

R Programming

UNIT – II

Control Structures:

Control structures are the statements, which are used to control the execution and flow of the program based on the conditions.

R Programming supports the following control structures or statements. Those are,

- if
- if-else
- if-else-if
- nested if
- while
- for
- repeat and break
- next

if statement:

The if statement is the simplest decision-making statement which helps us to take a decision based on the condition.

In other words, the if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.

Syntax:

```
if (condition)
{
    statements
}
```

Example:

Filename: ifdemo.R

```
x <- 50
# if statement
if(x > 10){
    cat(x, "is greater than 10")
}
```

Output:

```
Rscriptifdemo.R
50 is greater than 10
```

if-else statement:

An if-else statement is the if statement followed by an else statement. An if-else statement, else statement will be executed when the condition will false. In simple words, If a condition will have true value, then the if block gets executed otherwise, the else block will get executed.

Syntax:

```
if (condition){
    statements
}else{
    statements
}
```

Example:

Filename: ifelsedemo.R

```
num<- 21
# if-else statement
if(num %% 2 == 0){
    print("Even Number")
}else {
    print("Odd Number")
}
```

Output:

```
Rscriptifelsedemo.R
[1] "Odd Number"
```

if-else-ifstatement:

The if-else-if statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.

Syntax:

```
if (condition1){
    statements
}else if (condition2){
    statements
}elseif (condition3){
    statements
}else {
    statements
}
```

Example:

Filename: ifelseifdemo.R

```
age <- readline(prompt="Enter your age: ")
age <- as.integer(age)
if(age <18) {
  print("You are child")
} else if(age>50) {
  print("You are old guy")
} else {
  print("You are adult")
}
```

Output:

Sample Run1:

```
Rscriptifelseifdemo.R
Enter your age: 36
[1] "You are adult"
```

Sample Run2:

```
Rscriptifelseifdemo.R
Enter your age: 16
[1] "You are child"
```

nested if statement:

The nested-if is a conditional statement that contains another if statement within it. And it enables us to check multiple conditions.

Syntax:

```
if (condition1){
  statements
  if (condition2){
    statements
  }else{
    Statements
  }
}else {
  statements
}
```

Example:

Filename: nestedifdemo.R

```
num<- readline(prompt="Enter any number: ")
num<- as.integer(num)
if(num>0) {
  print("Number is Positive")
  if(num %% 2 == 0) {
    print("Number is Even")
  } else {
    print("Number is Odd")
  }
} else {
  print("Number is Negative")
}
```

Output:Sample Run1:

```
Rscriptnestedifdemo.R
Enter any number: 12
[1] "Number is Positive"
[1] "Number is Even"
```

Sample Run2:

```
Rscriptnestedifdemo.R
Enter any number: -25
[1] "Number is Negative"
```

while loop statement:

A while loop is a type of control flow statements which is used to iterate a block of code several numbers of times. The while loop terminates when the value of the condition will be false.

Syntax:

```
while (condition) {
  statement
}
```

Example:

Filename: whiledemo.R

```
# R program to print numbers from 1 to 5
n = 1
while(n <= 5) {
  cat(n, " ")
  n <- n + 1
}
```

Output:

```
Rscriptwhiledemo.R
1 2 3 4 5
```

for loop statement:

A for loop is the most popular control flow statement. A for loop is commonly used to iterate over items of a sequence (vector, list, ..).

Syntax:

```
for (value in sequence)
{
    statement
}
```

Example:

Filename: fordemo.R

```
# R program to iterate the vector
names_vec<- c("Madhu", "Durga", "Naveen")
for(name in names_vec)
{
    print(name)
}
```

Output:

Rscriptfordemo.R

```
[1] "Madhu"
[1] "Durga"
[1] "Naveen"
```

repeat and break statement:

repeat is a simple 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 (condition)
    {
        break
    }
}
```

Example:

Filename: repeatdemo.R

```
# R program to demonstrate repeat and break
num<- 1
repeat {
  cat(num, "\n")
  num<- num + 1
  if(num>5) {
    break
  }
}
```

Output:

```
Rscriptrepeatdemo.R
1
2
3
4
5
```

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.

Syntax:

```
next
```

Example:

Filename: nextdemo.R

```
# R program to demonstrate next
num<- 1
for(numin 1:10) {
  if(num %% 2 == 0) {
    next
  }
  cat(num, "\n")
}
```

Output:

