = {
        # Code for method 1
},
method2 = {
        # Code for method 2
},
replications = 100
)
print(result)
```

Measuring execution time is essential for optimizing your code and identifying areas that might need improvement. The choice of timing measurement method depends on the level of precision and the specific requirements of your performance analysis.

2.7.2 Optimizing Timing:

By identifying bottlenecks and applying code optimization techniques, R programmers can improve the speed and efficiency of their R code. Optimizing the execution time of your code in R involves identifying and addressing performance bottlenecks. Here are some strategies and techniques to optimize the timing of your R code:

Use Vectorized Operations: R is designed to work efficiently with vectors and matrices. Instead of using loops to perform operations on individual elements, try to use vectorized functions and operations. Vectorized code is generally faster than non-vectorized code.

Example:

```
# Non-vectorized code
result <- numeric(1000)
for (i in 1:1000) {
        result[i] <- i * 2
}

# Vectorized code
result <- 1:1000 * 2
```

Avoid Global Variables: Minimize the use of global variables, which can lead to slower code execution. Instead, use function arguments to pass data to functions, which helps improve code modularity and performance.

Use Efficient Data Structures: Choose the most appropriate data structures for your data and operations. For example, use data frames for tabular data and matrices for numeric operations. Avoid unnecessary conversions between data structures.

Memory Management: Pay attention to memory usage. If your code is using a large amount of memory, consider using data.table, ff, or other memory-efficient data structures. Regularly remove unnecessary objects from memory using rm(), and use the gc() function to trigger garbage collection.

Parallel Processing: Consider using parallel processing to take advantage of multi-core processors. The parallel package and the foreach package, along with parallel backends like doParallel, can help parallelize tasks.

Optimize Functions: Profile your functions to identify performance bottlenecks. Tools like profvis, microbenchmark, and bench can help you pinpoint slow code and optimize it. Look for opportunities to reduce function calls, minimize duplicated calculations, and simplify logic.

Avoid Unnecessary Data Copies: Creating unnecessary copies of data can be time-consuming and memory-intensive. Whenever possible, modify data in-place rather than creating new copies. Use functions like data.table::set() to modify data frames without copying.

Use Efficient Package Functions: R has a vast ecosystem of packages that provide optimized functions for various tasks. Make use of these packages to take advantage of their efficient algorithms. For example, the data.table package is known for its speed in data manipulation.

Caching and Memoization: Implement caching and memoization to store and reuse intermediate results of computationally intensive calculations. This can significantly reduce redundant computations.

Profiling: Profile your code to identify performance bottlenecks using tools like Rprof, profvis, or the built-in system.time() function. Profiling helps you understand where your code spends the most time.

Preallocate Data Structures: When creating large data structures, preallocate memory to avoid dynamic resizing, which can be slow. Use functions like matrix() or numeric() to preallocate matrices and vectors.

Optimize I/O Operations: Minimize unnecessary file and data I/O operations. Reading and writing to disk can be a significant performance bottleneck. Consider using binary formats like Feather or Parquet for data storage and exchange.

Compiler: Consider using the compiler package to byte-compile your functions. This can lead to performance improvements for certain types of computations.

Use Efficient Sorting Algorithms: When sorting data, choose an appropriate sorting algorithm, depending on the size of the data and the specific sorting requirements. The base R function sort() uses a hybrid sorting algorithm that adapts to the data size.

Consider Package-Specific Optimization: Some packages, like dplyr, data.table, and sqldf, provide specific optimizations for common data manipulation tasks. Familiarize yourself with these packages and their features.

Identifying Bottlenecks with Rprof(): One of the first steps in optimizing timing in R is identifying the bottlenecks in the code. Bottlenecks are areas of the code that take up a significant amount of time and slow down the overall execution time of the script. One way to identify bottlenecks is by using the **Rprof() function**, which provides a profile of the code and identifies the functions that take up the most time.

Code Optimization Techniques: Once identified, users can apply R code optimization techniques to address the bottleneck and improve the speed and efficiency of their code. Some of these techniques include:

- **Vectorization:** vectorized operations, instead of loops, can improve the speed of R scripts.
- **Caching**: Caching frequently used data or results can reduce the amount of time spent recalculating them. Classic trick.
- **Parallelization**: Splitting up tasks and running them in parallel can improve the speed of R scripts especially on multi-core systems.
- **Memory management**: Efficient memory management, such as removing unused objects and avoiding excessive copying, can reduce the execution time of R scripts.

Optimization techniques is a different subject than ways to time code in R, but in short, by applying techniques such as those listed above, we will be able to optimize the timing of our R programs. Optimizing code in R often involves a combination of these techniques, and the approach may vary depending on the specific requirements of your project. Profiling your code to identify performance bottlenecks is a critical step in the optimization process. Additionally, keep in mind that optimization should not compromise code readability and maintainability.

2.8 Visibility:

In R programming, scope refers to the accessibility or visibility of objects (variables, functions, etc.) within different parts of your code. In R, there are two main types of variables: global variables and local variables. Variable declared outside a function is a global variable which can be accessed from any block of the program, and a variable declared within a function is a local variable which is only accessible from within the function, not from outside of the function.

Visibility refers to the accessibility of objects (variables, functions, and data) within different scopes or environments. Understanding visibility is essential for writing clean and organized code and for managing the scope of objects within your R scripts or packages. There are several concepts related to visibility in R:

Global and Local Scopes:

- **Global Scope:** Objects defined in the global environment (top-level) are accessible from any part of your R script, function, or package.
- **Local Scope:** Objects defined within a specific function or block of code are only accessible within that function or block. They have a local scope and are not visible outside of that context.

Lexical Scoping: R uses lexical scoping, which means that the scope of an object is determined by where it is defined in the source code. Inner functions can access variables from their parent functions.
Example 1:

```
x <- 10

my_function <- function() {
  y <- 5
  inner_function <- function() {
    cat("x:", x, "y:", y, "\n")
  }
  return(inner_function)
}

result_function <- my_function()
result_function()
```

In this example, inner_function has access to both x and y because it's defined within the scope of my_function.

GlobalEnv and .Environment:

- .GlobalEnv refers to the global environment where objects defined at the top level are stored. You can access global objects using the <<- operator.
- .Environment refers to the current environment in which a function or block of code is executing. You can access local objects using the <- operator.

Visibility Rules:

- By default, objects in the global environment are visible to functions unless an object with the same name is defined within the function's local scope.
- If an object with the same name is defined within a local scope, it takes precedence over the global object.

Scoping Functions: R provides functions like ls(), ls.str(), and objects() that help you list and examine objects within a specific environment.

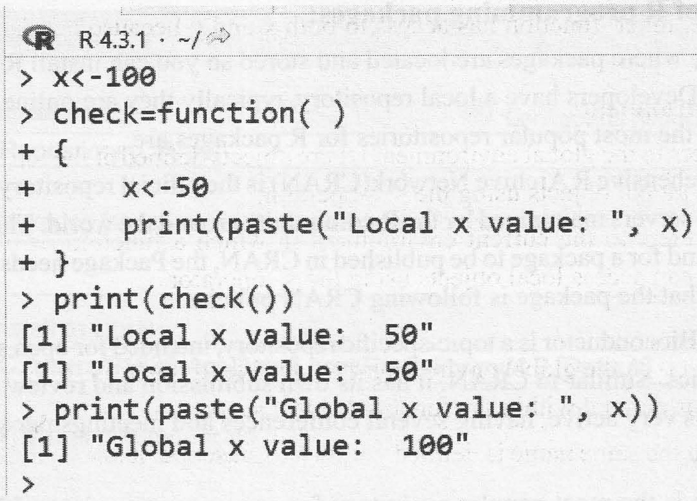
Example 2:

```
x <- 10
ls()
ls.str()
my_function <- function() {
        y <- 5
        ls()
}
my_function()
```

These functions allow you to inspect the objects visible within a given environment.

Example 2:

```
x<-100
check=function( )
{
        x<-50
        print(paste("Local x value: ", x))
}
print(check())
print(paste("Global x value: ", x))
```



```
R 4.3.1 · ~/
> x<-100
> check=function( )
+ {
+     x<-50
+     print(paste("Local x value: ", x))
+ }
> print(check())
[1] "Local x value:  50"
[1] "Local x value:  50"
> print(paste("Global x value: ", x))
[1] "Global x value:  100"
>
```

Accessing Non-Visible Objects:

In some cases, you may need to access objects that are not in your current environment. You can use the get() function to access global objects by name, even from within a function's local scope.

Example:

