|  |  |  |
| --- | --- | --- |
|  | **MODULE 2**  **R PROGRAMMING STRUCTURES** |  |

**R Programming Structures:**

R programming offers various control structures and programming constructs to help you manage the flow of your code, make decisions, and repeat tasks. These structures are essential for writing efficient and structured R code. Here are the primary programming structures in R:

1. **Sequential Execution**:
   * R executes statements sequentially, one after the other, by default. This is the basic flow of any R script or program.

x <- 5

y <- x + 3

1. **Conditional Statements (if-else)**:
   * Conditional statements allow you to execute different code blocks based on specified conditions. The **if** statement is used for single conditions, while **if-else** is used for branching between two or more conditions.

if (x > 0) {

print("x is positive")

} else {

print("x is non-positive")

}

") }

1. **Switch Statements (switch)**:
   * The **switch** function allows you to select one of several code blocks to execute based on the value of an expression.

fruit <- "apple"

switch(

fruit,

"apple" = print("It's an apple."),

"banana" = print("It's a banana."),

print("It's something else.")

)

1. **Loops**:

Loops are used for repetitive tasks. R supports several types of loops, including **for**, **while**, and **repeat** loops.

* + **For Loop**:

for (i in 1:5) {

print(i)

}

(i) }

* + **While Loop**:

i <- 1

while (i <= 5) {

print(i)

i <- i + 1

}

( **Repeat Loop**:

i <- 1

repeat {

print(i)

i <- i + 1

if (i > 5) break

}

1. **Control Statements (break and next)**:

for (i in 1:10) {

if (i == 5) break

if (i %% 2 == 0) next

print(i)

}

* + **break** is used to exit a loop prematurely, and **next** is used to skip the current iteration and continue to the next iteration.

(i) }

1. **Function Definitions (functions)**:

my\_function <- function(x, y) {

result <- x + y

return(result)

}

* + Functions allow you to define reusable blocks of code. You can create your own functions or use built-in functions in R.

my\_function) }

1. **Error Handling (try-catch)**:
   * The **tryCatch** function allows you to handle errors gracefully by specifying what to do in case of an error. This is important for robust and error-tolerant code.
   * result <- tryCatch({
   * # Code that might throw an error
   * }, error = function(e) {
   * # Code to handle the error
   * print(paste("An error occurred:", e))
   * return(NA)
   * })

})

1. **Apply Family Functions**:

matrix\_data <- matrix(1:12, nrow = 3)

result <- apply(matrix\_data, MARGIN = 2, FUN = sum)

R provides a set of functions like **apply**, **lapply**, **sapply**, and **mapply** for applying a function to elements of data structures like matrices, lists, and data frame

Control Statements in RTop of Form

**Control statements** are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to make a decision after assessing the variable. In this article, we’ll discuss all the control statements with the examples.

In [R programming](https://www.geeksforgeeks.org/introduction-to-r-programming-language/), there are 8 types of control statements as follows:

* [if condition](https://www.geeksforgeeks.org/control-statements-in-r-programming/#ifstatement)
* [if-else condition](https://www.geeksforgeeks.org/control-statements-in-r-programming/#if-else)
* [for loop](https://www.geeksforgeeks.org/control-statements-in-r-programming/#forloop)
* [nested loops](https://www.geeksforgeeks.org/control-statements-in-r-programming/#nestedloop)
* [while loop](https://www.geeksforgeeks.org/control-statements-in-r-programming/#whileloop)
* [repeat and break statement](https://www.geeksforgeeks.org/control-statements-in-r-programming/#repeatandbreak)
* [return statement](https://www.geeksforgeeks.org/control-statements-in-r-programming/#return)
* [next statement](https://www.geeksforgeeks.org/control-statements-in-r-programming/#next)

#### if condition

This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

**Syntax:**

if(expression){

statements

....

....

}

**Example:**

|  |
| --- |
| x <- 100    if(x > 10){  print(paste(x, "is greater than 10"))  } |

**Output:**

[1] "100 is greater than 10"

#### if-else condition

It is similar to **if** condition but when the test expression in if condition fails, then statements in **else** condition are executed.

**Syntax:**

if(expression){

statements

....

....

}

else{

statements

....

....

}

**Example:**

|  |
| --- |
| x <- 5    # Check value is less than or greater than 10  if(x > 10){    print(paste(x, "is greater than 10"))  }else{    print(paste(x, "is less than 10"))  } |

**Output:**

[1] "5 is less than 10"

#### ****for loop****

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

**Syntax:**

for(value in vector){

statements

....

....

}

**Example:**

|  |
| --- |
| x <- letters[4:10]  for(i in x){    print(i)  } |

**Output:**

[1] "d"

[1] "e"

[1] "f"

[1] "g"

[1] "h"

[1] "i"

[1] "j"

#### 

#### ****Nested loops****

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

**Example:**

|  |
| --- |
| # Defining matrix  m <- matrix(2:15, 2)    for (r in seq(nrow(m))) {    for (c in seq(ncol(m))) {      print(m[r, c])    }  } |

**Output:**

[1] 2

[1] 4

[1] 6

[1] 8

[1] 10

[1] 12

[1] 14

[1] 3

[1] 5

[1] 7

[1] 9

[1] 11

[1] 13

[1] 15

#### while loop

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

**Syntax:**

while(expression){

statement

....

....

}

**Example:**

|  |
| --- |
| x = 1    # Print 1 to 5  while(x <= 5){    print(x)    x = x + 1  } |

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

#### repeat loop and break statement

**repeat** is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, break statement is used to exit from the loop. **break**statement can be used in any type of loop to exit from the loop.

