Creating Functions

David García Callejas

Functions are the backbone of R, to the extent that any operation involving any variable is regarded as a function over one or more objects. There are many situations in which creating specific functions can be useful or efficient. In particular, containing code into functions will help us apply a divide-and-conquer strategy, whereby splitting a large problem in smaller tasks facilitates both their understanding and their inspection. It will also make our code more easy to share and readable.

The general syntax for using functions in R is the well known `function(argument1,argument2,...)` construct. Let's see how can we create one from scratch.

```
my_function <- function(argument){
  # code
  return()
}
```

In this example, we define a function called `my_function` with a single argument. We will place the code inside the definition, and the last line will tell R what do we want to return as the function's output.

As a first example, we can create a function to normalise a numeric vector with respect to its maximum value:

```
v1 <- sample(1:100,10,FALSE)
max(v1)

# this is the code we want to include in our functino
v2 <- v1/max(v1)

normalise_vec <- function(v = NULL){
  res <- v/max(v)
  return(res)
}

v3 <- normalise_vec(v1)
identical(v2,v3)
```

It is good practice for each function we define to be stored in a separate file. The name of the file should be equal to the name of the function (in our not-very-original example, it would be "normalise_vec.R"), and it should only include that function. Once we create and store that file, we can seamlessly use that function in any script by loading it in our R session in a similar way to what we do with external packages.

```
source("normalise_vec.R")
```

where, depending on our work directory and folder structure, we may specify the specific path to our file "normalise_vec.R". For example, if you are working within a Rstudio project, it could be worth to keep functions in a specific folder, for example "R/functions/..".

A function may accept as many arguments as necessary. We can specify default values for any argument, so that the user doesn't need to specify them is the default values are ok. If we do not specify such default values, calling the function will raise an error if we don't assign a value to every argument in it. In the following example, `arg1` and `arg3` have default values that we can overwrite, while `arg2` does not.

```r
argument_test <- function(arg1 = NULL, arg2, arg3 = 0){
  cat("arg1:",arg1,"\narg2:",arg2,"\narg3:",arg3)
}

argument_test(arg1 = 1,arg2 = 1,arg3 = 1)
argument_test(arg2 = 1)
# argument_test()
```

A key part in the development of functions is the documentation of their process, results, and arguments. You already know that every function we use in R has an associated documentation that can be accessed via the syntax `?function_name`. Ideally, we should aspire to document our own functions in the same thorough and complete way as the functions we see in base R... although sometimes this documentation looks like it's been written in a lost and forgotten language!). So, perhaps we should aspire to do better - if you think about it, the main objective of the documentation is to ease the use of a given function. In any case, we can easily replicate the standard documentation in our own functions, using the `roxygen2` package. In Rstudio, if we have the cursor inside the definition of a given function (i.e. in the `function(...)` part), we can clik on the menu tab `Code/Insert Roxygen Skeleton`. This action will insert a series of lines similar to this:

```r
#' Title
#'
#' @param v
#'
#' @return
#' @export
#'
#' @examples
normalise_vec.R <- function(v = NULL){
  res <- v/max(v)
  return(res)
}
```

These lines contain the basic slots that we need to document in our functions. As you can see, first of all they share a similar structure with standard R comments, but adding an apostrophe to the hash symbol. The first line is the title of our function. Below, in another line(s), we may include an extended description of the function. The arguments of the function are documented in the `@param` fields. Likewise, the object returned is explained in the `@return` field. The `@export` field can be left blank, it is simply used as a flag for indicating that this function should be available when loaded in memory. In the `@examples` field we may include, yes, you guessed it, examples of applying our function. All in all, a first take at documenting a function could look like this:

```r
#' Normalise a numeric vector
#'
#' Normalise a numeric vector with regards to the maximum value contained in it.
#'
#' @param v Numeric vector of arbitrary length
#'
#' @return Numeric vector of the same length as v, normalised in the range 0-1.
#' @export
#'
#' @examples
#' v1 <- runif(10,0,100)
#' vnorm <- normalise_vec(v1)
normalise_vec <- function(v = NULL){
  res <- v/max(v)
  return(res)
```

```
}
```

If our functions use, in turn, functions from other packages, we should call them using the notation package::function. See the following example, in which we define a function to return the number of words with more than N characters in a string.

```
#' Number of words with length higher than a given value
#'
#' @param v string or vector of strings
#' @param l numeric; number of characters in a given word
#'
#' @return numeric vector, number of words of length > l in each element.
#' @export
#'
#' @examples
#' v1 <- c("this string has an outstanding diversity in word lengths",
#' "this one does not")
#' v2 <- num_words(v1,6)
num_words <- function(v = NULL, l = 0){

  # create the object to be returned by the function
  res <- NULL

  if(!is.null(v)){
    #https://stackoverflow.com/questions/33226616/
    #how-to-remove-words-of-specific-length-in-a-string-in-r
    my.regex <- paste("\\w{",l,",}",sep="")
    # select words with length > l
    words <- stringr::str_extract_all(v, mi.regex)
    # counts them
    res <- sapply(words,length)
  }
  return(res)
}
```