```
x <- 10

my_function <- function() {
        cat("x from global scope:", get("x"), "\n")
}
my_function()
```

Understanding visibility in R is crucial for writing organized and efficient code. Proper scoping of objects and functions helps prevent naming conflicts, improve code readability, and ensure that objects are accessible where they are needed.

2.9 Packages:

Packages in R language are a set of R functions, compiled code, and sample data. These are stored under a directory called "library" within the R environment. By default, R installs a group of packages during installation. Once we start the R console, only the default packages are available by default. Other packages that are already installed need to be loaded explicitly to be utilized by the R program that's getting to use them.

In R, packages are collections of functions, data sets, and documentation bundled together for specific purposes or tasks. Packages are an essential part of the R ecosystem, as they provide a way to extend R's functionality and leverage the work of other R users and developers.

2.9.1 Repositories of R programming packages:

A repository is a place where packages are located and stored so you can install R packages from it. Organizations and Developers have a local repository, typically they are online and accessible to everyone. Some of the most popular repositories for R packages are:

- **CRAN**: Comprehensive R Archive Network(CRAN) is the official repository, it is a network of FTP and web servers maintained by the R community around the world. The R community coordinates it, and for a package to be published in CRAN, the Package needs to pass several tests to ensure that the package is following CRAN policies.
- **Bioconductor**: Bioconductor is a topic-specific repository, intended for open source software for bioinformatics. Similar to CRAN, it has its own submission and review processes, and its community is very active, having several conferences and meetings per year in order to maintain quality.
- **Github**: Github is the most popular repository for open-source projects. It's popular as it comes from the unlimited space for open source, the integration with git, a version control software, and its ease to share and collaborate with others.

2.9.2 Installing R Packages From CRAN:

For installing R Package from CRAN we need the name of the package and use the following command:

```
install.packages("package name")
install.packages(c("vioplot", "MASS"))
```

2.9.3 Update, Remove and Check Installed Packages in R:

To check what packages are installed on your computer, type this command:

```
installed.packages()
```

To update all the packages, type this command:

```
update.packages()
```

To update a specific package, type this command:

```
install.packages("PACKAGE NAME")
```

2.9.4 Installing Packages Using RStudio UI:

In R Studio goto **Tools -> Install Package**, and there we will get a pop-up window to type the package you want to install, Under Packages, type, and search Package which we want to install and then click on install button.

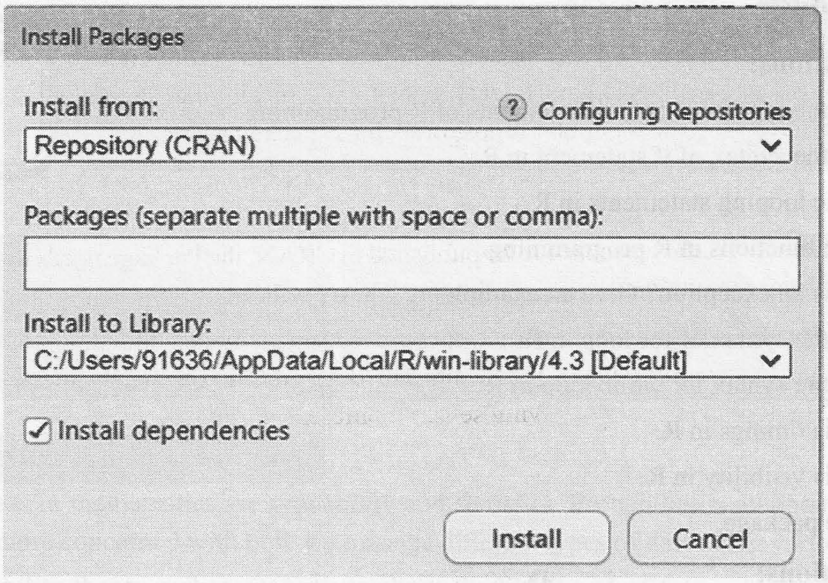


Figure 2.3: Tool->Install packages in RStudio

2.9.5 How to Load Packages in R Programming Language:

When a R package is installed, we are ready to use its functionalities. If we just need a sporadic use of a few functions or data inside a package we can access them with the following notation.

```
# Load a package using the library function
        library(dplyr)
# Load a package using the require function
        require(dplyr)
```

Both functions attempt to load the specified package, but there is a subtle difference between the two: **library()** returns an error if the package is not found or cannot be loaded, whereas **require()** returns a warning and sets the value of the package variable to FALSE.

2.9.6 Difference Between a Package and a Library:

library(): It is the command used to load a package, and it refers to the place where the package is contained, usually a folder on our computer.

Package: It is a collection of functions bundled conveniently. The package is an appropriate way to organize our own work and share it with others.

2.10 Exercise:

Short Questions:

1. Write the input and output statements of R programming.
2. Write the syntax of if statement in R.
3. List the looping statements in R.
4. Define functions in R programming.
5. What is an exception? Give an example.
6. Write the syntax of for loop in R.
7. Write the syntax for while loop in R.
8. What is timings in R.
9. What is visibility in R.
10. Define package.

Long Questions:

1. Explain the concept of reading and writing files in R.
2. Explain the different input and output statements in R.
3. Explain the different conditional statements in R.
4. Explain the different looping statements in R.
5. Differentiate between while loop and repeat loop.
6. Explain break and next statement in R with an example of each.
7. Differentiate between try( ) and trycatch( ) statements.
8. Create an R function for computing simple interest.
9. What is exception handling? Write an R code to handle exceptions.
10. Explain the package in R with a suitable example.

*****