Programming in R – functions, loops, lists

Programming in R – functions, loops, lists

In this laboratory you will focus on R as a programming language. You will practice writing your own functions (including simple wrappers around existing functions), doing basic linear algebra, using `for` loops, and working with lists and simple functional programming tools.

Throughout this instruction, type the examples into an R script and run them. Modify the code and experiment to see how R behaves.

1. Writing functions and simple wrappers

1.1 Basic functions

In R, functions are defined with `function(...)` and usually stored in variables:

```
# A simple function: compute the mean of a numeric vector,
# ignoring missing values
my_mean <- function(x) {
  mean(x, na.rm = TRUE)
}

values <- c(1, 2, 3, NA, 5)
my_mean(values)
```

Task 1.1

1. Write a function `my_range(x)` that returns the difference between the maximum and minimum of `x`.
2. Test it on a few numeric vectors and verify that it works when `x` contains `NA` values (use `na.rm = TRUE` inside your function).

1.2 Function wrappers around existing functions

Very often we write small wrapper functions that call an existing function but: - set sensible default values for arguments, or - pass only a subset of arguments in a convenient way.

```
# A wrapper around the built-in plot() function
my_plot <- function(x, y,
                    xlab = "x", ylab = "y",
                    main = "My line plot", ...) {
  plot(x, y,
       type = "l",      # line plot
       xlab = xlab,
       ylab = ylab,
       main = main,
       ...)
}

t <- seq(0, 2 * pi, length.out = 200)
y <- sin(t)
my_plot(t, y, main = "Sine wave")
```

Notice the `...` (ellipsis) argument: it allows you to pass any extra arguments through to `plot()`, such as `col`, `lwd`, `ylim`, etc.

Task 1.2

1. Modify `my_plot()` so that it has an additional argument `grid = TRUE`. If `grid` is `TRUE`, the function should call `grid()` after plotting (which adds a simple background grid to the plot).
2. Use your modified wrapper to plot:
   - `cos(t)` in blue (`col = "blue"`),
   - `exp(-t)` in red (`col = "red"`).

1.3 A wrapper for quick summaries and plots

Write a wrapper that combines a numeric summary and a simple plot:

```
quick_summary_plot <- function(x, main = "Quick summary plot") {
  cat("Summary of x:\n")
  print(summary(x))
  hist(x,
       breaks = 20,
       main   = main,
       xlab   = "x",
       col    = "lightgray")
}

set.seed(123)
z <- rnorm(200, mean = 10, sd = 3)
quick_summary_plot(z, main = "Normal data")
```

Task 1.3

Create your own wrapper `quick_boxplot(x, group)` that: 1. Prints the number of observations in each group (use `table(group)`), 2. Draws a boxplot of `x` grouped by `group` using `boxplot(x ~ group)`.

Test it on:

```
group <- sample(c("A", "B", "C"), size = 200, replace = TRUE)
quick_boxplot(z, group)
```

2. Linear algebra in R

R was originally designed for statistical computing and handles matrices and linear algebra very well.

2.1 Creating matrices and basic operations

```
# Create a 3x3 matrix filled by column
A <- matrix(c(1, 2, 3,
              4, 5, 6,
              7, 8, 9),
            nrow = 3, ncol = 3, byrow = FALSE)
```

```
# Create a column vector
b <- c(1, 0, 1)

A
b

# Matrix-vector multiplication
A %*% b

# Transpose
t(A)
```

Task 2.1

1. Create a `4 x 4` matrix `M` with entries from 1 to 16 (use `matrix(1:16, nrow = 4, byrow = TRUE)`).
2. Create a vector `v <- c(1, 2, 3, 4)` and compute `M %*% v`.
3. Compute `t(M) %*% v`. Compare the results.

