R for Bioinformatics

Fundamentals of R: Part III

Md. Jubayer Hossain

CHIRAL Bangladesh

Founder & Management Lead

Last Updated on Oct 6, 2023

Agenda

- Control Flow in R
- Functions
- Dates and Times
- Loop Functions
- Simulations - Generating Random Numbers
- Questions?

Control Flow in R

Control Flow Structure

- Control flow refers to the order in which instructions are executed in a program.
- In R, control flow is managed through various constructs that allow for `conditional` execution, `loop` iterations, and branching.
- Conditionals - `if`, `if..else`, `if..else..elif`
- Loops - `for`, `while`, `repeat`

`if` statement

The `if` statement allows for conditional execution of code blocks.

```
# Syntax
if (condition) {
  # Code block executed when condition is TRUE
}
```

`if` statement

```r
# example
x <- 5

if (x > 0) {
  print("x is positive.")
}
```

[1] “x is positive.”

- The variable `x` is assigned a value of `5`.
- The `if` statement checks if `x` is greater than `0`.
- Since the condition is `TRUE` (5 is greater than 0), the code block inside the curly braces is executed.
- Result: The message `"x is positive."` is printed.

`if..else` statement

The if-else statement allows you to execute different code blocks based on a condition.

```
# Syntax
if (condition) {
  # Code block executed when condition is TRUE
} else {
  # Code block executed when condition is FALSE
}
```

`if..else` statement

```r
# example
x <- -2

if (x > 0) {
  print("x is positive.")
} else {
  print("x is negative.")
}
```

[1] “x is negative.”

- The variable `x` is assigned a value of `-2`.
- The `if` statement checks if `x` is greater than `0`.
- Since the condition is `FALSE` (-2 is not greater than `0`), the code block inside the else clause is executed.
- Result: The message `"x is negative."` is printed.

if..else if..else statement

The if-else if-else statement allows you to specify multiple conditions and execute different code blocks accordingly.

```
# Syntax
if (condition1) {
  # Code block executed when condition1 is TRUE
} else if (condition2) {
  # Code block executed when condition1 is FALSE and condition2 is TRUE
} else {
  # Code block executed when all previous conditions are FALSE
}
```

if..else if..else statement

```
# example
x <- 0

if (x > 0) {
  print("x is positive.")
} else if (x < 0) {
  print("x is negative.")
} else {
  print("x is zero.")
}
```

[1] “x is zero.”

- The variable `x` is assigned a value of`0`.
- The `if` statement checks if x is greater than `0`, then if it is less than `0`.
- Since none of the conditions are `TRUE`, the code block inside the else clause is executed.
- Result: The message `"x is zero."` is printed.

`ifelse` function

- The `ifelse()` function in R is a vectorized version of the if-else statement.
- It allows you to perform conditional operations on elements of a vector or data frame based on a specified condition.

```
# Syntax
ifelse(condition, true_value, false_value)
```

ifelse function

```
# example
x <- c(1, 2, 3, 4, 5)
result <- ifelse(x > 3, "Greater", "Less or equal")
result
```

[1] “Less or equal” “Less or equal” “Less or equal” “Greater”
[5] “Greater”

- The vector x contains numeric values.
- The ifelse function checks if each element of x is greater than 3.
- For elements that satisfy the condition (TRUE), the corresponding element in the result is assigned the value “Greater”. Otherwise, it is assigned the value “Less or equal”.
- Result: The result vector will be c(“Less or equal”, “Less or equal”, “Less or equal”, “Greater”, “Greater”).

