# Introduction to R - Young Researchers Fellowship Program

## Lecture 7 - Select topics in R Programming

Daniel Sánchez Pazmiño

Laboratorio de Investigación para el Desarrollo del Ecuador

October 2024

# Programming in R

# Concepts of programming

- Programming, in this context, refers to the more "technical" aspects of working with a language.
  - Control structures
  - Loops and functions
  - Parallelization
  - Objects
  - Functional and object-oriented programming

# Concepts of programming

- The textbook Hands-On Programming with R covers many of these topicos at length, for beginners.
- For now, we will briefly cover:
  - Control structures
  - `lapply()`

# Control structures

# Conditionals (if)

- Conditional statements allow programs to execute certain blocks of code based on specified conditions.

```
# If statement checks if a condition is TRUE
if (x > 5) {
  print("x is greater than 5")
}

# If-else statement to cover both TRUE and FALSE conditions
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}

# If-else-if for multiple conditions
if (x > 10) {
```

# Loops

- **Loops**: A fundamental control structure that allows repeated execution of a block of code.

# Types of Loops

1. **For Loops**

```
for (i in 1:n) {
  # Code to execute
}
```

2. **While Loops**

```
while (condition) {
  # Code to execute
}
```

- While loops keep running as long as a condition is true.

# Types of loops

3. **Repeat Loops**: Used to repeatedly execute a block of code until a specified condition is met.

```
repeat {
  # Code to execute
  if (condition) {
    break
  }
}
```

- Must explicitly exit the loop using the break statement when a certain condition is satisfied.
- The `rep()` function performs the same in a more friendly manner.

# For Loop Example

```
# Print numbers 1 to 5
for (i in 1:5) {
  print(i)
}
```

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

# When to Use Loops?

Use loops when: - You need repetitive operations. - Each iteration depends on the previous result. - Code needs to run sequentially.

# Functions

# Intro to functions

- Functions allow you to group a set of instructions and reuse them throughout your code.
- A **function** is a block of reusable code that performs a specific task.
- Functions can take inputs (arguments) and return an output.

```
# Defining a function
add_numbers <- function(a, b) {
  # This function takes two inputs, 'a' and 'b', and returns the
  return(a + b)
}

# Calling the function
result <- add_numbers(3, 4)  # The result will be 7
print(result)  # Prints: 7
```

```
[1] 7
```

# Defining a Function

```
# eval: false
# Function to add two numbers
add_numbers <- function(a, b) {
  # a and b are arguments passed to the function
  result <- a + b  # Add the two numbers
  return(result)  # Return the result
}

# Calling the function with arguments
sum_result <- add_numbers(5, 3)
print(sum_result)  # Prints 8
```

```
[1] 8
```

# Function with a default argument

```r
greet <- function(name = "Guest") {
  # Default value for 'name' is "Guest" if no argument is passed
  message <- paste("Hello,", name)  # Concatenate "Hello," with
  return(message)
}

# Calling the function without an argument
greeting <- greet()
print(greeting)
```

```
[1] "Hello, Guest"
```

```r
# Calling the function with an argument
greeting <- greet("Daniel")
print(greeting)
```

```
[1] "Hello, Daniel"
```

# Returning multiple values

```r
# Function that returns multiple values as a list
calculate <- function(a, b) {
  sum_result <- a + b  # Sum of a and b
  product_result <- a * b  # Product of a and b

  # Return both values as a list
  return(list(sum = sum_result, product = product_result))
}

# Calling the function
results <- calculate(4, 5)

# Accessing the results from the list
print(results$sum)  # Prints the sum (9)
```

```
[1] 9
```

```r
print(results$product)  # Prints the product (20)
```

# lapply() and friends

# Introduction to `lapply`

- lapply is a function that applies a function over elements of a list or vector and returns a list.
- It is part of the apply family in R, which is optimized for repetitive tasks on data structures.

Syntax: `lapply(X, FUN)`

# Example: Using `lapply`

Suppose we have a list of numeric vectors. We want to apply the mean function to each element of the list.

```
# Example list
num_list <- list(a = 1:3, b = 4:6, c = 7:9)

# Apply the mean function to each element in the list
result <- lapply(num_list, mean)

print(result)
```

```
$a
[1] 2

$b
[1] 5

$c
```

# Benefits of Using lapply

- Efficiency: lapply is often faster than traditional loops.
- Clean Code: It simplifies repetitive operations by avoiding explicit loops.
- Output: It always returns a list, even if the input is a vector.

# Custom Functions with lapply

You can pass custom functions to lapply to perform more specific operations on each element of a list.

```
# Custom function to square elements
square <- function(x) {
  return(x^2)
}

# Apply the custom function to each element in num_list
result <- lapply(num_list, square)

print(result)
```

```
$a
[1] 1 4 9

$b
[1] 16 25 36
```

# Combining lapply with unlist()

After applying lapply, if you want to convert the result back into a vector, you can use the unlist function.

```
# Unlist result
vector_result <- unlist(lapply(num_list, mean))

print(vector_result)
```

```
a b c
2 5 8
```

- Explanation: The unlist function converts the list output from lapply into a simple vector.

# Loops vs. lapply

Use loops when: - You have more complex logic that can't be easily vectorized. - There are dependencies between iterations.

Use lapply when:

- You need to apply a function to every element of a list or vector.
- You want simpler, more efficient code.