R-intro - Session 3

João Gonçalves

27 de Julho de 2018

Contents

Objectives of session 3

- Lists
- User-defined/custom functions
- If conditionals
- For loops
- Combining it all together

List objects

A list is a special object in R that can store virtually *anything* which makes it pretty useful in different situations. You can have a list that contains several vectors, matrices, or dataframes of any size.

Lists are also useful to represent hierarchical or nested data structures! For example, you can have lists inside lists - go inception! :-)

To start a list object you use the function `list()`. Let's check the example below:

```
# A list with three elements named x, y and z:

myList <- list(
  x = rnorm(10), # 10 randomly generated numbers with normal(0,1) distribution
  y = rnorm(5, 20, 1),  # 5 randomly generated numbers with normal(20,1) distribution
```

```
  z = matrix(1:9, nrow = 3, ncol = 3) # A 3x3 matrix
)

print(myList)
```

```
## $x
##  [1]  0.1426445  0.6816563 -0.3983927  0.8361551 -0.4682876 -0.7527596
##  [7] -1.2650036  1.0526486 -1.0461663 -1.0151180
##
## $y
## [1] 19.35054 19.43282 19.90025 19.50295 19.22789
##
## $z
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

To index an element in a list use double brackets `[[]]` or `$` if the list has names. For example, to get the first element of a list named `myList`, you would use `myList[[1]]`. Let's see some examples using the previously created list object:

```
# Access the first element of the list
myList[[1]]
```

```
##  [1]  0.1426445  0.6816563 -0.3983927  0.8361551 -0.4682876 -0.7527596
##  [7] -1.2650036  1.0526486 -1.0461663 -1.0151180
```

```
# Access the third element of the list
myList[[3]]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
# Access the list by name using [[]] or $
myList[["x"]]
```

```
##  [1]  0.1426445  0.6816563 -0.3983927  0.8361551 -0.4682876 -0.7527596
##  [7] -1.2650036  1.0526486 -1.0461663 -1.0151180
```

```
myList[["y"]]
```

```
## [1] 19.35054 19.43282 19.90025 19.50295 19.22789
```

```
# or
myList$x
```

```
##  [1]  0.1426445  0.6816563 -0.3983927  0.8361551 -0.4682876 -0.7527596
##  [7] -1.2650036  1.0526486 -1.0461663 -1.0151180
```

```
myList$y
```

```
## [1] 19.35054 19.43282 19.90025 19.50295 19.22789
```

You can combine indices to get the elements out a list - like this:

```
# Extract the element named x from the list and then get the third to the fifth elements
myList$x[3:5]
```

```
## [1] -0.3983927  0.8361551 -0.4682876
```

```
# or, the same as
myList[["x"]][3:5]
```

```
## [1] -0.3983927  0.8361551 -0.4682876
```

Adding new elements to a previously created list is also fairly easy:

```
# By position

myList[[4]] <- rnorm(10, 2, 0.5)

myList[[5]] <- rnorm(5, 0.5, 0.01)

print(myList)
```

```
## $x
##  [1]  0.1426445  0.6816563 -0.3983927  0.8361551 -0.4682876 -0.7527596
##  [7] -1.2650036  1.0526486 -1.0461663 -1.0151180
##
## $y
## [1] 19.35054 19.43282 19.90025 19.50295 19.22789
##
## $z
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## [[4]]
##  [1] 2.5514397 2.0934139 0.9645268 1.7443386 2.3830024 1.5319136 2.1143379
##  [8] 2.0113764 2.3616810 2.6963917
##
## [[5]]
## [1] 0.4975418 0.4878161 0.5113728 0.4951817 0.4856347
```

```
# By name (in this case new data will be appended at the end of the list)
myList[["Cities"]] <- c("New York", "Madrid", "Paris")

# this is equal to the $ operator
myList$Cities <- c("New York", "Madrid", "Paris")

print(myList)
```

```
## $x
##  [1]  0.1426445  0.6816563 -0.3983927  0.8361551 -0.4682876 -0.7527596
##  [7] -1.2650036  1.0526486 -1.0461663 -1.0151180
##
## $y
## [1] 19.35054 19.43282 19.90025 19.50295 19.22789
##
## $z
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
##
## [[4]]
##  [1] 2.5514397 2.0934139 0.9645268 1.7443386 2.3830024 1.5319136 2.1143379
##  [8] 2.0113764 2.3616810 2.6963917
##
## [[5]]
## [1] 0.4975418 0.4878161 0.5113728 0.4951817 0.4856347
##
## $Cities
## [1] "New York" "Madrid"   "Paris"
```

Exercise 1

a) Create a list containing two elements named `a` and `b` with the following data:

b) a vector with a integer sequence from to 1 to 5

ii) a matrix with 5 rows and 4 columns filled with a sequence from `1:20`

b) Add a new vector element to the list named `mult` which is the result of multiplying `a` by the first column of `b`

c) Using list and vector indexation access to the third element of `mult`

Exercise 2

Take a look at the (nested) list object below and then run the code chunk to set it in R. Solve the exercises below using this list.