Handling Missing Values with `ifelse()`

The ifelse function can handle missing values (NA) in the input vectors.

```
x <- c(1, 2, NA, 4, 5)

result <- ifelse(is.na(x), "Missing", ifelse(x > 3, "Greater", "Less or equal"))
result
```

[1] “Less or equal” “Less or equal” “Missing” “Greater”
[5] “Greater”

`ifelse()` with Data Frames

The `ifelse()` function can be applied to specific columns of a data frame.

```r
df <- data.frame(Name = c("John", "Alice", "Emily"), Age = c(25, 30, 35))

df$Category <- ifelse(df$Age >= 30, "Senior", "Junior")
```

- The data frame df contains columns for names and ages.
- The ifelse function is applied to the Age column.
- If the age is greater than or equal to 30, the Category column is assigned the value “Senior”. Otherwise, it is assigned “Junior”.
- Result: The df data frame will have an additional column Category with values c(“Junior”, “Junior”, “Senior”).

for loop

The for loop in R allows for executing a block of code repeatedly for a specified number of iterations.

```
# Syntax
for (variable in sequence) {
  # Code block to be executed in each iteration
}
```

for loop Example

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

[1] 1 [1] 2 [1] 3 [1] 4 [1] 5

- The for loop iterates over the sequence 1:5.
- In each iteration, the loop variable i takes the value of the current element in the sequence.
- The code block inside the curly braces is executed for each iteration, which in this case, is printing the value of i.
- Result: The numbers 1 to 5 are printed.

Using for Loop with Vector

The for loop can be used to iterate over elements of a vector.

```
fruits <- c("apple", "banana", "orange")

for (fruit in fruits) {
  print(fruit)
}
```

[1] “apple” [1] “banana” [1] “orange”

- The vector fruits contains strings representing different fruits.
- The for loop iterates over each element of the fruits vector.
- In each iteration, the loop variable fruit takes the value of the current element.
- The code block inside the curly braces is executed for each iteration, which in this case, is printing the value of fruit.
- Result: The strings “apple”, “banana”, and “orange” are printed.

Controlling `for` Loop with Conditions

You can control the behavior of a `for` loop using conditional statements.

```r
for (i in 1:10) {
  if (i %% 2 == 0) {
    print(paste(i, "is even"))
  } else {
    print(paste(i, "is odd"))
  }
}
```

[1] “1 is odd” [1] “2 is even” [1] “3 is odd” [1] “4 is even” [1] “5 is odd” [1] “6 is even” [1] “7 is odd” [1] “8 is even” [1] “9 is odd” [1] “10 is even”

Controlling `for` Loop with Conditions

- The for loop iterates over the sequence 1:10.
- In each iteration, the loop variable i takes the value of the current element.
- The if statement checks if i is even (divisible by 2) using the modulo operator (%%).
- Based on the condition, the code block inside the corresponding branch of the if statement is executed, printing whether the number is even or odd.
- Result: The numbers from 1 to 10 are printed along with their even or odd classification.

`while` loop

The `while` loop in R allows for executing a block of code repeatedly as long as a specified condition is `TRUE`.

```
while (condition) {
  # Code block to be executed
}
```

`while` Loop Example

```
1 i <- 1
2
3 while (i <= 5) {
4   print(i)
5   i <- i + 1
6 }
```

[1] 1 [1] 2 [1] 3 [1] 4 [1] 5

Using `while` Loop with Conditional Statements

You can use conditional statements within a while loop to control its behavior.

```
i <- 1

while (i <= 10) {
  if (i %% 2 == 0) {
    print(paste(i, "is even"))
  } else {
    print(paste(i, "is odd"))
  }
  i <- i + 1
}
```

[1] “1 is odd” [1] “2 is even” [1] “3 is odd” [1] “4 is even” [1] “5 is odd” [1] “6 is even” [1] “7 is odd” [1] “8 is even” [1] “9 is odd” [1] “10 is even”

Using `while` Loop with Conditional Statements

Controlling `while` Loop with External Conditions

You can control the behavior of a `while` loop by modifying external conditions within the loop.

```
x <- 10

while (x > 0) {
  print(x)
  x <- x - 2
}
```

[1] 10 [1] 8 [1] 6 [1] 4 [1] 2

Controlling `while` Loop with External Conditions

- The variable x is initially set to 10.
- The while loop continues executing as long as x is greater than 0.
- In each iteration, the code block inside the curly braces is executed, which in this case, is printing the value of x.
- After printing, x is decremented by 2 using the assignment statement x <- x - 2.
- The loop continues until x becomes non-positive, at which point the condition becomes FALSE, and the loop terminates.
- Result: The numbers 10, 8, 6, 4, and 2 are printed.