```
Rscriptnextdemo.R
1
3
5
7
9
```

Functions in R:

In R, A Function is defined as a set of statements which are organized together to perform a specific task.

R provides various ***built-in functions***, and it allows the user to define their own functions (***user-defined functions***).

Built-in functions:

R provides various built-in functions, some of them are,

- | | | | |
|--------------|------------|-----------|------------------|
| • print() | • paste() | • sqrt() | • mean() |
| • cat() | • c() | • floor() | • sum() |
| • readline() | • list() | • ceil() | • as.numeric() |
| • scan() | • matrix() | • round() | • as.character() |
| • class() | • array() | • min() | • as.logical() |
| • sprintf() | • abs() | • max() | • as.integer() |

User-defined functions:

R enables us to create our own functions; these functions are known as user-defined functions.

- In R, the function is created using a keyword called ***function***.

Syntax:

```
func_name<- function(arg_1, arg_2, ...) {  
    Function body  
}
```

- To call a function, use the *function name* followed by parenthesis, like ***func_name()***.

Example: R program to create a simple function.

Filename: sampleFun.R

```
# function creation  
sample_fun<- function() {  
    print("This is Sample Function")  
}  
# function calling  
sample_fun()
```

Output:

```
RscriptsampleFun.R  
[1] "This is Sample Function"
```

Parameters or Arguments in R Functions:

Parameters or arguments are used to pass the information(values) to the function. These are optional. A function can have any number of arguments, they are separated by comma in parentheses.

Example: R program to create a function with two arguments/parameters.

Filename: addFunction.R

```
# function to add 2 numbers
add_nums<- function(x, y) {
  result <- x + y
  print(result)
}
add_nums(25,36)# calling add_nums function
```

Output:

```
RscriptaddFunction.R
[1] 61
```

Default Value of Parameter:

R allows us to use default values for the parameters in user defined functions. These values are used by functions if user doesn't pass any parameter value while calling a function.

Example: R program to create a function with default parameter values.

Filename: defaultFunction.R

```
# function definition with default values
area_Rect = function(length=4, width=3){
  area = length * width
  print(area)
}
# function calling with arguments
area_Rect(6, 3)

# function calling with arguments by changing order
area_Rect(width = 6, length = 5)

# function calling without arguments
area_Rect()
```

Output:

```
RscriptdefaultFunction.R
[1] 18
[1] 30
[1] 12
```

Return Value (or) return function:

You can use **return()** function if you want your function to return the result.

Example:R program to create a function with return() function.

Filename:returnFunction.R

```
# function definition with return
sqr_fun = function(x){
  result = x * x
  return(result) # return the result
}
# function calling
print(sqr_fun(6))
```

Output:

```
RscriptreturnFunction.R
[1] 36
```

Recursive Function:

R also supports function recursion, which means that a function can call itself. This forms a loop, where every time the function is called, it calls itself again and again and this technique is known as recursion.

Example:R program to find the factorial of the given number using recursive function.

Filename:factorial_fun.R

```
# function definition
factorial_fun<- function(x){
  if(x==0 || x==1)
  {
    return(1)
  }
  else
  {
    return(x * factorial_fun(x-1))
  }
}
# function calling
print(factorial_fun(5))
```

Output:

```
Rscriptfactorial_fun.R
[1] 120
```

Global Variables:

Variables that are created outside of a function are known as global variables. Global variables can be used by everyone, both inside of functions and outside.

Normally, when you create a variable inside a function, that variable is **local**, and can only be used inside that function.

- To create a global variable inside a function, you can use the **global assignment(<<-)** operator.

Example: R program to create a global variable.

Filename: globalVar.R

```
# function definition
my_fun<- function() {
# global variable
msg<<- "Programming Language"
print(paste("R is", msg))
}
my_fun()# funtion calling
print(msg) # print global variable
```

Output:

```
RscriptglobalVar.R
[1] "R is Programming Language"
[1] "Programming Language"
```

- Also, use the **global** assignment operator if you want to change a global variable inside a function.

Example: R program to change the value of a global variable.

Filename: globalvarVal.R

```
msg<- "Programming Language"
# function definition
my_fun<- function() {
# changing value of global variable
msg<<- "Scripting Language"
print(paste("R is", msg))
}
my_fun()# funtion calling
print(msg) # print global variable
```

Output:

```
RscriptglobalVar.R
[1] "R is Scripting Language"
[1] "Scripting Language"
```

Vectors in R:

In R, a **vector** is a basic data structure used to store multiple values of the same type. It is a **one-dimensional** data structure and can hold numeric, character, logical, or other atomic types.

In R, To create a vector, we use **c()** function(*combine function*) and in this, the elements are separated by a comma(,).

Syntax:

```
vect_name<- c(e1,e2,e3,...)
```

Example:

```
># Numeric vector
>numeric_vec<- c(1, 2, 3, 4, 5)

># Character vector
>char_vec<- c("apple", "banana", "cherry")

># Logical vector
>logical_vec<- c(TRUE, FALSE, TRUE)
```

Generating sequences:

R provides various ways to generate sequences or to create vectors. Those are,

- Using **:** Operator
- Using **seq()** Function
- Using **rep()** Function
- Using **sample()** Function

Using : Operator

The **:** **operator** is used to generate sequences with a step size of 1.

Example:

```
# Sequence from 1 to 10
seq1 <- 1:10
print(seq1) # Output: 1 2 3 4 5 6 7 8 9 10

# Sequence from 10 to 1 (decreasing)
seq2 <- 10:1
print(seq2) # Output: 10 9 8 7 6 5 4 3 2 1
```

Using seq() Function:

The **seq()** is used to generate sequences and this function allows more control over start, end, and step size.

Example:

```
># Sequence from 1 to 10 with step size 2
> seq1 <- seq(1, 10, by = 2)
> print(seq1) # Output: 1 3 5 7 9

># Sequence from 1 to 10 with step size 3
> seq2 <- seq(1, 10, by = 3)
> print(seq2) # Output: 1 4 7 10

># Sequence from 0 to 10 with 5 equally spaced values
> seq3 <- seq(0, 10, length.out = 5)
> print(seq3) # Output: 0.0 2.5 5.0 7.5 10.0

># Sequence from 0 to 10 with 6 equally spaced values
> seq3 <- seq(0, 10, length.out = 6)
> print(seq3) # Output: 0 2 4 6 8 10
```

Using rep() Function:

The **rep()** is used to generate sequences and this function generates repeated values.

Example:

```
# Repeat a single value
rep1 <- rep(5, times = 4)
print(rep1) # Output: 5 5 5 5

# Repeat a vector
rep2 <- rep(c(1, 2, 3), times = 3)
print(rep2) # Output: 1 2 3 1 2 3 1 2 3

# Repeat each element a specific number of times
rep3 <- rep(c(1, 2, 3), each = 2)
print(rep3) # Output: 1 1 2 2 3 3
```

Using sample() Function:

The **sample()** function is used to generate random sequences from a given set of values.

Example:

```
># Random sample of 5 integers from 1 to 10 (without replacement)
> sample1 <- sample(1:10, 5)
> print(sample1) #Output: 3 8 1 5 9

># Random sample of 5 integers from 1 to 10 (with replacement)
> sample1 <- sample(1:10, 5, replace=TRUE)
> print(sample1) #Output: 2 2 8 2 3
```

Extracting elements of a vector using subscripts: (or) Vector Indexing:

In R, you can extract elements from a vector using subscripts (also known as indices). There are multiple ways to do this. Those are,

- **Using Positive Indices:**

Extract specific elements from vector by specifying their indices or positions.

Example:

```
>vec<- c(10, 20, 30, 40, 50)
># First element
>print(vec[1])# Output: 10

># Second element
>print(vec[2])# Output: 20

># Multiple elements
>print(vec[c(1, 3, 5)])# Output: 10 30 50
```

- **Using Negative Indices:**

Exclude specific elements from vector by specifying negative indices or positions.

Example:

```
>vec<- c(10, 20, 30, 40, 50)
># exclude First element
>print(vec[-1])# Output: 20 30 40 50

># exclude Multiple elements
>print(vec[-c(2, 3)])# Output: 10 40 50
```

- **Using Logical Indexing:**

In R, **logical subscripts** (or logical indexing) allow you to extract elements from a vector based on a logical condition.

Example:

```
>vec<- c(10, 20, 30, 40, 50)
># Extract elements which are > 25
>print(vec[vec>25])# Output: 30 40 50

># Extract elements which are == 20
>print(vec[vec == 20])# Output: 20 40

># Extract elements which are > 10 and < 50
>print(vec[vec>10 &vec<50])# Output: 20 30 40

># Extract elements which are != 30
>print(vec[!(vec == 30)])# Output: 10 20 40 50
```

- **Using : (colon operator):**

Extract a range or sequence of elements from vector using :(colon operator).

Example:

```
>vec<- c(10, 20, 30, 40, 50)
># Extract sequence of elements
>print(vec[1:3]) # Output: 10 20 30

>print(vec[2:4]) # Output: 20 30 40
```

- **Using Named Indices:**

If the vector is named vector, we can extract elements using named indices.

Example:

```
>named_v<- c(a = 100, b = 200, c = 300)
># extract element using named index
>print(named_v["c"]) # Output: c 300