```
nestList <- list(

  x = list(
    a1 = 1:10,
    a2 = rnorm(10)
  ),

  y = list(
        b1 = 1:10,
        b2 = rnorm(10)
  )
)
```

a) Using indexation by name, extract the `x` element from `nestList`

b) Extract the third element of `a1` and the second to fifth elements of `b2`

c) Calculate `a2` times `b1`

Lists are very flexible and, in fact, many outputs from R functions are formatted as *'list-like'* objects (although often these have specific classes). This is one of the reasons why learning to work with lists and know how to access their content is so important.

To check the structure of R objects you can use the function `str()`. Let's see one example based on hypothesis testing from the previous session:

```
iris_vers <- iris$Sepal.Length[iris$Species=="versicolor"]
iris_virg <- iris$Sepal.Length[iris$Species=="virginica"]
```

```
iris_ttest <- t.test(iris_vers, iris_virg)

class(iris_ttest)
```

```
## [1] "htest"
```

```
str(iris_ttest)
```

```
## List of 9
##  $ statistic  : Named num -5.63
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 94
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 1.87e-07
##  $ conf.int   : atomic [1:2] -0.882 -0.422
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num [1:2] 5.94 6.59
##   ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "difference in means"
##  $ alternative: chr "two.sided"
##  $ method     : chr "Welch Two Sample t-test"
##  $ data.name  : chr "iris_vers and iris_virg"
##  - attr(*, "class")= chr "htest"
```

Using `str` you can see that the result of applying a t-Test is an object of class `htest` which is a sort of list with 9 elements (see `?t.test` for more details)

QUICK-EXERCISE 1

Using the internal dataset `airquality` (tested in session #2) calculate the Pearson correlation between `Ozone` and `Temp` to an object named `aq_cor` (apply function `cor.test()` for this).

After performing the calculation, use function `str()` to inspect the `aq_cor` object and extract the correlation and the p-value using list indexation.

User-defined functions

Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting.

In a nutshell a function has the main following components:

- **Name**: What is the name of your function? You can give it any valid object name.
- **Arguments**: What are the inputs to the function? Does it need a vector of numeric data? Or some text?
- **Actions**: What do you want the function to do with the inputs is specified in this part inside the curly brackets `{ }`
- **Output**: a piece of data that is returned by the function as the result of the defined actions

```
# The basic structure of a function
```

```
NAME <- function(ARGUMENTS){

  # ACTIONS ...

  # MORE ACTIONS ...

  return(OUTPUT)
}
```

Let's check some examples:

1) A single input function that will check how many missing values a vector has:

```
count_NA <- function(x){

  return(sum(is.na(x)))
}

count_NA(airquality$Ozone)
```

```
## [1] 37
```

```
count_NA(airquality$Temp)
```

```
## [1] 0
```

2) Make a custom histogram plot

```
# Create a custom histogram function named myHistogram

myHistogram <- function(x, ...) {

  # Create a customized histogram
  hist(x, col = gray(.5, .2), border = "white", ...)

  # Calculate the 95% conf interval of the sample mean
  ci <- t.test(x)$conf.int

  # Define and add top-text
  top.text <- paste(
    "Mean = ", round(mean(x), 2),
    " (95% CI [", round(ci[1], 2),
    ", ", round(ci[2], 2),
    "]), SD = ", round(sd(x), 2),
    sep = "")

  mtext(top.text, side = 3)
}


# Let's call our brand new function on some sample data
# and write the output to a png file

png(filename = "hist.png", res = 300, height = 1500, width = 1800)

myHistogram(airquality$Temp, xlab="Temperature (degrees F)",
            main="Histogram of Temperature\n")
```

```
dev.off()
```

```
## pdf
##   2
```

3) A function to calculate the Normalized Difference Vegetation between the reflectance values of the red and near-infrared bands:

```
ndvi <- function(red, nir){
  return((nir - red) / (nir + red))
}

ndvi(0.1, 0.4)
```

```
## [1] 0.6
```

Exercise 3

a) Make a function called `Celsius2Kelvin` that converts temperature from Celsius to Kelvin (formula: $T(K) = T(°C) + 273.15$). Test it with `a <- 20.5`.

b) Make a function called `Celsius2Fahrenheit` that converts temperature from Celsius to Fahrenheit (formula: $T(°F) = T(°C) \times 9/5 + 32$). Test it with `b <- 16.7`.

c) Make a function named `recode2NA` to change values in an input vector to `NA` if those values are below 10 or above 100. Test it the following vector `v <- c(1, 5, 10, 15, 25, 78, 90, 34, 55, 120, 100, 105, 103, 12, 101)`

d) Write a function called `standardize.me` that takes a vector `x` as an argument, and returns a vector that standardizes the values of `x` (standardization means subtracting the mean and dividing by the standard deviation). Test it with with `d <- rnorm(100, 100, 5)`.

e) Create a function named `CoeffVar` that calculates the Coefficient of Variation (which equals the mean divided by the standard-deviation). Test it with with `e <- rnorm(100)`.

f) Create a function that multiplies the two largest values of a vector and divides them by two (hint: use `sort` for getting the two largest values). Test this function with the following vector `f <- rnorm(1000, 100, 10)`

g) Make a "personalized" version of a histogram with three vertical lines corresponding to each one of the quartiles of the distribution (hint: check `abline` and the `v` argument which is the x-value(s) for vertical line(s)). Test this function with the following vector `g <- rnorm(1000, 10, 3)`

If conditionals

---

if statements

Briefly presented, if statements allow to control the flow of execution of a script.

The conditional if statement is used to test an expression. If the test_expression is `TRUE`, the statement inside the curly brackets gets executed. If it is `FALSE`, nothing happens.

```
# syntax of an if statement

if (test_expression) {
```

```
  # do something here
}
```

Let's see one example of an if conditional used to check if a number is positive

```
x <- 15

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

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

(Obviously) the print message is issued because `x` is in fact positive ;-)

if... else if... else statements

The conditional `if...else` statement is used to test an expression similar to the `if` statement. However, rather than nothing happening if the test_expression is `FALSE`, the else part of the function will be evaluated in sequence. The general structure is like this:

```
if (test_expression_1) {
  # do something
} else if (test_expression_2) {
  # do something else
} else {
  # if nothing happened before.. do this
}
```

Usually if... else statements are more useful within functions. Let's see one example that checks the temperature values:

```
temp_feel <- function(temp){
  if (temp <= 0) {
    "freezing"
  } else if (temp <= 10) {
    "cold"
  } else if (temp <= 20) {
    "cool"
  } else if (temp <= 30) {
    "warm"
  } else {
    "hot"
  }
}

temp_feel(22.5)
```

```
## [1] "warm"
```

Exercise 4

a) Create a function that returns `TRUE` if the input argument `x` is positive and `FALSE` otherwise. Test the function with a positive and a negative value to see if it is working;

b) Create a function that checks if the Spearman correlation between two vectors (`x` and `y`) is higher than $|r| > 0.6$ (absolute value) and returns `TRUE` if it verifies that condition. Use `cor.test()` to calculate the

correlation (hint: use list indexation to extract values from the `cor.test()` output). Test this function using vectors `Ozone` and `Temp` from the `airquality` dataset;

c) Also using `cor.test()` function create a functions that checks if the p-value is:

- = 0.1 return n.s. (i.e., non-significant);
- < 0.1 returns -;
- < 0.05 returns *;
- < 0.01 returns **;
- < 0.001 returns ***;

(hint: also use here list indexation) Test this function using vectors `Ozone` and `Temp` from the `airquality` dataset.

For loops

A for loop is used to iterate through the elements of R objects.

It is used to execute repetitive code statements for a particular number of times allowing to automate certain tasks.

The general syntax is provided below where `i` is the counter and as `i` assumes each sequential value defined the code in the body will be performed for that `i`-th value.

```
# syntax of for loop

for(i in 1:n) {

  # <do stuff here with i>
}
```

For example, the following `for` loop iterates through each value (2015, . . . , 2018) and performs the `paste` and `print` functions inside the curly brackets.

```
for (i in 2015:2018){
  output <- paste("The year is", i)
  print(output)
}
```

```
## [1] "The year is 2015"
## [1] "The year is 2016"
## [1] "The year is 2017"
## [1] "The year is 2018"
```

Exercise 5

a) Do a function that prints the mean and the standard-deviation for each column of an input matrix or dataframe. Use the `airquality` dataset to test it.

b) Create a function that takes a **vector** (named `x`) and a **matrix** (named `y`) to calculate the correlation and prints the results **for each column** in the matrix. Use function `cor.test` to calculate the Pearson correlation. If the correlation is higher than |rho| > thresh (an input variable to be defined as a function argument) then show the result using the `print` function.

For testing the function, use `airquality$Ozone` as `x` and a matrix with all of the remaining columns from `airquality` as `y`. Set the thresh value to 0.6.