`break` statement

- The break statement in R is used to exit a loop prematurely.
- It is typically used within conditional statements to terminate the loop based on a specific condition.

`break` Statement Example

```
i <- 1

while (i <= 10) {
  print(i)
  if (i == 5) {
    break
  }
  i <- i + 1
}
```

[1] 1 [1] 2 [1] 3 [1] 4 [1] 5

`break` Statement Example

- The variable i is initially set to 1.
- The while loop continues executing as long as i is less than or equal to 10.
- In each iteration, the code block inside the curly braces is executed, which in this case, is printing the value of i.
- The if statement checks if i is equal to 5.
- If the condition is TRUE, the break statement is encountered, causing the loop to terminate immediately.
- After the break statement, the remaining code within the loop is not executed.
- Result: The numbers 1 to 5 are printed, and the loop terminates when i becomes 5.

next statement

- The next statement in R is used to skip the current iteration of a loop and move to the next iteration.
- It is typically used within conditional statements to control the flow of the loop based on a specific condition.

next Statement Example

```
for (i in 1:5) {
  if (i == 3) {
    next
  }
  print(i)
}
```

[1] 1 [1] 2 [1] 4 [1] 5

- The for loop iterates over the values 1 to 5.
- In each iteration, the code block inside the curly braces is executed.
- The if statement checks if i is equal to 3.
- If the condition is TRUE, the next statement is encountered, causing the current iteration to be skipped, and the program flow moves to the next iteration.
- If the condition is FALSE, the code block continues executing, and the value of i is printed.
- Result: The numbers 1, 2, 4, and 5 are printed. The iteration with i equal to 3 is skipped.

`repeat` loop

- The repeat loop in R allows for executing a block of code repeatedly until a specified condition is met.
- It provides a way to create an infinite loop that can be terminated using control flow statements.

`repeat` loop Example

```
i <- 1

repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break
  }
}
```

[1] 1 [1] 2 [1] 3 [1] 4 [1] 5

`repeat` loop Example

Functions

Functions in R

- Functions in R are blocks of reusable code that perform specific tasks.
- They provide modularity and allow for the organization and reuse of code.
- Functions can accept input arguments, perform computations, and return output values.

Types of Functions in R

- Built-in Functions
- User-Defined Functions
- Recursive Functions
- Anonymous Functions (Lambda Functions)
- Higher-Order Functions

Built-in Functions

- R comes with a wide range of built-in functions that are available for immediate use.
- Examples: `mean()`, `sum()`, `max()`, `min()`, `sqrt()`, `length()`, etc.
- These functions are part of the R language and provide basic operations and computations.

User-Defined Functions

- Users can define their own functions in R to perform specific tasks.
- These functions are created using the function keyword followed by the function name, arguments, and function body.
- Examples: Custom functions created by users to solve specific problems.

Recursive Functions

- Recursive functions are functions that call themselves during their execution.
- They are useful for solving problems that can be broken down into smaller, similar sub-problems.
- Examples: Functions that calculate factorials, Fibonacci series, or perform tree traversal.

Anonymous Functions (Lambda Functions)

- Anonymous functions, also known as lambda functions, are functions without a formal name.
- They are typically used for one-time or short computations and are defined using the function keyword without assigning them to a variable.
- Examples: Functions used with higher-order functions like `apply()`, `lapply()`, `sapply()`, etc.

Higher-Order Functions

- Higher-order functions take other functions as arguments or return functions as their output.
- They allow for functional programming paradigms in R.
- Examples: `apply()` family of functions, `map()` functions from the purrr package.