>print(named_v["b"]) # Output: b 200
```

Adding and Deleting Vector Elements:

In R, Vectors have fixed length, so we cannot modify (adding or deleting) vectors directly, but there are different techniques to **add** and **delete** elements from a vector dynamically.

Adding Elements to a Vector:

Since vectors in R have a fixed length, you cannot modify them in-place, but you can create a new vector with additional elements.

- **Adding Elements at the Beginning:**

Example:

```
> vec_num <- c(10, 20, 30,40,50)
> vec_num <- c(5, vec_num)# Add5at beginning
>print(vec_num) # Output: 5 10 20 30 40 50
```

- **Adding Elements at the Ending:**

Example:

```
>vec_num<- c(10, 20, 30, 40, 50)
>vec_num <- c(vec_num, 60, 70) # Appends 60 and 70
>print(vec_num) # Output: 10 20 30 40 50 60 70
```

- **Inserting Elements at a Specific Position:**

Example:

```
> vec_num <- c(10, 20, 30, 40, 50)
> vec_num <- append(vec_num, values = 25, after = 2)
> print(vec_num) # Output: 10 20 25 30 40 50
```

Deleting Elements from a Vector:

R does not provide direct deletion functions, but you can remove elements by **creating a new vector without them**.

- **Removing by Index:**

Example:

```
> vec <- c(10, 20, 30, 40, 50)
> vec <- vec[-3] # Removes the element at index 3
> print(vec) # Output: 10 20 40 50
```

- **Removing Multiple Elements by Index:**

Example:

```
> vec <- c(10, 20, 30, 40, 50)
> vec <- vec[-c(2, 4)] # Removes elements at positions 2 and 4
> print(vec) # output: 10 30 50
```

- **Removing Elements by Value:**

Example:

```
> vec <- c(10, 20, 30, 40, 50)
> vec <- vec[vec != 30] # Removes the value 30
> print(vec) # output: 10 20 40 50
```

Here is the summary of various methods for adding and deleting elements from vectors.

Operation	Method
Add elements at the beginning	<code>v <- c(new_values, v)</code>
Add elements at the end	<code>v <- c(v, new_values)</code>
Insert elements at a position	<code>v <- append(v, values, after = index)</code>
Remove element by index	<code>v <- v[-index]</code>
Remove multiple elements	<code>v <- v[-c(index1, index2)]</code>
Remove element by value	<code>v <- v[v != value]</code>

Obtaining the Length of a Vector:

In R, we can obtain the **length (number of elements)** of a vector using the *length()* function.

Syntax:

```
length(vector)
```

Example:

```
> nums <- c(10, 20, 30, 40, 50)
> print(length(nums)) # Output: 5

> nums <- c(10, 20, 30, 40, 50, NA)
> print(length(nums)) # Output: 6

> names <- c("one", "two", "three", "four")
> print(length(names)) # Output: 4
```

Matrices and Arrays as Vectors:

In R, **matrices** and **arrays** are internally stored as **vectors**. This means they can be treated like vectors in many operations.

Matrices as Vectors:

A matrix is just a vector with a dimension attribute (dim). If you remove the dimensions, it behaves like a regular vector. The function *as.vector()* used to convert matrix into vector.

Syntax:

```
as.vector(matrix)
```

Example:

```
> m <- matrix(1:9, nrow = 3, ncol = 3)
> print(m)
# Output:
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9

# Converting a Matrix to a Vector
> v <- as.vector(m)
> print(v)
# Output: 1 2 3 4 5 6 7 8 9

# Converting a Matrix to a Vector by Removing Dimensions
> dim(m) <- NULL
> print(m) # Output: 1 2 3 4 5 6 7 8 9
```

Arrays as Vectors:

An array is a multi-dimensional vector. It behaves like a vector when dimensions are removed. The function **as.vector()** is used to convert an array into vector.

Syntax:

```
as.vector(array)
```

Example:

```
>a <- array(1:12, dim = c(2, 3, 2))
>print(a)
# Output :
# , , 1
#      [,1] [,2] [,3]
# [1,]    1    3    5
# [2,]    2    4    6
#
# , , 2
#      [,1] [,2] [,3]
# [1,]    7    9   11
# [2,]    8   10   12

# Converting an Array to a Vector
> v <- as.vector(a)
>print(v)
# Output:  1  2  3  4  5  6  7  8  9 10 11 12

# Removing Dimensions
> dim(a) <- NULL
>print(a)
# Output:  1  2  3  4  5  6  7  8  9 10 11 12
```

Vector Arithmetic and Logical Operations:

In R, vectors are fundamental data structures that allow us to perform both arithmetic and logical operations. These operations are carried out element-wise, without requiring loops.

Arithmetic Operations on Vectors:

We can perform arithmetic operations on vectors of the **same length**. Here are some examples.

Example:

```
> v1 <- c(2, 4, 6)
> v2 <- c(1, 2, 3)

> # Addition
> print(v1 + v2)
# Output: 3 6 9
```

```

> # Subtraction
> print(v1 - v2)
# Output: 1 2 3

> # Multiplication
> print(v1 * v2)
# Output: 2 8 18

> # Division
> print(v1 / v2)
# Output: 2 2 2

> # Exponentiation
> print(v1 ^ v2)
# Output: 2 16 216

```

- If vectors have different lengths, the **shorter one is recycled** (Recycling Rule in Arithmetic) to perform arithmetic operations.

Example:

```

> v1 <- c(10, 20, 30, 40)
> v2 <- c(2, 4)

> # Addition
> print(v1 + v2) # v2 is repeated: (2, 4, 2, 4)
# Output: 12 24 32 44

> print(v1 - v2) # v2 is repeated: (2, 4, 2, 4)
# Output: 8 16 28 36

```

Logical Operations on Vectors:

We can perform logical operations *element-wise* on vectors. Here are some examples.

Example:

```

> v1 <- c(10, 20, 30)
> v2 <- c(15, 20, 25)
# Element-wise comparisons
> print(v1 > v2)
# Output: FALSE FALSE TRUE

> print(v1 < v2)
# Output: TRUE FALSE FALSE

> print(v1 == v2)
# Output: FALSE TRUE FALSE

```

Dates and Times:

R has developed a special representation for dates and times. Dates are represented by the **Date** class and times are represented by the **POSIXct** or the **POSIXlt** class. Dates are stored internally as the number of days since 1970-01-01 while times are stored internally as the number of seconds since 1970-01-01.

Dates in R:

Dates are represented by the **Date** class and we can get dates from a string of characters using the **as.Date()** function.

- To get the Current Date use **Sys.Date()** function.

Example:

```
> x <- Sys.Date()  
> print(x)  
# Output: "2025-03-05"
```

- To get a 'Date' object from character.

Example:

```
> x <- as.Date("2025-01-01")  
> print(x)  
# Output: "2025-01-01"
```

- You can see the internal representation of a Date object by using the **unclass()** function.

Example:

```
> x <- as.Date("1970-01-03")  
> print(unclass(x))  
# Output: 2
```

Times in R:

Times are represented by the **POSIXct** or the **POSIXlt** class. R uses **POSIXct** (compact, for calculations) and **POSIXlt** (list-like, for extraction of components).

- To get the Current Date and Time use **Sys.time()** function.

Example:

```
> dt <- Sys.time()  
> print(dt)  
# Output: "2025-03-10 21:03:46 IST"  
> print(class(dt))  
# Output: "POSIXct" "POSIXt"
```

- Extracting Components using the POSIXlt object.

Example:

```
# converging date object into POSIXlt object
> p_lt <- as.POSIXlt(dt)

> p_lt$year + 1900 # Year
# Output: 2025
> p_lt$mon + 1     # Month (0-based index, so add 1)
# Output: 3
> p_lt$mday        # Day of the month
# Output: 10
> p_lt$hour        # Hour
# Output: 21
> p_lt$min         # Minutes
# Output: 3
> p_lt$sec         # Seconds
# Output: 46.38
```

- Arithmetic operations on *Date* and *POSIXct* objects.

Example:

```
> today_date <- Sys.Date()
> # Adding days
> next_week <- today_date + 7
> print(next_week)
# Output: "2025-03-17"

> # Difference between dates
> days_diff <- as.Date("2025-04-01") - today_date
> print(days_diff)
# Output: Time difference of 22 days
```

Formatting Dates and Times

- Use `format()` to display dates/times in different formats.

Example:

```
> format(today_date, "%B %d, %Y")
# Output: "March 10, 2025"
> format(p_ct, "%Y-%m-%d %H:%M:%S %Z")
# Output: "2025-03-10 14:30:00 UTC"
```

Code	Description	Example
%Y	4-digit year	2025
%y	2-digit year	25
%m	Month (01-12)	03
%d	Day (01-31)	10
%B	Full month name	March

Code	Description	Example
%b	Abbreviated month	Mar
%H	Hours (00-23)	14
%M	Minutes (00-59)	30
%S	Seconds (00-59)	00
%Z	Timezone	UTC