Topic 2: Programming with R

2023-08-08

In this topic, you will learn about :

- Flow control
- Introduction to function
- Packages, libraries, and repositories

Flow Control

Flow Control

Flow control is a fundamental concept in programming that allows you to control the execution of code based on certain conditions. In R programming, there are several flow control structures that help you manage the flow of your program. These flow control structures include:

1. if-else Statements:

The if-else statement is used to make decisions based on a condition. It allows the program to execute different code blocks depending on whether a certain condition is true or false.

```
# Example 1: Simple if-else statement
x <- 10
if (x > 0) {
  print("x is positive.")
} else {
  print("x is non-positive.")
}
```

```
## [1] "x is positive."
```

```
# Example 2: Nested if-else statement
y <- 15
if (y > 0) {
  if (y %% 2 == 0) {
    print("y is a positive even number.")
  } else {
    print("y is a positive odd number.")
  }
} else {
  print("y is non-positive.")
}
```

```
## [1] "y is a positive odd number."
```

2. if-else if-else Statements:

This is an extension of the if-else statement, allowing you to check multiple conditions and execute different code blocks accordingly.

```r
# Example: if-else if-else statement
z <- 5
if (z > 0) {
  print("z is positive.")
} else if (z < 0) {
  print("z is negative.")
} else {
  print("z is zero.")
}
```

```
## [1] "z is positive."
```

3. For Loops:

The for loop is used to repeat a block of code a specified number of times. It is useful when you know the exact number of iterations needed.

```r
# Example: for loop to print numbers from 1 to 5
for (i in 1:5) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

4. while Loops:

The while loop is used to repeat a block of code as long as a certain condition remains true.

```r
# Example: while loop to print numbers from 1 to 5
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

5. repeat Loop:

The repeat loop is an infinite loop that continues to execute a block of code until a specific condition is met or a break statement is used to exit the loop.

```r
# Example: repeat loop with break statement
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break
  }
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

6. next Statement:

The next statement is used to skip the current iteration of a loop and move to the next iteration.

```
# Example: using next statement in a for loop to skip odd numbers
for (i in 1:5) {
  if (i %% 2 == 1) {
    next
  }
  print(i)
}
```

```
## [1] 2
## [1] 4
```

7. Control Flow with Logical Operators:

Logical operators (&&, ||, !) are used to combine multiple conditions in flow control statements.

```
# Example: combining conditions with logical operators
x <- 10
y <- 5
if (x > 0 && y > 0) {
  print("Both x and y are positive.")
}
```

```
## [1] "Both x and y are positive."
```

These flow control structures allow you to write more complex and flexible programs in R, making your code more efficient and easier to read and maintain.

Introduction to function

Introduction to function

In R programming, functions are blocks of code designed to perform specific tasks. Functions are essential for organizing code, promoting reusability, and making programs more modular. R provides several built-in functions, but you can also create your own custom functions. A function typically takes one or more input arguments, processes them, and returns an output.

Creating Functions in R: In R, you can define functions using the function() keyword followed by the function name, input arguments, and the function body enclosed in curly braces {}. The return() statement is used to specify the output of the function.

Syntax:

```
function_name <- function(arg1, arg2, ...) {
  # Function body
  # Perform computations or tasks
  return(output)
}
```

Example: Simple Function

```r
# Function to add two numbers
add_numbers <- function(a, b) {
  sum <- a + b
  return(sum)
}

# Call the function and store the result in a variable
result <- add_numbers(5, 3)
print(result)
```

```
## [1] 8
```

Example: Function with Default Argument

```r
# Function to calculate the area of a rectangle
rectangle_area <- function(length, width = 1) {
  area <- length * width
  return(area)
}

# Call the function with both arguments
result1 <- rectangle_area(4, 5)
print(result1)
```

```
## [1] 20
```

```r
# Call the function with only one argument (uses the default value for width)
result2 <- rectangle_area(4)
print(result2)
```

```
## [1] 4
```

Example: Function with Multiple Outputs

```r
# Function to calculate both the sum and product of two numbers
sum_and_product <- function(x, y) {
  sum <- x + y
  product <- x * y
  return(list(sum = sum, product = product))
}

# Call the function and store the results in a list
result <- sum_and_product(2, 3)
print(result$sum)
```

```
## [1] 5
```

```r
print(result$product)
```

```
## [1] 6
```

Example: Recursive Function

```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))
```

```
  }
}

# Call the recursive function
result <- factorial(5)
print(result)
```

```
## [1] 120
```

Functions are powerful tools that enable you to write more organized and efficient code in R. They are especially useful when you need to perform repetitive tasks or when you want to encapsulate complex operations into reusable components.