Functions in R

Dr. Noah Mutai

1 Introduction

A function is a set of statements organized together to perform a specific task.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

R has a large number of in-built functions and the user can create their own functions.

```
min(), max(), mean(), median() - return the minimum / maximum / mean /
median value of         a numeric vector, correspondingly
sum() - returns the sum of a numeric vector
range() - returns the minimum and maximum values of a numeric vector
abs() - returns the absulute value of a number
str() - shows the structure of an R object
print() - displays an R object on the console
ncol() - returns the number of columns of a matrix or a dataframe
length() - returns the number of items in an R object (a vector, a list, etc.)
nchar() - returns the number of characters in a character object
sort() - sorts a vector in ascending or descending (decreasing=TRUE) order
exists() - returns TRUE or FALSE depending on whether or not a variable
is defined in the R environment
```

1.1 Example

```
vector <- c(3, 5, 2, 3, 1, 4)
print(min(vector))
print(mean(vector))
print(median(vector))
print(sum(vector))
print(range(vector))
print(str(vector))
print(length(vector))
print(sort(vector, decreasing=TRUE))
print(exists('vector'))  ## note the quotation marks
```

1.2 Creating a Function in R

While applying built-in functions facilitates many common tasks, often we need to create our own function to automate the performance of a particular task.

To declare a user-defined function in R, we use the keyword function.

The syntax is as follows:

function_name <- function(parameters){ function body }

Above, the main components of an R function are:

```
function name,
function parameters, and
function body.
```

Let's take a look at each of them separately.

1.2.1 Function Name

This is the name of the function object that will be stored in the R environment after the function definition and used for calling that function.

It should be concise but clear and meaningful so that the user who reads our code can easily understand what exactly this function does.

For example, if we need to create a function for calculating the circumference of a circle with a known radius, we'd better call this function circumference rather than function_1 or circumference_of_a_circle.

1.3 Function Parameters

Sometimes, they are called formal arguments.

Function parameters are the variables in the function definition placed inside the parentheses and separated with a comma that will be set to actual values (called arguments) each time we call the function. For example:

```
circumference <- function(r){
    2*pi*r
}
print(circumference(2))
```

Above, we created a function to calculate the circumference of a circle with a known radius using the formula , so the function has the only parameter r.

After defining the function, we called it with the radius equal to 2 (hence, with the argument 2).

It's possible, even though rarely useful, for a function to have no parameters:

```
hello_world <- function(){
    'Hello, World!'
}
print(hello_world())
```

Also, some parameters can be set to default values (those related to a typical case) inside the function definition, which then can be reset when calling the function.

Returning to our circumference function, we can set the default radius of a circle as 1, so if we call the function with no argument passed, it will calculate the circumference of a unit circle (i.e., a circle with a radius of 1).

Otherwise, it will calculate the circumference of a circle with the provided radius:

```
circumference <- function(r=1){
    2*pi*r
}
print(circumference())
print(circumference(2))
```

1.4 Function Body

The function body is a set of commands inside the curly braces that are run in a predefined order every time we call the function.

In other words, in the function body, we place what exactly we need the function to do:

```
sum_two_nums <- function(x, y){
    x + y
}
print(sum_two_nums(1, 2))
```

It's possible to drop the curly braces if the function body contains a single statement.

For example:

```
sum_two_nums <- function(x, y) x + y
print(sum_two_nums(1, 2))
```

1.5 Return Values

To let a function return a result, use the return() function:

```r
mean_median <- function(vector){
    mean <- mean(vector)
    median <- median(vector)
    return(c(mean, median))
}
print(mean_median(c(1, 1, 1, 2, 3)))
```

1.6 Calling a Function in R

In all the above examples, we actually already called the created functions many times.

To do so, we just put the punction name and added the necessary arguments inside the parenthesis.

In R, function arguments can be passed by position, by name (so-called named arguments), by mixing position-based and name-based matching, or by omitting the arguments at all.

If we pass the arguments by position, we need to follow the same sequence of arguments as defined in the function:

```r
subtract_two_nums <- function(x, y){
    x - y
}
print(subtract_two_nums(3, 1))
```

```r
subtract_two_nums <- function(x, y){
    x - y
}
print(subtract_two_nums(x=3, y=1))
print(subtract_two_nums(y=1, x=3))
```

Since we explicitly assigned x=3 and y=1, we can pass them either as x=3, y=1 or y=1, x=3 – the result will be the same.