2.2 Solving linear systems

To solve the system $Ax = b$ we use the `solve()` function.

```
A <- matrix(c(2, 1,
              1, 3),
            nrow = 2, byrow = TRUE)
b <- c(1, 2)

# Solve A x = b
x <- solve(A, b)
x

# Check the result
A %*% x
```

Task 2.2

1. Construct a `3 x 3` matrix `B` and a vector `d` of length 3 such that the system `B x = d` has a unique solution.
   (Hint: choose `B` with a non-zero determinant, for example a triangular matrix.)
2. Use `solve(B, d)` to find `x`, and verify the result by computing `B %*% x`.

2.3 Eigenvalues (optional)

The function `eigen()` computes eigenvalues and eigenvectors:

```
C <- matrix(c(2, 0,
              0, 1),
            nrow = 2, byrow = TRUE)
e <- eigen(C)
e$values
e$vectors
```

Task 2.3 (optional)
Try `eigen()` on a symmetric `3 x 3` matrix of your choice and interpret the results.

3. `for` loops in R

3.1 Basic `for` loop

```
numbers <- 1:10
sum_result <- 0

for (i in numbers) {
  sum_result <- sum_result + i
}

sum_result
```

Task 3.1

1. Write a `for` loop that computes the sum of squares $\sum_{i=1}^{n} i^2$ for a given `n`.
2. Compare your result with `sum((1:n)^2)` to check correctness.

3.2 Filling a vector or matrix in a loop

```
n <- 10
squares <- numeric(n)  # pre-allocate vector

for (i in 1:n) {
  squares[i] <- i^2
}

squares
```

Task 3.2

1. Create an empty matrix `M` of size `n x n` (use `matrix(0, n, n)`).
2. Use nested `for` loops to fill `M[i, j]` with the value `i * j`.
3. Compare your result with `outer(1:n, 1:n, "*")`.

4. Lists and basic functional programming

Lists in R can store objects of different types and sizes. They are the main "container" structure in many R packages.

4.1 Creating and accessing lists

```
student <- list(
  name   = "Alice",
  age    = 21,
  scores = c(80, 90, 85)
)

student$name
student$scores
student[["age"]]
```

Task 4.1

1. Create a list `course` with elements:
   - `title` – a character string,
   - `ects` – a numeric value,
   - `students` – a vector of student names.
2. Access each element using the `$` operator and using double square brackets `[[ ]]`.

4.2 Lists of similar objects

```
students <- list(
  list(name = "Alice", scores = c(80, 90, 85)),
  list(name = "Bob",   scores = c(70, 75, 72)),
  list(name = "Carol", scores = c(95, 92, 98))
)

students[[1]]$name
students[[2]]$scores
```

Task 4.2

1. For the `students` list above, compute the average score for each student using a `for` loop and store the results in a numeric vector.
2. Then repeat the same task using `sapply()` (see below).

4.3 `lapply` and `sapply`

The functions `lapply()` and `sapply()` apply a function to each element of a list:

```
# Average scores for each student (list of students)
avg_scores <- sapply(
  students,
  function(s) mean(s$scores)
)

avg_scores
```

Task 4.3

1. Create a list `X` containing three numeric vectors of different lengths (for example, `rnorm(5)`, `runif(10)`, `rpois(7, lambda = 3)`).
2. Use `lapply(X, mean)` to compute the mean of each vector.
3. Use `sapply(X, length)` to compute the length of each vector.

5. Summary exercise

Put everything together in a small programming task.

Task 5.1

Write a function `simulate_means(n, m)` that:

1. Simulates `m` samples, each of size `n`, from a standard normal distribution (use `rnorm(n)`).
2. Stores these samples in a list of length `m`.
3. Uses `sapply()` to compute the mean of each sample.
4. Returns a list with:
   - the vector of sample means,
   - the overall mean of these means,
   - the overall standard deviation of these means.

Then:

```
set.seed(42)
result <- simulate_means(n = 30, m = 1000)
str(result)
hist(result$means,
     breaks = 30,
     main = "Distribution of sample means",
     xlab  = "Sample mean")
```

Comment briefly on the shape of the histogram of sample means.