**Syntax:**

repeat {

statements

....

....

if(expression) {

break

}

}

**Example:**

|  |
| --- |
| x = 1    # Print 1 to 5  repeat{    print(x)    x = x + 1    if(x > 5){      break     }  } |

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

#### return statement

**return**statement is used to return the result of an executed function and returns control to the calling function.

**Syntax:**

return(expression)

**Example:**

|  |
| --- |
| # Checks value is either positive, negative or zero  func <- function(x){    if(x > 0){      return("Positive")    }else if(x < 0){      return("Negative")    }else{      return("Zero")    }  }    func(1)  func(0)  func(-1) |

**Output:**

[1] "Positive"

[1] "Zero"

[1] "Negative"

#### next statement

**next**statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

**Example:**

|  |
| --- |
| # Defining vector  x <- 1:10    # Print even numbers  for(i in x){    if(i%%2 != 0){      next #Jumps to next loop    }    print(i)  } |

**Output:**

[1] 2

[1] 4

[1] 6

[1] 8

[1] 10

**Example: Check Leap Year**

# Program to check if the input year is a leap year or not

year = as.integer(readline(prompt="Enter a year: "))

if((year %% 4) == 0) {

if((year %% 100) == 0) {

if((year %% 400) == 0) {

print(paste(year,"is a leap year"))

} else {

print(paste(year,"is not a leap year"))

}

} else {

print(paste(year,"is a leap year"))

}

} else {

print(paste(year,"is not a leap year"))

}

**Example: Find the factorial of a number**

# take input from the user

num = as.integer(readline(prompt="Enter a number: "))

factorial = 1

# check is the number is negative, positive or zero

if(num < 0) {

print("Sorry, factorial does not exist for negative numbers")

} else if(num == 0) {

print("The factorial of 0 is 1")

} else {

for(i in 1:num) {

factorial = factorial \* i

}

print(paste("The factorial of", num ,"is",factorial))

}

**Sum of digits**

* n<-readline(prompt="please enter any integer value: ")
* please enter any integer value: 12367906.
* n <- as.integer(n)
* sum<-0.
* while(n!=0){
* sumsum=sum+(n%%10)
* n=as.integer(n/10)
* }