1.7 Mixing position and names

It's possible to mix position - and name-based matching of the arguments.

Let's look at the example of the function for calculating BMR (basal metabolic rate), or daily consumption of calories, for women based on their weight (in kg), height (in cm), and age (in years).

The formula that will be used in the function is the Mifflin-St Jeor equation:

```
calculate_calories_women <- function(weight, height, age){
    (10 * weight) + (6.25 * height) - (5 * age) - 161
}
```

Now, let's calculate the calories for a woman 30 years old, with a weight of 60 kg and a height of 165 cm. However, for the age parameter, we'll pass the argument by name and for the other two parameters, we'll pass the arguments by position:

```
print(calculate_calories_women(age=30, 60, 165))
```

In the case like above (when we mix matching by name and by position), the named arguments are extracted from the whole succession of arguments and are matched first, while the rest of the arguments are matched by position, i.e., in the same order as they appear in the function definition. However, this practice isn't recommended and can lead to confusion.

Finally, we can omit some (or all) of the arguments at all. This can happen if we set some (or all) of the parameters to default values inside the function definition. Let's return to our calculate_calories_women function and set the default age of a woman as 30 y.o.:

```
calculate_calories_women <- function(weight, height, age=30){
    (10 * weight) + (6.25 * height) - (5 * age) - 161
}
print(calculate_calories_women(60, 165))
```

1.8 Using Functions Inside Other Functions

Inside the definition of an R function, we can use other functions.

We've already seen such an example earlier, when we used the built-in mean() and median() functions inside a user-defined function mean_median:

```
mean_median <- function(vector){
    mean <- mean(vector)
    median <- median(vector)
    return(c(mean, median))
}
```

It's also possible to pass the output of calling one function directly as an argument to another function:

```
radius_from_diameter <- function(d){
    d/2
    circumference <- function(r){
    2*pi*r
}
```

```r
print(circumference(radius_from_diameter(4)))
}
```

1.9 Nested functions

Functions can be nested, meaning that we can define a new function inside another function.

Let's say that we need a function that sums up the circle areas of 3 non-intersecting circles:

```r
sum_circle_ares <- function(r1, r2, r3){
    circle_area <- function(r){
        pi*r^2
    }
    circle_area(r1) + circle_area(r2) + circle_area(r3)
}

print(sum_circle_ares(1, 2, 3))
```

Above, we defined the circle_area function inside the sum_circle_ares function.

We then called that inner function three times (circle_area(r1), circle_area(r2), and circle_area(r3)) inside the outer function to calculate the area of each circle for further summing up those areas.

Now, if we try to call the circle_area function outside the sum_circle_ares function, the program throws an error, because the inner function exists and works only inside the function where it was defined:

```r
# print(circle_area(10))
```

Error in circle_area(10): could not find function "circle_area" Traceback:

1. print(circle_area(10)) When nesting functions, we have to keep in mind two things:
2. Similar to creating any function, the inner function is supposed to be used at least 3 times inside the outer function. Otherwise, it isn't viable to create it.
3. If we want to be able to use the function independent of the bigger function, we should create it outside the bigger function instead of nesting these functions.

For example, if we were going to use the circle_area function outside the sum_circle_ares function, we would write the following code:

1.10 Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

Create a function with arguments.

```
new.function <- function(a, b) {
   print(a^2)
   print(a)
   print(b)
}
```

Evaluate the function without supplying one of the arguments.y

```
# new.function(6)
```

2 Summary

We learned quite a few aspects related to functions in R. In particular, we discussed the following:

```
(a) Types of functions in R
(b) Why and when we would need to create a function
(c) Some of the most popular built-in functions in R and what they are used for
(d) How to define a user-defined function
(e) The main components of a function
(f) The best practices for naming a function
(g) When and how to set function parameters to default values
(h) The function body and the nuances of its syntax
(i) When we should explicitly include the return statement in
the function definition.
(j) How to call an R function with named, positional, or mixed arguments
(k) What happens when we mix positional and named arguments - and why this
isn't a good practice.
(l) When we can omit some (or all) of the arguments
(m) How to apply functions inside other functions
(n) How to pass a function call as an argument to another function
(o) When and how to nest functions
```