Creating a Function

- To create a function in R, use the function keyword followed by the function name and parentheses for the input arguments.
- The function body is enclosed in curly braces {} and contains the code to be executed.

```r
f <- function() {
  # empty function
}
# Function have their own class

class(f)

# Execute / Call this function
f()
```

Creating a Function

```r
# Function definition
my_function <- function(arg1, arg2) {
  # Code block
  # Perform computations
  result <- arg1 + arg2
  return(result)
}
```

Calling a Function

- To call a function in R, use the function name followed by parentheses, passing the required input arguments.
- The function executes the code within its body and returns the specified output.

```r
# Function call
result <- my_function(3, 5)
print(result)
```

```
[1] 8
```

Function with Default Arguments

- Functions in R can have default values assigned to their arguments, which are used when the arguments are not explicitly provided during function call.
- Default arguments are defined using the assignment operator (=) within the function definition.

```r
# Function definition with default argument
greet <- function(name = "Guest") {
  message <- paste("Hello,", name)
  print(message)
}
```

Function with Variable Arguments

- In R, functions can accept variable arguments using the `...` notation.
- Variable arguments allow for flexibility in the number of inputs passed to the function.

```r
# Function definition with variable arguments
calculate_sum <- function(...) {
  numbers <- list(...)
  total <- sum(numbers)
  return(total)
}
```

Dates and Times

Working with Dates and Times in R

- Dates and times in R can be represented using different classes and functions.
- Let’s explore some common operations and functions related to dates and times in R.

Date and Time Classes

- `Date class`: Represents dates without time information.
- `POSIXct class`: Represents dates and times with second-level precision.
- `POSIXlt class`: Represents dates and times as a list of components.

Creating Dates and Times

- `as.Date()`: Converts a character or numeric value to a Date object.
- `as.POSIXct()` or as.`POSIXlt()`: Converts a character or numeric value to a POSIXct or POSIXlt object.

```
1 date <- as.Date("2023-07-05")
2 datetime <- as.POSIXct("2023-07-05 10:30:00")
```

Formatting and Parsing Dates and Times

- `format()`: Converts a date or time object to a character string with a specified format.
- `strptime()`: Parses a character string representing a date or time into a POSIXlt object, based on a specified format.

```r
formatted_date <- format(date, format = "%Y/%m/%d")
parsed_datetime <- strptime("2023-07-05 10:30:00", format = "%Y-%m-%d %H:%M:%S")
```

Extracting Components

Functions like`year()`, `month()`, `day()`, `hour()`, `minute()`, `second()` can be used to extract specific components from date and time objects.

```
hour <- parsed_datetime$hour
minute <- parsed_datetime$min
second <- parsed_datetime$sec
```

Arithmetic Operations

- Dates and times can be manipulated using arithmetic operations.
- Arithmetic operations on Date objects return new `Date` objects.
- Arithmetic operations on `POSIXct` objects return new `POSIXct` objects, maintaining the time information.

```
1 next_day <- date + 1
```

Time Zones

- R allows working with dates and times in different time zones.
- Time zones can be specified using the tz parameter when creating or converting date and time objects.

```
datetime_ny <- as.POSIXct("2023-07-05 10:30:00", tz = "America/New_York")
```

- Dates and times in R are represented using different classes: Date, POSIXct, and POSIXlt.
- Functions like `as.Date()`, `as.POSIXct()`, `format()`, `strptime()`, etc., facilitate working with dates and times.