Example: How to check a number is a palindrome or not in R

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10  11  12  13  14  15  16  17  18  19  20  21  22 | n = as.integer(readline(prompt = "Enter a number :"))        rev = 0      num = n        while (n > 0) {        r = n %% 10        rev = rev \* 10 + r       n = n %/% 10      }        if (rev == num)      {        print(paste("Number is palindrome :", rev))      }      else{        print(paste("Number is not palindrome :", rev))      }  }    **Example: Check Prime Number**  # Program to check if the input number is prime or not  # take input from the user  num = as.integer(readline(prompt="Enter a number: "))  flag = 0  # prime numbers are greater than 1  if(num > 1) {  # check for factors  flag = 1  for(i in 2:(num-1)) {  if ((num %% i) == 0) {  flag = 0  break  }  }  }  if(num == 2) flag = 1  if(flag == 1) {  print(paste(num,"is a prime number"))  } else {  print(paste(num,"is not a prime number"))  } Looping over Objects in R Programming One of the biggest issues with the “for” loop is its memory consumption and its slowness in executing a repetitive task.  And when it comes to dealing with large data set and iterating over it, for loop is not advised.  [R](https://www.geeksforgeeks.org/introduction-to-r-programming-language/) provides many alternatives to be applied to vectors for looping operations that are pretty useful when working interactively on a command line. **apply()**   * **lapply()** * **sapply()** * **tapply()** * **mapply()**  |  |  | | --- | --- | | **apply()** | Applies a function over the margins of an array or matrix | | **lapply()** | Apply a function over a list or a vector | | **sapply()** | Same as lapply() but with simplified results | | **tapply()** | Apply a function over a ragged array | | **mapply()** | Multivariate version of lapply() |  * **apply()**: This function applies a given function over the margins of a given array.   *apply(array, margins, function, …)*  *array = list of elements margins = dimension of the array along which the function needs to be applied function = the operation which you want to perform*  **Example:**   |  | | --- | | # R program to illustrate apply() function    # Creating a matrix  A = matrix(1:9, 3, 3)  print(A)  # Applying apply() over row of matrix  # Here margin 1 is for row  r = apply(A, 1, sum)  print(r)  # Applying apply() over column of matrix  # Here margin 2 is for column  c = apply(A, 2, sum)  print(c) |   **Output:**  [, 1] [, 2] [, 3]  [1, ] 1 4 7  [2, ] 2 5 8  [3, ] 3 6 9  [1] 12 15 18  [1] 6 15 24   * **lapply():** This function is used to apply a function over a list. It always returns a list of the same length as the input list.   *lapply(list, function, …)*  *list = Created list function = the operation which you want to perform*  **Example:**   |  | | --- | | # R program to illustrate lapply() function Creating a matrix  A = matrix(1:9, 3, 3)    # Creating another matrix  B = matrix(10:18, 3, 3)    # Creating a list  myList = list(A, B)    # applying lapply()  determinant = lapply(myList, det)  print(determinant) |   **Output:**  [[1]]  [1] 0  [[2]]  [1] 5.329071e-15   * **sapply():** This function is used to simplify the result of lapply(), if possible. Unlike lapply(), the result is not always a list. The output varies in the following ways:-   + If output is a list containing elements having length 1, then a vector is returned.   + If output is a list where all the elements are vectors of same length(>1), then a matrix is returned.   + If output contains elements which cannot be simplified or elements of different types, a list is returned.   *sapply(list, function, …)*  *list = Created list function = the operation which you want to perform*  **Example:**   |  | | --- | | # R program to illustrate sapply() function.Creating a list  A = list(a = 1:5, b = 6:10)    # applying sapply()  means = sapply(A, mean)  print(means) |   **Output:**  a b  3 8  A vector is returned since the output had a list with elements of length 1.   * **tapply()**: This function is used to apply a function over subset of vectors given by a combination of factors.   *tapply(vector, factor, function, …)*  *vector = Created vector factor = Created factor function = the operation which you want to perform*  **Example:**   |  | | --- | | # R program to illustrate tapply() function    # Creating a factor  Id = c(1, 1, 1, 1, 2, 2, 2, 3, 3)    # Creating a vector  val = c(1, 2, 3, 4, 5, 6, 7, 8, 9)    # applying tapply()  result = tapply(val, Id, sum)  print(result) |   **Output:**  1 2 3  10 18 17  https://media.geeksforgeeks.org/wp-content/uploads/20200502005931/f120-300x287.png   * **mapply()**: It’s a multivariate version of lapply(). This function can be applied over several list simultaneously.   *mapply(function, list1, list2, …)*  *function = the operation which you want to perform*  list1, list2 = Created lists  **Example:**   |  | | --- | | # R program to illustrate mapply() function.Creating a list  A = list(c(1, 2, 3, 4))    # Creating another list  B = list(c(2, 5, 1, 6))  # Applying mapply()  result = mapply(sum, A, B)  print(result) |   **Output:**  [1] 24  **Arithmetic and Boolean operators and values in r programming:**  **Operators** are the symbols directing the compiler to perform various kinds of operations between the operands. Operators simulate the various mathematical, logical, and decision operations performed on a set of Complex Numbers, Integers, and Numericals as input operands. R Operators R supports majorly four kinds of binary operators between a set of operands. types of **operators in**[**R Programming language**](https://www.geeksforgeeks.org/r-programming-language-introduction/)and their usage. Types of the operator in R language  * [Arithmetic Operators](https://www.geeksforgeeks.org/r-operators/#Arithmetic%20Operators) * [Logical Operators](https://www.geeksforgeeks.org/r-operators/#Logical%20Operators) * [Relational Operators](https://www.geeksforgeeks.org/r-operators/#Relational%20Operators) * [Assignment Operators](https://www.geeksforgeeks.org/r-operators/#Assignment%20Operators) * [Miscellaneous Operator](https://www.geeksforgeeks.org/r-operators/#Miscellaneous%20Operator)  ****Arithmetic Operators**** Arithmetic operations in R simulate various math operations, like addition, subtraction, multiplication, division, and modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The R operators are performed element-wise at the corresponding positions of the vectors. ****Addition operator (+)**** The values at the corresponding positions of both operands are added. Consider the following R operator snippet to add two vectors:   |  | | --- | | a <- c (1, 0.1)  b <- c (2.33, 4)  print (a+b) |   **Output : 3.33 4.10** ****Subtraction Operator (-)**** The second operand values are subtracted from the first. Consider the following R operator snippet to subtract two variables:   |  | | --- | | a <- 6  b <- 8.4  print (a-b) |   **Output : -2.4** ****Multiplication Operator (\*)**** The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the ‘\*’ operator.   |  | | --- | | B= c(4,4)  C= c(5,5)  print (B\*C) |   **Output : 20 20** ****Division Operator (/)**** The first operand is divided by the second operand with the use of the ‘/’ operator   |  | | --- | | a <- 10  b <- 5  print (a/b) |   **Output : 2** ****Power Operator (^)**** The first operand is raised to the power of the second operand.   |  | | --- | | a <- 4  b <- 5  print(a^b) |   **Output : 1024** ****Modulo Operator (%%)**** The remainder of the first operand divided by the second operand is returned.   |  | | --- | | list1<- c(2, 22)  list2<-c(2,4)  print(list1 %% list2) |   **Output : 0 2**  The following R code illustrates the usage of all Arithmetic R operators.   |  | | --- | | # R program to illustrate the use of Arithmetic operators  vec1 <- c(0, 2)  vec2 <- c(2, 3)    # Performing operations on Operands  cat ("Addition of vectors :", vec1 + vec2, "\n")  cat ("Subtraction of vectors :", vec1 - vec2, "\n")  cat ("Multiplication of vectors :", vec1 \* vec2, "\n")  cat ("Division of vectors :", vec1 / vec2, "\n")  cat ("Modulo of vectors :", vec1 %% vec2, "\n")  cat ("Power operator :", vec1 ^ vec2) |   **Output**  Addition of vectors : 2 5  Subtraction of vectors : -2 -1  Multiplication of vectors : 0 6  Division of vectors : 0 0.6666667  Modulo of vectors : 0 2  Power operator : 0 8 ****Logical Operators**** Logical operations in R simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non-zero integer value is considered as a TRUE value, be it a complex or real number. ****Element-wise Logical AND operator (&)**** Returns True if both the operands are True.   |  | | --- | | list1 <- c(TRUE, 0.1)  list2 <- c(0,4+3i)  print(list1 & list2) |   **Output : FALSE TRUE**  Any non zero integer value is considered as a TRUE value, be it complex or real number. ****Element-wise Logical OR operator (|)**** Returns True if either of the operands is True.   |  | | --- | | list1 <- c(TRUE, 0.1)  list2 <- c(0,4+3i)  print(list1|list2) |   **Output : TRUE TRUE** ****NOT operator (!)**** A unary operator that negates the status of the elements of the operand.   |  | | --- | | list1 <- c(0,FALSE)  print(!list1) |   **Output : TRUE TRUE** ****Logical AND operator (&&)**** Returns True if both the first elements of the operands are True.   |  | | --- | | list1 <- c(TRUE, 0.1)  list2 <- c(0,4+3i)  print(list1 && list2) |   **Output : FALSE**  Compares just the first elements of both the lists. ****Logical OR operator (||)**** Returns True if either of the first elements of the operands is True.   |  | | --- | | list1 <- c(TRUE, 0.1)  list2 <- c(0,4+3i)  print(list1||list2) |   **Output : TRUE**  The following R code illustrates the usage of all Logical Operators in R:   |  | | --- | | R program to illustrate the use of Logical operators  vec1 <- c(0,2)  vec2 <- c(TRUE,FALSE)    # Performing operations on Operands  cat ("Element wise AND :", vec1 & vec2, "\n")  cat ("Element wise OR :", vec1 | vec2, "\n")  cat ("Logical AND :", vec1 && vec2, "\n")  cat ("Logical OR :", vec1 || vec2, "\n")  cat ("Negation :", !vec1) |   **Output**  Element wise AND : FALSE FALSE  Element wise OR : TRUE TRUE  Logical AND : FALSE  Logical OR : TRUE  Negation : TRUE FALSE ****Relational Operators**** The relational operators in R carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE. ****Less than (<)**** Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.   |  | | --- | | list1 <- c(TRUE, 0.1,"apple")  list2 <- c(0,0.1,"bat")  print(list1<list2) |   **Output : FALSE FALSE TRUE** ****Less than equal to (<=)**** Returns TRUE if the corresponding element of the first operand is less than or equal to that of the second operand. Else returns FALSE.   |  | | --- | | list1 <- c(TRUE, 0.1, "apple")  list2 <- c(TRUE, 0.1, "bat")    # Convert lists to character strings  list1\_char <- as.character(list1)  list2\_char <- as.character(list2)    # Compare character strings  print(list1\_char <= list2\_char) |   **Output : TRUE TRUE TRUE** ****Greater than (>)**** Returns TRUE if the corresponding element of the first operand is greater than that of the second operand. Else returns FALSE.   |  | | --- | | list1 <- c(TRUE, 0.1, "apple")  list2 <- c(TRUE, 0.1, "bat")  print(list1\_char > list2\_char) |   **Output : FALSE FALSE FALSE** ****Greater than equal to (>=)**** Returns TRUE if the corresponding element of the first operand is greater or equal to that of the second operand. Else returns FALSE.   |  | | --- | | list1 <- c(TRUE, 0.1, "apple")  list2 <- c(TRUE, 0.1, "bat"  print(list1\_char >= list2\_char) |   **Output : TRUE TRUE FALSE** ****Not equal to (!=)**** Returns TRUE if the corresponding element of the first operand is not equal to the second operand. Else returns FALSE.   |  | | --- | | list1 <- c(TRUE, 0.1,'apple')  list2 <- c(0,0.1,"bat")  print(list1!=list2) |   **Output : TRUE FALSE TRUE**  The following R code illustrates the usage of all Relational Operators in R:   |  | | --- | | # R program to illustrate  # the use of Relational operators  vec1 <- c(0, 2)  vec2 <- c(2, 3)  # Performing operations on Operands  cat ("Vector1 less than Vector2 :", vec1 < vec2, "\n")  cat ("Vector1 less than equal to Vector2 :", vec1 <= vec2, "\n")  cat ("Vector1 greater than Vector2 :", vec1 > vec2, "\n")  cat ("Vector1 greater than equal to Vector2 :", vec1 >= vec2, "\n")  cat ("Vector1 not equal to Vector2 :", vec1 != vec2, "\n") |   **Output**  Vector1 less than Vector2 : TRUE TRUE  Vector1 less than equal to Vector2 : TRUE TRUE  Vector1 greater than Vector2 : FALSE FALSE  Vector1 greater than equal to Vector2 : FALSE FALSE  Vector1 not equal to Vector2 : TRUE TRUE ****Assignment Operators**** Assignment operators in R are used to assigning values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stored by the assigned variable names. There are two kinds of assignment operators: Left and Right ****Left Assignment (<- or <<- or =)**** Assigns a value to a vector.   |  | | --- | | vec1 = c("ab", TRUE)  print (vec1) |   **Output : "ab" "TRUE"** ****Right Assignment (-> or ->>)**** Assigns value to a vector.   |  | | --- | | c("ab", TRUE) ->> vec1  print (vec1) |   **Output : "ab" "TRUE"**  The following R code illustrates the usage of all Relational Operators in R:   |  | | --- | | # R program to illustrate the use of Assignment operators  vec1 <- c(2:5)  c(2:5) ->> vec2  vec3 <<- c(2:5)  vec4 = c(2:5)  c(2:5) -> vec5    # Performing operations on Operands  cat ("vector 1 :", vec1, "\n")  cat("vector 2 :", vec2, "\n")  cat ("vector 3 :", vec3, "\n")  cat("vector 4 :", vec4, "\n")  cat("vector 5 :", vec5) |   **Output**  vector 1 : 2 3 4 5  vector 2 : 2 3 4 5  vector 3 : 2 3 4 5  vector 4 : 2 3 4 5  vector 5 : 2 3 4 5 ****Miscellaneous Operators**** These are the mixed operators in R that simulate the printing of sequences and assignment of vectors, either left or right-handed. ****%in% Operator**** Checks if an element belongs to a list and returns a boolean value TRUE if the value is present  else FALSE.   |  | | --- | | val <- 0.1  list1 <- c(TRUE, 0.1,"apple")  print (val %in% list1) |   **Output : TRUE**  Checks for the value 0.1 in the specified list. It exists, therefore, prints TRUE. ****%\*% Operator**** This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of the first matrix must be equal to the number of rows of the second matrix. Multiplication of the matrix A with its transpose, B, produces a square matrix.    |  | | --- | | mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)          print (mat)          print( t(mat))          pro = mat %\*% t(mat)          print(pro) |   **Input :**  **Output :[,1] [,2] [,3] #original matrix of order 2x3**  **[1,] 1 3 5**  **[2,] 2 4 6**  **[,1] [,2] #transposed matrix of order 3x2**  **[1,] 1 2**  **[2,] 3 4**  **[3,] 5 6**  **[,1] [,2] #product matrix of order 2x2**  **[1,] 35 44**  **[2,] 44 56**  The following R code illustrates the usage of all Miscellaneous Operators in R:   |  | | --- | | # R program to illustrate the use of Miscellaneous operators  mat <- matrix (1:4, nrow = 1, ncol = 4)  print("Matrix elements using : ")  print(mat)    product = mat %\*% t(mat)  print("Product of matrices")  print(product,)  cat ("does 1 exist in prod matrix :", "1" %in% product) |   **Output**  [1] "Matrix elements using : "  [,1] [,2] [,3] [,4]  [1,] 1 2 3 4  [1] "Product of matrices"  [,1]  [1,] 30  does 1 exist in prod matrix : FALSE  **FUNCTION**  A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.  