# Conditions and Loops

## If Statement

The if statement evaluates a condition (a boolean expression) and executes a block of code if the condition is TRUE.If the condition is FALSE, the code block is skipped.

```
score <- 70
if (score > 50) {
  print("PASS")
}
```

```
## [1] "PASS"
```

In this example, since score is 70, which is greater than 50, the condition is TRUE, and "PASS" is printed.

## If. . . else Statement

The if. . . else statement allows you to execute one block of code if the condition is TRUE and another block if the condition is FALSE.

```
score <- 45
if (score > 50) {
  print("PASS")
} else {
  print("FAIL")
}
```

```
## [1] "FAIL"
```

Here, since score is 45, which is not greater than 50, the condition is FALSE, and "FAIL" is printed.

1. **if. . . else if. . . else Statement**,For multiple conditions, you can use the if. . . else if. . . else structure. This allows you to check several conditions sequentially.

```
score <- 85
if (score >= 90) {
  print("A")
} else if (score >= 80) {
  print("B")
} else if (score >= 70) {
  print("C")
} else {
  print("F")
}
```

```
## [1] "B"
```

In this example, since score is 85, the second condition (score >= 80) is TRUE, and "B" is printed.

2. **Inline if Statement**,R also supports a more concise form of the if statement, which can be useful for simple conditions.

```
age <- 19
status <- if (age > 18) "Adult" else "Child"
print(status)
```

```
## [1] "Adult"
```

## Using if else for Element-wise Checks

The if else function in R is a handy tool for performing element-wise checks on vectors. It allows you to apply a condition to each element of a vector and return a corresponding value based on whether the condition is TRUE or FALSE.

```
numbers <- c(1, 2, 3, 4, 5)
result <- ifelse(numbers %% 2 == 0, # checks if each number is
                                    # divisible by 2
                 "Even", # If the number is even, return "Even".
                 "Odd")  # If the number is not even (i.e., odd),
                         # return "Odd".
print(result)
```

```
## [1] "Odd"  "Even" "Odd"  "Even" "Odd"
```

## Nesting and Stacking if Statements

## Nesting if Statements

Nesting if statements means placing one if statement inside another.This is useful when you need to check multiple conditions sequentially.

```
x <- 15
if (x > 10) {
  print("Above ten")
  if (x > 20) {
    print("and also above 20!")
  } else {
    print("but not above 20.")
  }
} else {
  print("below 10.")
}
```

```
## [1] "Above ten"
## [1] "but not above 20."
```

## Stacking if. . . else Statements

Stacking if. . . else statements, also known as an if. . . else if. . . else ladder, allows you to evaluate multiple conditions in sequence. This is useful for handling multiple mutually exclusive conditions.

```
score <- 85
if (score >= 90) {
  print("A")
} else if (score >= 80) {
  print("B")
} else if (score >= 70) {
  print("C")
} else {
  print("F")
}
```

```
## [1] "B"
```

## Combining Nesting and Stacking

```
age <- 25
income <- 30000

if (age > 18) {
  if (income > 25000) {
    print("Eligible for loan")
  } else {
    print("Not eligible due to low income")
  }
} else {
  print("Not eligible due to age")
}
```

```
## [1] "Eligible for loan"
```

## Loop

1. **for loop**,a for loop in R is used to iterate over a sequence of elements, applying a set of operations to each element. This helps in automating repetitive tasks and making the code more efficient and readable.

```
for (i in 1:5) {
  print(i)
}
```