Loop Functions

Loop Functions - Apply Family

- The `apply` family of functions in R provides a convenient way to apply a function to subsets of data structures such as vectors, matrices, and data frames.
- These functions eliminate the need for explicit looping and can significantly simplify code.

apply()

- apply() function applies a function over margins of an array or matrix.
- Syntax: apply(X, MARGIN, FUN, ...)
- X: The input data structure (array or matrix).
- MARGIN: The dimension or dimensions along which the function should be applied.
- FUN: The function to be applied.
- ...: Additional arguments to be passed to the function.

```r
# Apply sum function to rows of a matrix
mat <- matrix(1:9, nrow = 3)
result <- apply(mat, 1, sum)
result
```

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

lapply()

- lapply() function applies a function to each element of a list or vector.
- Syntax: lapply(X, FUN, ...)
- X: The input list or vector.
- FUN: The function to be applied.
- ...: Additional arguments to be passed to the function.

```r
# Apply sqrt function to each element of a list
numbers <- list(a = 4, b = 9, c = 16)
result <- lapply(numbers, sqrt)
```

sapply()

- sapply() function is similar to lapply() but simplifies the result into a vector, matrix, or array if possible.
- Syntax: sapply(X, FUN, ...)
- X: The input list or vector.
- FUN: The function to be applied.
- ...: Additional arguments to be passed to the function.

```r
# Apply sum function to each element of a list and simplify the result
numbers <- list(a = 4, b = 9, c = 16)
result <- sapply(numbers, sum)
```

vapply()

- vapply() function is similar to sapply() but allows specifying the output type and shape explicitly.
- Syntax: vapply(X, FUN, FUN.VALUE, ...)
- X: The input list or vector.
- FUN: The function to be applied.
- FUN.VALUE: The desired output type and shape.
- ...: Additional arguments to be passed to the function.

```r
# Apply sum function to each element of a list and specify the output type
numbers <- list(a = 4, b = 9, c = 16)
result <- vapply(numbers, sum, FUN.VALUE = numeric(1))
```

mapply()

- mapply() function applies a function to multiple vectors or lists in parallel.
- Syntax: mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE)
- FUN: The function to be applied.
- ...: Multiple input vectors or lists.
- MoreArgs: Additional arguments to be passed to the function.
- SIMPLIFY: Whether to simplify the result if possible.

```r
# Apply a function to multiple vectors in parallel
vector1 <- c(1, 2, 3)
vector2 <- c(4, 5, 10)
```

Simulations - Generating Random Numbers

Simulations

- Simulations play a crucial role in statistical analysis and modeling.
- R provides various functions to generate random numbers and conduct simulations.

Generating Random Numbers

- runif(): Generates random numbers from a uniform distribution.
- Syntax: runif(n, min = 0, max = 1)
- n: Number of random numbers to generate.
- min: Minimum value of the range.
- max: Maximum value of the range.

```
# Generate 5 random numbers between 0 and 1
random_numbers <- runif(5)
```

Generating Random Integers

- sample(): Generates random integers from a specified range.
- Syntax: sample(x, size, replace = FALSE)
- x: A vector of values from which to sample.
- size: Number of random integers to generate.
- replace: Whether sampling should be done with replacement.

```r
# Generate 3 random integers from 1 to 10
random_integers <- sample(1:10, 3)
```

Generating Random Samples from a Vector

- sample(): Generates random samples from a vector.
- Syntax: sample(x, size, replace = FALSE)
- x: A vector of values from which to sample.
- size: Number of random samples to generate.
- replace: Whether sampling should be done with replacement.

```r
# Generate a random sample of 4 elements from a vector
vector <- c("A", "B", "C", "D", "E")
random_sample <- sample(vector, 4)
```

Generating Random Numbers from Distributions

- R provides functions to generate random numbers from various probability distributions:
  - rnorm(): Generates random numbers from a normal distribution.
  - rexp(): Generates random numbers from an exponential distribution.
  - rgamma(): Generates random numbers from a gamma distribution.
  - rbinom(): Generates random numbers from a binomial distribution.
  - rpois(): Generates random numbers from a Poisson distribution.

```r
# Generate 5 random numbers from a normal distribution with mean 0 and standard devi
random_numbers <- rnorm(5, mean = 0, sd = 1)
```

Seeding Random Number Generation

- To reproduce random results, set a seed value using set.seed().
- Syntax: set.seed(seed)
- seed: A numeric value to initialize the random number generator.

```
# Set a seed value for reproducibility
set.seed(123)
```

Questions?