Unit-4
Conditionals and Control FLow

Relational Operators:

- The symbols which are used to compare any two numeric values, are known as relational operators.
- Syntax on relational expression: (num1) operator (num2)
- If the specified relation is valid/true then the expression returns TRUE, else it returns FALSE. (i.e., boolean values)

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

```
x <- 5
y <- 8

print(x > y)
print(x == y)
print(x <= y)
```

```
[1] FALSE
[1] FALSE
[1] TRUE
```

Relational Operators and Vectors:

- Every relational operator that can be applied on numeric values, can be applied on numeric vectors too.
- Since for the comparison b/w two numeric variables, the values of variables will be compared. Similarly, for the comparison b/w two numeric vectors, the corresponding values of two vectors will be compared.
- As a result, the expression returns a vector containing boolean values of corresponding comparisons of two vector values.

```r
vector_1 <- c(2, 4, 6, 8)
vector_2 <- c(1, 4, 6, 10)

result_vector <- vector_1 >= vector_2
# (2>=1, 4>=4, 6>=6, 8>=10)

print(result_vector)
```

```
[1]  TRUE  TRUE  TRUE FALSE
```

- If the lengths of two vectors are not the same, then the comparisons are done by iterating through the vector with a smaller length.

```r
vector_1 <- c(2, 4, 6, 8)
vector_2 <- c(1, 4)

result_vector <- vector_1 >= vector_2
# (2>=1, 4>=4, 6>=1, 8>=4)

print(result_vector)
```

```
[1] TRUE TRUE TRUE TRUE
```

Logical Operators (and) Logical Operators and Vectors:

- These are the symbols used to combine any two boolean/logical values (TRUE / FALSE).
- The final result depends on the operator used in the expression.
- It is mostly used when two or more conditions are to be specified.

Operator	Description
&	Element-wise Logical AND operator. It returns TRUE if both elements are TRUE
&&	Logical AND operator - Returns TRUE if both statements are TRUE
|	Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE
||	Logical OR operator. It returns TRUE if one of the statement is TRUE.
!	Logical NOT - returns FALSE if statement is TRUE

- The main difference between && / || and & / | in R lies in their short-circuiting behavior:
    1) && (and) ||:
        - These are used for scalar logical expressions. (single valued numeric variables)
        - If the first operand of && is FALSE, the second operand is not evaluated because the overall result is already determined to be FALSE. Similarly, if the first operand of || is TRUE, the second operand is not evaluated because the overall result is already determined to be TRUE.

```
a <- TRUE
b <- FALSE

print(a && b )
print(a || b )
print(!a)
```

```
[1] FALSE
[1] TRUE
[1] FALSE
```

    2) & (and) |:
        - These are used for element-wise logical operations on vectors.
        - They do not short-circuit; both operands are always evaluated.
        - When used with vectors, (&) and (|) perform element-wise comparisons, producing a vector of logical results.

```
vector_1 <- c(TRUE, FALSE, TRUE, TRUE)
vector_2 <- c(FALSE, TRUE, FALSE, TRUE)

result_vector_1 <- vector_1 & vector_2
result_vector_2 <- vector_1 | vector_2

print(result_vector_1)
print(result_vector_2)
```

```
[1] FALSE FALSE FALSE  TRUE
[1] TRUE TRUE TRUE TRUE
```

Conditional Statements:

- These are used for decision making. I.e., executing a condition only when the specifies conditional satisfies.
- The main constructs are: if, else if, and else.

```
score <- 75

if (score >= 90) {
  print("Grade: A")
} else if (score >= 80) {
  print("Grade: B")
} else if (score >= 70) {
  print("Grade: C")
} else {
  print("Grade: F")
}
```

```
[1] "Grade: C"
```

Iterative Programming in R

Introduction:

- Iterative programming involves the use of loops to repeat a set of instructions until a specific condition is met.
- Iteration refers to the process of repeating a set of instructions multiple times.
- Loops are a fundamental concept in iterative programming.
- They allow you to execute a block of code repeatedly until a certain condition is met.
- In R, there are two main types of loops:
    - while loops
    - for loops
- Additionally, we can use these loops to iterate over elements in a list, too.

While Loop:

- A while loop in R continues to execute a block of code as long as a specified condition is true.

```r
count <- 1

while (count <= 5) {
  print(paste("Iteration", count))
  count <- count + 1
}
```

```
[1] "Iteration 1"
[1] "Iteration 2"
[1] "Iteration 3"
[1] "Iteration 4"
[1] "Iteration 5"
```

For Loop:

- A for loop is used to iterate over a sequence (vectors, array, lists, etc) and execute a block of code for each element in the sequence.
- We use the in keyword to specify a sequence of data structures through which the loop is iterating.

```r
vector <- c(1,2,3,4,5)

for (x in vector) {
  print(paste("Iteration", x))
}
```

```
[1] "Iteration 1"
[1] "Iteration 2"
[1] "Iteration 3"
[1] "Iteration 4"
[1] "Iteration 5"
```

Looping over a List:

- we can use loops to iterate over elements in a list. Lists in R can contain various data types, and you can access elements using indexing.

```r
fruits <- list("apple", "banana", "cherry", "mango", "orange")

for (x in fruits) {
  if (x == "cherry") {
    break
  }
  print(x)
}
```

```
[1] "apple"
[1] "banana"
```

