Practical 114

Stefano De Sabbata

2020-12-17

R programming

Stefano De Sabbata

R Scripts

The RStudio Console is handy to interact with the R interpreter and obtain results of operations and commands. However, moving from simple instructions to an actual program or scripts to conduct data analysis, the Console is usually not sufficient anymore. In fact, the Console is not a very comfortable way of providing long and complex instructions to the interpreter and editing past instructions when you want to change something. A better option to create programs or data analysis script of any significant size is to use the RStudio integrated editor to create an *R script*.

To create an R script, select from the top menu *File* > *New File* > *R Script*. That opens the embedded RStudio editor and a new empty R script folder. Copy the two lines below into the file. The first loads the `tidyverse` library, whereas the second simply calculates the square root of two.

```
# Load the Tidyverse
library(tidyverse)

# Calculate the square root of two
2 %>% sqrt()
```

```
## [1] 1.414214
```

From the top menu, select *File* > *Save*, type in *My_first_script.R* (make sure to include the underscore and the *.R* extension) as *File name*, and click *Save*. That is your first R script, congratulations!

New lines of code can be added to the file, and the whole script can then be executed. Edit the file by adding the line of code shown below, and save it. Then click the *Source* button on the top-right of the editor to execute the file. What happens the first time? What happens if you click *Source* again?

```
# First variable in a script
a_variable <- "This is my first script"
```

Alternatively, you can click on a specific line or select one or more lines, and click *Run* to execute only the selected line(s).

Delete the two lines calculating the square root of two and defining the variable `a_variable` from the script, leaving only the line loading the Tidyverse library. In the following sections, add the code to the script to execute it, rather than using the Console.

Vectors

Vectors can be defined in R by using the function `c`, which takes as parameters the items to be stored in the vector – stored in the order in which they are provided.

```
east_midlands_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")
length(east_midlands_cities)
```

```
## [1] 4
```

Once the vector has been created and assigned to an identifier, elements within the vector can be retrieved by specifying the identifier, followed by square brackets, and the *index* (or indices as we will see further below) of the elements to be retrieved – remember that indices start from 1.

```
# Retrieve the third city
east_midlands_cities[3]
```

```
## [1] "Lincoln"
```

To retrieve any subset of a vector (i.e., not just one element), specify an integer vector containing the indices of interest (rather than a single integer value) between square brackets.

```
# Retrieve first and third city
east_midlands_cities[c(1, 3)]
```

```
## [1] "Derby"   "Lincoln"
```

The operator `:` can be used to create integer vectors, starting from the number specified before the operator to the number specified after the operator.

```
# Create a vector containing integers between 2 and 4
two_to_four <- 2:4
two_to_four
```

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

```
# Retrieve cities between the second and the fourth
east_midlands_cities[two_to_four]
```

```
## [1] "Leicester"  "Lincoln"    "Nottingham"
```

```
# As the second element of two_to_four is 3...
two_to_four[2]
```

```
## [1] 3
```

```
# the following command will retrieve the third city
east_midlands_cities[two_to_four[2]]
```

```
## [1] "Lincoln"
```

```
# Create a vector with cities from the previous vector
selected_cities <- c(east_midlands_cities[1], east_midlands_cities[3:4])
```

The functions `seq` and `rep` can also be used to create vectors, as illustrated below.

```
seq(1, 10, by = 0.5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
## [16]  8.5  9.0  9.5 10.0
```

```
seq(1, 10, length.out = 6)
```

```
## [1]  1.0  2.8  4.6  6.4  8.2 10.0
```

```r
rep("Ciao", 4)
```

```
## [1] "Ciao" "Ciao" "Ciao" "Ciao"
```

The logical operators `any` and `all` can be used to test conditional statements on the vector. The former returns `TRUE` if at least one element satisfies the statement, the second returns `TRUE` if all elements satisfy the condition

```r
any(east_midlands_cities == "Leicester")
```

```
## [1] TRUE
```

```r
my_sequence <- seq(1, 10, length.out = 7)
my_sequence
```

```
## [1]  1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

```r
any(my_sequence > 5)
```

```
## [1] TRUE
```

```r
all(my_sequence > 5)
```

```
## [1] FALSE
```

All built-in numerical functions in R can be used on a vector variable directly. That is, if a vector is specified as input, the selected function is applied to each element of the vector.

```r
one_to_ten <- 1:10
one_to_ten
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
one_to_ten + 1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```r
sqrt(one_to_ten)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

Filtering

As seen in the first practical session, a conditional statement entered in the Console is evaluated for the provided input, and a logical value (`TRUE` or `FALSE`) is provided as output. Similarly, if the provided input is a vector, the conditional statement is evaluated for each element of the vector, and a vector of logical values is returned – which contains the respective results of the conditional statements for each element.

```r
minus_three <- -3
minus_three > 0
```

```
## [1] FALSE
```

```r
minus_three_to_three <- -3:3
minus_three_to_three
```

```
## [1] -3 -2 -1  0  1  2  3
```

```r
minus_three_to_three > 0
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

A subset of the elements of a vector can also be selected by providing a vector of logical values between brackets after the identifier. A new vector returned, containing only the values for which a `TRUE` value has been specified correspondingly.