In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.  The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects. Definition An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows −  function\_name <- function(arg\_1, arg\_2, ...) {  Function body  } Function Components The different parts of a function are −   * **Function Name** − This is the actual name of the function. It is stored in R environment as an object with this name. * **Arguments** − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values. * **Function Body** − The function body contains a collection of statements that defines what the function does. * **Return Value** − The return value of a function is the last expression in the function body to be evaluated.   R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions. Built-in Function Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs. You can refer [**most widely used R functions.**](https://cran.r-project.org/doc/contrib/Short-refcard.pdf)  # Create a sequence of numbers from 32 to 44.  print(seq(32,44))  # Find mean of numbers from 25 to 82.  print(mean(25:82))  # Find sum of numbers frm 41 to 68.  print(sum(41:68))  When we execute the above code, it produces the following result −  [1] 32 33 34 35 36 37 38 39 40 41 42 43 44  [1] 53.5  [1] 1526 User-defined Function:user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used. # Create a function to print squares of numbers in sequence.  new.function <- function(a) {  for(i in 1:a) {  b <- i^2  print(b)  }  } Calling a Function # Create a function to print squares of numbers in sequence.  new.function <- function(a) {  for(i in 1:a) {  b <- i^2  print(b)  }  }  # Call the function new.function supplying 6 as an argument.  new.function(6)  When we execute the above code, it produces the following result −  [1] 1  [1] 4  [1] 9  [1] 16  [1] 25  [1] 36 Calling a Function without an Argument # Create a function without an argument.  new.function <- function() {  for(i in 1:5) {  print(i^2)  }  }  **# Call the function without supplying an argument.**  new.function()  When we execute the above code, it produces the following result −  [1] 1  [1] 4  [1] 9  [1] 16  [1] 25 Calling a Function with Argument Values (by position and by name) The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.  # Create a function with arguments.  new.function <- function(a,b,c) {  result <- a \* b + c  print(result)  }  # Call the function by position of arguments.  new.function(5,3,11)  # Call the function by names of the arguments.  new.function(a = 11, b = 5, c = 3)  When we execute the above code, it produces the following result −  [1] 26  [1] 58 Calling a Function with Default Argument: We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.  # Create a function with arguments.  new.function <- function(a = 3, b = 6) {  result <- a \* b  print(result)  }  # Call the function without giving any argument.  new.function()  # Call the function with giving new values of the argument.  new.function(9,5)  When we execute the above code, it produces the following result −  [1] 18  [1] 45 Lazy Evaluation of Function Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.  # Create a function with arguments.  new.function <- function(a, b) {  print(a^2)  print(a)  print(b)  }  # Evaluate the function without supplying one of the arguments.  new.function(6)  When we execute the above code, it produces the following result −  [1] 36  [1] 6  Error in print(b) : argument "b" is missing, with no default  **Default Values for Argument**:  A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden. Function Default Arguments: In R, function can have default values of it's parameters. It means while defining a function, it's parameters can be set values. It makes function more generic. For example   1. function\_name ← function(a, b= 10){ 2. print(a+b); 3. }   # Calling function   1. function\_name(10)   **Output:**  20  # Calling function   1. function\_name(10,20)   **Output:**  30  First time, function is called with single argument and default value is used for second argument. Function called successfully and produce a result.  Second time, function is called with two arguments, default value of second argument is overridden. Return Value from R Function **Method 1: R function with return value**  In this scenario, we will use the return statement to return some value  **Syntax:**  *function\_name <- function(parameters) {*  *statements*  *return(value)*  *}*  *function\_name(values)*  Where,   * function\_name is the name of the function * parameters are the values that are passed as arguments * return() is used to return a value * function\_name(values) is used to pass values to the parameters   **Example:**R program to perform addition operation with return value   |  | | --- | | # define addition function perform addition operation on two values  addition= function(val1,val2) {      # add    add=val1+val2      # return the result    return(add)    }    # pass the values to the function  addition(10,20) |   **Output:**  [1] 30  **Method 2: R function without using return:**  Here without using return function we will return a value. For this just passing the name of the variable that stores the value to returned works.  **Syntax:**  *function\_name <- function(parameters) {*  *statements*  *value*  *}*  *function\_name(values)*  where,   * function\_name is the name of the function * parameters are the values that are passed as arguments * value is the return value * function\_name(values) is used to pass values to the parameters   **Example:**R program to perform addition operation without using return function   |  | | --- | | # define addition function  # perform addition operation on two values  addition= function(val1,val2) {    # add    add=val1+val2      # return the result with out using return    add  }    # pass the values to the function  addition(10,20) |   **Output:**  [1] 30  **Method 3: R function to return multiple values as a list**  In this scenario, we will use the list() function in the return statement to return multiple values.  **Syntax:**  *function\_name <- function(parameters) {*  *statements*  *return(list(value1,value2,.,value n)*  *}*  *function\_name(values)*  where,   * function\_name is the name of the function * parameters are the values that are passed as arguments * return() function takes list of values as input * function\_name(values) is used to pass values to the parameters   **Example:**R program to perform arithmetic operations and return those value   |  | | --- | | # define arithmetic function  # perform arithmetic operations on two values  arithmetic = function(val1,val2) {      # add    add=val1+val2      # subtract    sub=val1-val2      # multiply    mul=val1\*val2      # divide    div=val2/val1      # return the result as a list    return(list(add,sub,mul,div))  }    # pass the values to the function  arithmetic(10,20) |   **Output:**  [[1]]  [1] 30  [[2]]  [1] -10  [[3]]  [1] 200  [[4]]  [1] 2 R Return Value from Function In this article, you will learn to return a value from a function in R. You'll also learn to use functions without the return function.  Many times, we will require our [functions](https://www.datamentor.io/r-programming/function) to do some processing and return the result. This is accomplished with the return() function in R.  Syntax of return()  The syntax of the return() function is:  return(expression)  Here, expression is evaluated, and its value is stored as the result of the function.  