Functions in R

- A function is a block of reusable code designed to perform a specific task
- Functions help in modularizing code, making it more readable and maintainable.
- R comes with many built-in functions, and users can also create their own functions.
- A function only runs/executes when it is called. (function name with arguments)
- A function can and cannot have a return statement.

```
my_function <- function(arg1, arg2, ...) {
  # Body of the function
  # Perform operations using arg1, arg2, ...
  # Return a result if needed
}
```

Writing a function in R:

- function keyword is used to define a function with its body under curly braces{ }.
- The data required for computation of a function code can be provided as arguments to the parameters of the function, specified while defining, under parentheses.
- A function can be named by storing its definition into a variable, and calling it with arguments under parentheses, whenever required.

```
add_numbers <- function(x, y) {
  result <- x + y
  return(result)
}

# Call the function
sum_result <- add_numbers(5, 3)
print(sum_result)  # Output: 8
```

Nested Functions:

- We can define a function inside another function. These are called nested functions.
- They can access variables from the outer function.

Ex 1:

```r
outer_function <- function(x) {
  inner_function <- function(y) {
    result <- x + y
    return(result)
  }
  return(inner_function)
}

# Use nested functions
my_nested_function <- outer_function(5)
nested_result <- my_nested_function(3)
print(nested_result)  # Output: 8
```

Ex 2:

```r
Nested_function <- function(x, y) {
  a <- x + y
  return(a)
}

nested_result <- Nested_function(Nested_function(2, 2), Nested_function(3, 3))
print(nested_result)
# Output: 10
```

Function Scoping:

- Understanding scoping is crucial when working with functions.
- R has both global and local scopes.
- Variables defined inside a function are local to that function, which can only be accessed throughout the function block. I.e., cannot be accessed outside the function.
- Variables defined outside a function are global scoped, which can be accessed by any function and can be used throughout the program.

```r
global_var <- 10

my_function <- function() {
  local_var <- 5
  result <- global_var + local_var
  return(result)
}

# Accessing global and local variables
output <- my_function()
print(output)  # Output: 15
```

Recursion:

- Recursion is a common mathematical and programming concept.
- Recursion is a technique where a function calls itself.

```r
factorial <- function(n) {
  if (n == 0 || n == 1) {
    return(1)
  } else {
    return(n * factorial(n - 1))
  }
}

result <- factorial(5)
print(result)  # Output: 120
```

Loading an R Package:

- Packages are collections of R functions, data, and compiled code in a well-defined format.
- They can be easily shared and reused, making it convenient for users to organize and distribute their work.
- There are numerous pre-built packages available in R that cover a wide range of functionalities, such as:
    - **dplyr** - a package for data manipulation and transformation.
    - **ggplot2** - a package for creating elegant and complex data visualizations.
- Inorder to use a package in an R program, it must be loaded first into it.
- To do that, we have **library( )** and **require( )** functions.
- library( ) and require( ) functions are used to load a required package into the current R program.

```
# Load a single package
library(package_name)

# Load multiple packages
library(package1, package2, ...)

# Example: Load dplyr and ggplot2
library(dplyr)
library(ggplot2)
```

```
# Load a single package
require(package_name)

# Load multiple packages
require(package1)
require(package2)
# ...

# Example: Load dplyr and ggplot2
require(dplyr)
require(ggplot2)
```

- The difference b/w **library( )** and **require( )** is:
    + If the specified package is not installed, library() will throw an error and stop the script, while require() will return a logical value (TRUE if the package is available, FALSE otherwise) and allow the script to continue.

- It's a common practice to use library() for interactive work and require() in scripts where we might want to check if a package is available and handle it accordingly.

<u>Mathematical Functions in R:</u>

- R provides a wide range of mathematical functions which are pre-loaded, by default.
- The Mathematical functions are:
    - exp( ) - exponential
    - log( ), log10( ) - logarithmic
    - sin( ), cos( ), tan( ) - trigonometric
    - sqrt( ) - square root
    - abs( ) - absolute value

- ceiling( ) - Ceiling value of a float
- floor( ) - floor value of a float
- runif(num, min = minNum, max = maxNum) - random number generation b.w minNum and maxNum.
- round( ) - rounding off a float number

Ex:

```r
# exp( ) - exponential - e^x
result_1 <- exp(2)
print(result_1)

# log( ), log10( ) - logarithmic
result_2 <- log(10)
print(result_2)
result_3 <- log10(100)
print(result_3)

# sin( ), cos( ), tan( ) - trigonometric
result_4 <- sin(pi/2)
print(result_4)
result_5 <- cos(pi/2)
print(result_5)
result_6 <- tan(pi/4)
print(result_6)

# sqrt( ) - square root
result_7 <- sqrt(9)
print(result_7)

# abs( ) - absolute value
result_8 <- abs(-10)
print(result_8)

# ceiling( ) - Ceiling value of a float
result_9 <- ceiling(2.3)
print(result_9)

# floor( ) - floor value of a float
result_10 <- floor(2.3)
print(result_10)

# runif(num, min = minNum, max = maxNum ) - random number generation b.w  minNum and maxNum.
result_11 <- runif(1, min = 1, max = 10)
print(result_11)

# round( ) - rounding off a float number
result_12 <- round(2.3)
print(result_12)
```

```
[1] 7.389056
[1] 2.302585
[1] 2
[1] 1
[1] 6.123234e-17
[1] 1
[1] 3
[1] 10
[1] 3
[1] 2
[1] 9.54393
[1] 2
```