```
minus_two_to_two <- -2:2
minus_two_to_two
```

```
## [1] -2 -1  0  1  2
```

```
minus_two_to_two[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
```

```
## [1] -2 -1  2
```

As the result of evaluating the conditional statement on a vector is a vector of logical values, this can be used to filter vectors based on conditional statements. If a conditional statement is provided between square brackets (after the vector identifier, instead of an index), a new vector is returned, which contains only the elements for which the conditional statement is true.

```
minus_two_to_two > 0
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

```
minus_two_to_two[minus_two_to_two > 0]
```

```
## [1] 1 2
```

Conditional statements

Conditional statements are fundamental in (procedural) programming, as they allow to execute or not execute part of a procedure depending on whether a certain condition is true. The condition is tested and the part of the procedure to execute in the case the condition is true is included in a *code block*.

```
temperature <- 25

if (temperature > 25) {
  cat("It really warm today!")
}
```

A simple conditional statement can be created using `if` as in the example above. A more complex structure can be created using both `if` and `else`, to provide not only a procedure to execute in case the condition is true, but also an alternative procedure, to be executed when the condition is false.

```
temperature <- 12

if (temperature > 25) {
  cat("It really warm today!")
} else {
  cat("Today is not warm")
}
```

```
## Today is not warm
```

Finally, conditional statements can be **nested**. That is, a conditional statement can be included as part of the code block to be executed after the condition is tested. For instance, in the example below, a second conditional statement is included in the code block to be executed in the case the condition is false.

```
temperature <- -5

if (temperature > 25) {
  cat("It really warm today!")
} else {
```

```
  if (temperature > 0) {
    cat("There is a nice temperature today")
  } else {
    cat("This is really cold!")
  }
}
```

```
## This is really cold!
```

Similarly, the first example seen in the lecture should be coded as follows.

```
a_value <- -7

if (a_value == 0) {
  cat("Zero")
} else {
  if (a_value < 0) {
    cat("Negative")
  } else {
    cat("Positive")
  }
}
```

```
## Negative
```

Loops

Loops are another core component of (procedural) programming and implement the idea of solving a problem or executing a task by performing the same set of steps a number of times. There are two main kinds of loops in R - **deterministic** and **conditional** loops. The former is executed a fixed number of times, specified at the beginning of the loop. The latter is executed until a specific condition is met. Both deterministic and conditional loops are extremely important in working with vectors.

Conditional Loops

In R, conditional loops can be implemented using `while` and `repeat`. The difference between the two is mostly syntactical: the first tests the condition first and then execute the related code block if the condition is true; the second executes the code block until a `break` command is given (usually through a conditional statement).

```
a_value <- 0
# Keep printing as long as x is smaller than 2
while (a_value < 2) {
  cat(a_value, "\n")
  a_value <- a_value + 1
}
```

```
## 0
## 1
```

```
a_value <- 0
# Keep printing, if x is greater or equal than 2 than stop
repeat {
  cat(a_value, "\n")
  a_value <- a_value + 1
  if (a_value >= 2) break
}
```

```
## 0
## 1
```

Deterministic Loops

The deterministic loop executes the subsequent code block iterating through the elements of a provided vector. During each iteration (i.e., execution of the code block), the current element of the vector (in the definition below) is assigned to the variable in the statement (in the definition below), and it can be used in the code block.

```
for (<VAR> in <VECTOR>) {
    ... code in loop ...
    }
```

It is, for instance, possible to iterate over a vector and print each of its elements.

```
east_midlands_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")
for (city in east_midlands_cities){
  cat(city, "\n")
}
```

```
## Derby
## Leicester
## Lincoln
## Nottingham
```

It is common practice to create a vector of integers on the spot (e.g., using the `:` operator) to execute a certain sequence of steps a pre-defined number of times.

```
for (iterator in 1:3) {
  cat("Exectuion number", iterator, ":\n")
  cat("    Step1: Hi!\n")
  cat("    Step2: How is it going?\n")
}
```

```
## Exectuion number 1 :
##     Step1: Hi!
##     Step2: How is it going?
## Exectuion number 2 :
##     Step1: Hi!
##     Step2: How is it going?
## Exectuion number 3 :
##     Step1: Hi!
##     Step2: How is it going?
```

Exercise 114.1

Question 114.1.1: Use the modulo operator `%%` to create a conditional statement that prints `"Even"` if a number is even and `"Odd"` if a number is odd.