The value specified in the return() statement is passed as the output of the function. Example of return() Let us look at an example which will return whether a given number is positive, negative or zero.  check <- function(x) {  if (x > 0) {  result <- "Positive"  }  else if (x < 0) {  result <- "Negative"  }  else {  result <- "Zero"  }  return(result)  }  check(1)  check(-10)  check(0)  **Output**  [1] "Positive"  [1] "Negative"  [1] "Zero"  Functions without return()  If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically in R.  For example, the following is equivalent to the above function.  check <- function(x) {  if (x > 0) {  result <- "Positive"  }  else if (x < 0) {  result <- "Negative"  }  else {  result <- "Zero"  }  result  }  We generally use explicit return() functions to return a value immediately from a function.  If it is not the last statement of the function, it will prematurely end the function bringing the control to the place from which it was called.  check <- function(x) {  if (x>0) {  return("Positive")  }  else if (x<0) {  return("Negative")  }  else {  return("Zero")  }  }  In the above example, if x > 0, the function immediately returns "Positive" without evaluating the rest of the body.  **Multiple Returns:**  The return() function can return only a single object. If we want to return multiple values in R, we can use a [list](https://www.datamentor.io/r-programming/list) (or other objects) and return it.  Following is an example:  multi\_return <- function() {  my\_list <- list("color" = "red", "size" = 20, "shape" = "round")  return(my\_list)  }  # call function multi\_return() and assign the result to variable a  a <- multi\_return()  a$color  a$size  a$shape  **Output**  [1] "red"  [1] 20  [1]"round"  Here, the function multi\_return() creates a list object my\_list, assigns some values to its elements, and then returns the entire list as the output of the function.  The list contains three elements: "color" with the value "red", "size" with the value **20**, and "shape" with the value "round".  The multi\_return() function is called and its result is assigned to the variable a.  Here,   * a$color - accesses the value of the "color" element * a$size - accesses the value of the "size" element * a$shape - accesses the value of the "shape" element   **Deciding Whether to explicitly call return in R**  **Return**is a primitive function that is used to clearly describe the leaves of a code where, the routine should end, jump out of the function and return a value.  There is no danger in using return, and it comes down to the strategy and programming style of the programmer whether or not to use it.  The return function when used with the last value as an argument needs one extra call than not using a return.   According to Hadley Wickham-Chief Scientist at RStudio Tthe speed of return is really the last thing you should be worrying about.  Core R programmers use both of these approaches ie. with and without explicit return() as it is possible to find in sources of 'base' functions.  The return() is often used without arguments returning NULL in cases to conditionally stop the function. It can’t be said if it is better or not as a standard user or analyst using R cannot see the real difference.  **Examples:**  *(function() {1;2;3;4})() #without return()*  *[1] 4*  *(function() {1;2;3;4;return(10)})() #with return value*  *[1] 10*  *(function() {1;2;return();3;4})() #return used to stop the flow*  *NULL*  **Returning Complex Objects in R :**  In R, you can return complex objects, such as lists, data frames, or custom-defined objects, from functions just like you would return simpler objects like numbers or characters. Complex objects are often returned when you need to encapsulate multiple pieces of data or different types of data together. Here are some examples of how to return complex objects in R:   1. Returning a List: You can return a list containing various elements, including vectors, data frames, or other lists. Here's an example:   return\_complex\_object <- function() {  data <- data.frame(Name = c("Alice", "Bob", "Charlie"),  Age = c(25, 30, 22))  info\_list <- list(Name = c("John", "Jane"),  Scores = c(90, 85))  result <- list(Data = data, Info = info\_list  return(result)  # Call the function and store the returned complex object  my\_result <- return\_complex\_object()  # Access elements of the returned complex object  data <- my\_result$Data  info <- my\_result$Info  **Returning a Data Frame**: If your function primarily processes data and you want to return a complex object, you can return a data frame. Data frames are useful for storing structured data. Here's an example:  return\_data\_frame <- function() {  data <- data.frame(Name = c("Alice", "Bob", "Charlie"),  Age = c(25, 30, 22))  return(data)  }  # Call the function and store the returned data frame  my\_data\_frame <- return\_data\_frame()  Returning Custom Objects: You can define your own custom classes and return objects of those classes from functions. This is useful when you have specific data structures or objects with methods. Here's a simplified example:  # Define a custom class  Person <- R6::R6Class(  classname = "Person",  public = list(  name = NULL,  age = NULL,  initialize = function(name, age) {  self$name <- name  self$age <- age  },  introduce = function() {  cat("My name is", self$name, "and I am", self$age, "years old.\n")  }  )  )  return\_custom\_object <- function() {  person <- Person$new("Alice", 25)  return(person)  }  # Call the function and use methods of the custom object  my\_person <- return\_custom\_object()  my\_person$introduce()  You can return any complex data structure that suits your needs, whether it's a list, data frame, custom class object, or another type of complex object. The key is to ensure that the returned object contains the data and structure you require for further analysis or processing.    **Functions are Objective in R:**  In R, functions are first-class objects, which means they can be treated as objects, assigned to variables, passed as arguments to other functions, and returned from functions. This property allows you to work with functions in a flexible and powerful way, making it a functional programming language.  Here are some key characteristics of functions as first-class objects in R:   1. **1.Functions can be assigned to variables:** 2. You can assign a function to a variable, and then use that variable to call the function. 3. **add <- function(a, b) {** 4. **return(a + b)** 5. **}** 6. **result <- add(3, 5)** 7. **2.** 8. **Functions can be passed as arguments to other functions:** 9. You can pass functions as arguments to other functions. This is commonly used in functional programming techniques like **apply** and **lapply**. 10. **apply\_operation <- function(func, a, b) {** 11. **return(func(a, b))** 12. **}** 13. **addition\_result <- apply\_operation(add, 3, 5)** 14. **3.**   **Functions can be returned from other functions:** Functions can return other functions, allowing you to create closures or generate functions dynamically.   1. **create\_multiplier <- function(factor) {** 2. **return(function(x) {** 3. **return(x \* factor)** 4. **})** 5. **}** 6. **double <- create\_multiplier(2)** 7. **triple <- create\_multiplier(3)** 8. **result1 <- double(4) # Returns 8** 9. **result2 <- triple(4) # Returns 12**   **4.Functions can be elements of lists or data frames:** You can store functions in lists or data frames like any other objects.  **my\_functions <- list(add, subtract, multiply)**  **result <- my\_functions[[1]](3, 5) # Calls the 'add' function**  **5.Functions can be defined within other functions:** You can define functions inside other functions, creating closures that capture their environment.  **outer\_function <- function(a) {**  **inner\_function <- function(b) {**  **return(a \* b)**  **}**  **return(inner\_function)**  **}**  **my\_closure <- outer\_function(3)**  **result <- my\_closure(5) # Returns** .  **NO POINTERS IN R** |

