Practical 114

Stefano De Sabbata

2020-10-01

R programming

Stefano De Sabbata

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 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
```

```
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

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

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

```
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
```

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

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

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

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

Filtering

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.

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

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

```
one_to_ten + 1
```

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

```
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
```

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.

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

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

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

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

```
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 structures

Conditional structures 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 structure 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.

```r
temperature <- 12

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

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

Finally, conditional structures can be **nested**. That is, a conditional structure 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 structure is included in the code block to be executed in the case the condition is false.

```r
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!
```

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).

```r
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
```

```r
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?
```

Function definition

Recall from the first 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.

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

Functions can be defined by typing the definition in the Console in RStudio. However, entering functions from the command line is not always very convenient. If you make a typing error in an early line of the definition, it isn't possible to go back and correct it. You would have to type in the definition every time you used R A more sensible approach is to type the function definition into an R script.

Create a new R project for this practical, named *Practical_111*. Create a new R script named `functions_Practical_111.R`. Copy the definition of `cube_root` in the R script, and save the file. If you execute the script, the R interpreter creates the new function from its definition, which should then be visible in the *Environment* tab in RStudio.

If you type the instruction below in the *Console*, the function is called using `27` as an argument, thus returning `3`.

```
cube_root(27)
```

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

It is furthermore possible to load the function(s) defined in one script from another script – in a fashion similar to when a library is loaded. Create a new R script as part of the *Practical_111* project, named `main_Practical_111.R` and copy the code below in that second R script and save the file.

```
source("functions_Practical_111.R")

cube_root(27)
```

Executing the `main_Practical_111.R` instructs the interpreter first to run the `functions_Practical_111.R` script, thus creating the `cube_root` function, and then invoke the function using `27` as an argument, thus returning again `3`. That is a simple example, but this can be an extremely powerful tool to create your own library of functions to be used by different scripts.

Exercise 6.1

Extend the code in the script `functions_Practical_111.R` to include the code necessary to solve the questions below.

Question 6.1.1: Write a function that calculates the areas of a circle, taking the radius as the first parameter.

Question 6.1.2: 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 6.1.3: 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.