Question 114.1.2: Encapsulate the conditional statement written for *Question 114.1.1* into a `for` loop that executes the conditional statement for all numbers from 1 to 10.

Question 114.1.3: Encapsulate the conditional statement written for *Question 114.1.1* into a `for` loop that prints the name of cities in odd positions (i.e., first, third, fifth) in the vector `c("Birmingham", "Derby", "Leicester", "Lincoln", "Nottingham", "Wolverhampton")`.

Question 114.1.4: Write the code necessary to print the name of the cities in the vector `c("Birmingham", "Derby", "Leicester", "Lincoln", "Nottingham", "Wolverhampton")` as many times as their position

in the vector (i.e., once for the first city, two times for the second, and so on and so forth).

Function definition

Recall from the lecture that *an* **algorithm** *or effective procedure is a mechanical rule, or automatic method, or programme for performing some mathematical operation* (Cutland, 1980). A **program** is a specific set of instructions that implement an abstract algorithm. The definition of an algorithm (and thus a program) can consist of one or more **function**s, which are sets of instructions that perform a task, possibly using an input, possibly returning an output value.

The code below is a simple function with one parameter. The function simply calculates the square root of a number. Add the code below to your script and run that portion of the script (or type the code into the Console).

```
cube_root <- function (input_value) {
   result <- input_value ^ (1 / 3)
   result
}
```

Once the definition of a function has been executed, the function becomes part of the environment, and it should be visible in the Environment panel, in a subsection titled *Functions*. Thereafter, the function can be called from the Console, from other portions of the script, as well as from other scripts.

If you type the instruction below in the *Console*, or add it to the script and run it, the function is called using `27` as an argument, thus returning `3`.

```
cube_root(27)
```

```
## [1] 3
```

Functions and control structures

One issue when writing functions is making sure that the data that has been given to the data is the right kind. For example, what happens when you try to compute the cube root of a negative number?

```
cube_root(-343)
```

```
## [1] NaN
```

That probably wasn't the answer you wanted. As you might remember `NaN` (*Not a Number*) is the value return when a mathematical expression is numerically indeterminate. In this case, this is actually due to a shortcoming with the `^` operator in R, which only works for positive base values. In fact -7 is a perfectly valid cube root of -343, since (-7)x(-7)x(-7) = -343.

To work around this limitation, we can state a conditional rule:

- If `x < 0`: calculate the cube root of x 'normally'.
- Otherwise: work out the cube root of the positive number, then change it to negative.

Those kinds of situations can be dealt with in an R function by using an `if` statement, as shown below. Note how the operator `-` (i.e., the symbol minus) is here used to obtain the inverse of a number, in the same way as `-1` is the inverse of the number `1`.

```
cube_root <- function (input_value) {
    if (input_value >= 0){
        result <- input_value^(1 / 3)
    }else{
        result <- -( (-input_value)^(1/3) )
    }
    result
```

```
}

cube_root(343)
cube_root(-343)
```

However, other things can go wrong. For example, `cube_root("Leicester")` would cause an error to occur, `Error in x^(1 / 3) : non-numeric argument to binary operator`. That shouldn't be surprising because cube roots only make sense for numbers, not character variables. Thus, it might be helpful if the cube root function could spot this and print a warning explaining the problem, rather than just crashing with a fairly cryptic error message such as the one above, as it does at the moment.

The function could be re-written to making use of `is.numeric` in a second conditional statement. If the input value is not numeric, the function returns the value `NA` (*Not Available*) instead of a number. Note that here there is an `if` statement inside another `if` statement, as it is always possible to nest code blocks – and `if` within a `for` within a `while` within an `if` within ... etc.

```
cube_root <- function (input_value) {
    if (is.numeric(input_value)) {
        if (input_value >= 0){
            result <- input_value^(1/3)
        }else{
            result <- -(-input_value)^(1/3)
        }
        result
    }else{
        cat("WARNING: Input variable must be numeric\n")
        NA
    }
}
```

Finally, `cat` is a printing function, that instructs R to display the provided argument (in this case, the phrase within quotes) as output in the Console. The `\n` in `cat` tells R to add a *newline* when printing out the warning.

Exercise 114.2

Question 114.2.1: Write a function that calculates the areas of a circle, taking the radius as the first parameter.

Question 114.2.2: Write a function that calculates the volume of a cylinder, taking the radius of the base as the first parameter and the height as the second parameter. The function should call the function defined above and multiply the returned value by the height to calculate the result.

Question 114.2.3: Write a function with two parameters, a vector of numbers and a vector of characters (text). The function should check that the input has the correct data type. If all the numbers in the first vector are greater than zero, return the elements of the second vector from the first to the length of the first vector.