**Simplicity and Safety:** R is designed to be an accessible language for statisticians, data scientists, and researchers who may not have a strong background in computer science. Eliminating pointers simplifies the language and reduces the risk of memory-related bugs, such as segmentation faults or memory leaks.

1. **Garbage Collection:** R uses automatic garbage collection to manage memory, so users don't need to worry about deallocating memory explicitly. This approach is more user-friendly and helps prevent memory leaks.
2. **High-Level Abstractions:** R provides high-level data structures like vectors, lists, data frames, and factors, which are designed to simplify data manipulation. These structures abstract away the low-level memory details.
3. **Focus on Data Analysis:** R's primary purpose is data analysis and statistical computing. By abstracting memory management, R allows users to concentrate on data-related tasks rather than managing memory.

While R doesn't have pointers in the traditional sense, it does provide a powerful and flexible system for working with data through its data structures and functions. If you need to interact with low-level memory, interface with other languages like C or C++ is possible through R packages like "Rcpp," which allows you to write and call C/C++ code from within R while still benefiting from R's high-level features and memory management.

**Recursion in R**

Recursion is a programming technique in which a function calls itself to solve a problem. In R, you can create recursive functions just like in many other programming languages. Here's a basic example of a recursive function in R:

# Recursive function to calculate the factorial of a number

factorial <- function(n) {

if (n == 0 || n == 1) {

return(1)

} else {

return(n \* factorial(n - 1))

}

}

# Calculate the factorial of 5

result <- factorial(5)

print(result) # Output: 120

In this example, the **factorial** function calculates the factorial of a number **n** using recursion. It checks for the base cases where **n** is 0 or 1 (the factorial of 0 and 1 is 1), and in other cases, it recursively calls itself with **n-1** until it reaches the base cases.

Here's how recursion works in this example:

* **factorial(5)** calls **factorial(4)** since **5 \* factorial(4)** is required.
* **factorial(4)** calls **factorial(3)** since **4 \* factorial(3)** is required.
* This pattern continues until **factorial(1)** is reached, which returns 1.