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

2. **Nested For Loop**,a nested for loop is a loop inside another loop. This is useful for iterating over multi-dimensional data structures like matrices or data frames.

```
for (i in 1:3) {
  for (j in 1:3) {
    print(paste(i, "x", j, "=", i * j))
  }
}
```

```
## [1] "1 x 1 = 1"
## [1] "1 x 2 = 2"
## [1] "1 x 3 = 3"
## [1] "2 x 1 = 2"
## [1] "2 x 2 = 4"
## [1] "2 x 3 = 6"
## [1] "3 x 1 = 3"
## [1] "3 x 2 = 6"
## [1] "3 x 3 = 9"
```

```
# Create an empty 4x4 matrix
matrix <- matrix(nrow = 4, ncol = 4)

# Fill the matrix using nested for loops
for (i in 1:4) {
  for (j in 1:4) {
    matrix[i, j] <- i * j
  }
}

print(matrix)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

3. **Using Index in a For Loop**,when you use an index in a for loop, you iterate over the positions of elements in a sequence. This is useful when you need to access or modify elements based on their position.

```
# Create a vector
numbers <- c(10, 20, 30, 40, 50)
```

4. Loop through the indices

```
for (i in 1:length(numbers)) {
  print(paste("Index:", i, "Value:", numbers[i]))
}
```

```
## [1] "Index: 1 Value: 10"
## [1] "Index: 2 Value: 20"
## [1] "Index: 3 Value: 30"
## [1] "Index: 4 Value: 40"
## [1] "Index: 5 Value: 50"
```

5. **Using Value in a For Loop**,when you use the value directly in a for loop, you iterate over the actual elements of the sequence. This is simpler when you only need to work with the values themselves.

```
# Create a vector
numbers <- c(10, 20, 30, 40, 50)

# Loop through the values
for (value in numbers) {
  print(paste("Value:", value))
}
```

```
## [1] "Value: 10"
## [1] "Value: 20"
## [1] "Value: 30"
## [1] "Value: 40"
## [1] "Value: 50"
```

6. **While Loop**,a while loop in R is used to repeatedly execute a block of code as long as a specified condition remains TRUE. This type of loop is useful when the number of iterations is not known beforehand and depends on dynamic conditions.

```
# Initialize the counter
counter <- 1

# Loop while counter is less than or equal to 5
while (counter <= 5) {
  print(counter)
  counter <- counter + 1
}
```

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

Let's use a while loop to sum numbers from 1 to 10

```
# Initialize variables
sum <- 0
i <- 1

# Loop while i is less than or equal to 10
while (i <= 10) {
  sum <- sum + i
  i <- i + 1
}
```

## Implicit Looping with apply

In R, the apply family of functions provides a way to perform operations on data structures without explicitly writing loops. This is known as implicit looping. These functions are efficient and often more readable than traditional loops.

1. **The apply Function**, the apply function is used to apply a function to the rows or columns of a matrix or data frame.

```
# Create a matrix
matrix_data <- matrix(1:9, nrow = 3, ncol = 3)

# Apply the sum function to each row
row_sums <- apply(matrix_data, 1, sum)
print(row_sums)
```

```
## [1] 12 15 18
```

2. **The lapply Function**,the lapply function applies a function to each element of a list and returns a list.

```
# Create a list
list_data <- list(a = 1:5, b = 6:10, c = 11:15)

# Apply the mean function to each element of the list
list_means <- lapply(list_data, mean)
print(list_means)
```

```
## $a
## [1] 3
##
## $b
## [1] 8
##
## $c
## [1] 13
```

3. **The sapply Function**,the sapply function is similar to lapply, but it tries to simplify the result into a vector or matrix if possible.

```
# Create a list
list_data <- list(a = 1:5, b = 6:10, c = 11:15)

# Apply the mean function to each element of the list
list_means <- sapply(list_data, mean)
print(list_means)
```

```
##  a  b  c
##  3  8 13
```

4. **The tapply Function**,the tapply function applies a function to subsets of a vector,defined by another vector (usually a factor).

```r
# Create a vector and a factor
values <- c(1, 2, 3, 4, 5, 6)
groups <- factor(c("A", "A", "B", "B", "C", "C"))

# Apply the sum function to each group
group_sums <- tapply(values, groups, sum)
print(group_sums)
```

```
##  A  B  C
##  3  7 11
```