Then, the results are propagated back up the call stack to calculate the final result, which is 120.

When writing recursive functions in R, it's important to define one or more base cases where the recursion stops. Without base cases, the function will keep calling itself indefinitely, leading to a stack overflow error.

Here are some best practices for using recursion in R:

1. Ensure that you have base cases that terminate the recursion.
2. Make sure that the recursive calls reduce the problem towards the base cases.
3. Be mindful of potential performance issues with deep recursion, as R may not optimize tail recursion.

Recursion can be a powerful tool for solving problems that naturally exhibit a recursive structure, but it should be used judiciously and with care to avoid unnecessary performance overhead and stack overflow errors.

**A Quicksort Implementation-Extended Extended Example:**

**# QuickSort function**

**quick\_sort <- function(arr) {**

**# Base case: if the array has 0 or 1 elements, it's already sorted**

**if (length(arr) <= 1) {**

**return(arr)**

**}**

**# Choose a pivot element (in this case, we'll choose the first element)**

**pivot <- arr[1]**

**# Divide the array into three parts: elements less than the pivot,**

**# elements equal to the pivot, and elements greater than the pivot**

**less <- arr[arr < pivot]**

**equal <- arr[arr == pivot]**

**greater <- arr[arr > pivot]**

**# Recursively sort the "less" and "greater" parts**

**sorted\_less <- quick\_sort(less)**

**sorted\_greater <- quick\_sort(greater)**

**# Concatenate the three parts to form the sorted array**

**sorted\_arr <- c(sorted\_less, equal, sorted\_greater)**

**return(sorted\_arr)**

**}**

**# Example usage:**

**unsorted\_list <- c(5, 2, 9, 3, 6, 1, 8, 7, 4)**

**sorted\_list <- quick\_sort(unsorted\_list)**

**print(sorted\_list)**

In this implementation:

1. **quick\_sort** is a recursive function that takes an array **arr** as its argument.
2. The base case checks if the length of the array is 0 or 1. If so, it returns the array because it's already sorted.
3. A pivot element is chosen. In this example, we select the first element of the array as the pivot.
4. The array is divided into three parts: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot.
5. The **quick\_sort** function is called recursively on the "less" and "greater" parts of the array.
6. Finally, the sorted parts are concatenated together to form the fully sorted array.

When you run this code with **unsorted\_list**, you'll get the sorted list as the output.

QuickSort is a divide-and-conquer algorithm known for its efficiency. However, it's important to note that this basic implementation may not be the most efficient for very large lists, and you can optimize it further by choosing different pivot selection strategies or implementing a more sophisticated partitioning scheme, such as the Lomuto or Hoare partition scheme.

**# Define a structure for tree nodes**

TreeNode <- R6Class("TreeNode",

public = list(

key = NULL,

left = NULL,

right = NULL,

initialize = function(key) {

self$key <- key

self$left <- NULL

self$right <- NULL

}

)

)

**# Define the BinarySearchTree class**

BinarySearchTree <- R6Class("BinarySearchTree",

private = list(

root = NULL,

**# Recursive function to insert a key into the tree**

insert\_recursive = function(node, key) {

if (is.null(node)) {

return(TreeNode$new(key))

}

if (key < node$key) {

node$left <- self$insert\_recursive(node$left, key)

} else if (key > node$key) {

node$right <- self$insert\_recursive(node$right, key)

}

return(node)

},

**# Recursive function to search for a key in the tree**

search\_recursive = function(node, key) {

if (is.null(node) || node$key == key) {

return(node)

}

if (key < node$key) {

return(self$search\_recursive(node$left, key))

} else {

return(self$search\_recursive(node$right, key))

}

},

**# Utility function to perform an in-order traversal (for demonstration purposes)**

inorder\_traversal\_recursive = function(node) {

if (!is.null(node)) {

self$inorder\_traversal\_recursive(node$left)

cat(node$key, " ")

self$inorder\_traversal\_recursive(node$right)

}

}

),

public = list(

initialize = function() {

self$root <- NULL

},

**# Public method to insert a key into the tree**

insert = function(key) {

self$root <- self$insert\_recursive(self$root, key)

},

**# Public method to search for a key in the tree**

search = function(key) {

return(self$search\_recursive(self$root, key))

},

**# Public method to perform an in-order traversal (for demonstration purposes)**

inorder\_traversal = function() {

self$inorder\_traversal\_recursive(self$root)

cat("\n")

}

)

)

**# Example usage:**

**bst <- BinarySearchTree$new()**

**bst$insert(50)**

**bst$insert(30)**

**bst$insert(70)**

**bst$insert(20)**

**bst$insert(40)**

**bst$insert(60)**

**bst$insert(80)**

**bst$inorder\_traversal() # In-order traversal to print sorted keys**

**# Output: 20 30 40 50 60 70 80**

**result <- bst$search(60)**

**if (!is.null(result)) {**

**cat("Key 60 found in the tree.\n")**

**} else {**

**cat("Key 60 not found in the tree.\n")**

**}**

* We define a **TreeNode** class to represent individual nodes in the BST.
* We define a **BinarySearchTree** class that contains methods for inserting keys, searching for keys, and performing an in-order traversal.

You can create a new BST, insert keys into it, search for keys, and perform in-order traversals as demonstrated in the example usage section.

This is a basic implementation of a BST in R, and you can further extend it by adding other operations like deletion or additional traversal methods as needed.