Chapter 4 - Writing Functions (part 1)

1 Aims of the Chapter

By the end of this chapter you should know

- how to write your own functions
- how to specify default argument values
- how to return multiple objects from a function
- how to use pipes

2 Basic elements of a function

A function is just another R object. They are easy to create. They have similarities with mathematical functions: they take a set of inputs (data, numbers, character strings etc) and generally return something.

2.1 Example function 1

Let’s look at a simple example. Suppose we want to create a function to square the input value.

```
square_it <- function(x){
  squared_value <- x ^ 2
  return(squared_value)
}
```

The code to create our `sqaure_it` function consisted of

- a function name (`sqaure_it`)
- the keyword `function` (to identify it as a function)
- some argument names inside `( )` separated by commas if there are more than one (we just have one argument `x`)
- the R code inside `{}`
- `return` on the last line which tells R what you want to return from the function

2.2 Running your function

You can run your function by typing the function name with the function arguments inside `()`

```
square_it(x = 5)
```

```
## [1] 25
```

You have to use the argument names that your function expects. Our `square_it` function has an argument `x` so

```
square_it(y = 5)
```

gives an error

Error in square_it(y = 5) : unused argument (y = 5)

2.3 Example function 2

Below is a function to determine the proportion of elements in a vector that are greater than some user-defined threshold

```
calculate_proportion <- function(data_vector, threshold){
  more_than_threshold <- data_vector > threshold
  values_in_interval <- data_vector[more_than_threshold]
  proportion_in_interval <- length(values_in_interval) / length(data_vector)
  return(proportion_in_interval)
}
```

The code

1. creates a **logical** vector of `TRUE` or `FALSE` values called `more_than_threshold` . An element of `more_than_threshold` is `TRUE` if the corresponding element of `data_vector` is more than `threshold`
2. uses the subsetting operator `[]` and the logical vector `more_than_threshold` to **extract** the elements of `data_vector` that are more than `threshold`
3. calculates the proportion by dividing the number of elements in the `values_in_interval` vector by the number of elements in `data_vector`

Again we can see the function code consists of

- a function name (`calculate_proportion`)
- the keyword `function` (to identify it as a function)
- argument names inside `( )` separated by commas (our arguments are `data_vector` and `threshold`)
- the R code inside `{}`

- `return` on the last line specifying the object to return

To use the function we need to supply it with a vector of numbers. We can create this vector in the Workspace and use this as an input to the function

```
some_data <- rnorm(100, mean = 6, sd = 3)
calculate_proportion(data_vector = some_data, threshold = 2)
```

```
## [1] 0.9
```

If we decrease the mean of the normal distribution used to create our normal realisations we would expect the proportion of realisations greater than 2 to decrease

```
some_data <- rnorm(100, mean = 4, sd = 3)
calculate_proportion(data_vector = some_data, threshold = 2)
```

```
## [1] 0.72
```

```
some_data <- rnorm(100, mean = 2, sd = 3)
calculate_proportion(data_vector = some_data, threshold = 2)
```

```
## [1] 0.46
```

```
some_data <- rnorm(100, mean = 0, sd = 3)
calculate_proportion(data_vector = some_data, threshold = 2)
```

```
## [1] 0.29
```

> Try this
>
> Modify the function `calculate_proportion` so that it calculates the proportion of the supplied data that are in an interval specified by the user. The interval should have an upper **and** a lower threshold. Hint: You may need to use some of the Boolean operators described in Chapter 3. Call your new function `calculate_proportion_in_interval`

> Try this
>
> Further modify your `calculate_proportion_in_interval` function so that it simulates realisations from a normal distribution **inside** the function. You should allow the user to specify the number of realisations, and the mean and standard deviation of the normal distribution. Call this function `calculate_proportion_in_interval_from_normal` . Check it with some different argument values (how can you check that your function is working correctly using `pnorm` ?)

2.4 Example function 3

In our next function we will use the existing function `which` so let's first see what this function does

```
some_nums <- c(1, 3, 5, 7, 4, -2, 6)
which(some_nums <0 )
```

```
## [1] 6
```

```
which(some_nums > 3.5)
```

```
## [1] 3 4 5 7
```

So `which` returns the **indices** of the `TRUE` elements of the logical vector supplied as the function argument.

Suppose we wanted to write a function that allows the user to remove all elements of a numeric vector that are below some user-supplied threshold. We could do this as follows

```
remove_low_values <- function(x, threshold){
  low_values_indices <- which(x < threshold)
  return(x[-low_values_indices])
}
```

The function has two arguments, `x` and `threshold` and these need to be supplied by the user when the function is called.

Looking at the code we see that in the function

- we create the object `low_values_indices` which contains the **indices** of the elements of `x` which are less than `threshold`
- we use subsetting to remove the elements with these indices
- those that are not removed get returned

We call the function as follows

```
remove_low_values(x = c(1, -6, 9, 224, 67, 3, 89), threshold = 50)
```

```
## [1] 224  67  89
```

The values below 50 have been removed from the numeric vector `x` as required.

Note the difference between

```
remove_low_values
```

```
## function(x, threshold){
##   low_values_indices <- which(x < threshold)
##   return(x[-low_values_indices])
## }
```

which shows the function code and

```
remove_low_values(x = c(1, -6, 9, 224, 67, 3, 89), threshold = 50)
```

```
## [1] 224  67  89
```

which runs the function code.

What happens if we don't supply a threshold value as a function argument?

```
remove_low_values(x = c(1, -6, 9, 224, 67, 3, 89))
```

We get the error message

Error in which(x < threshold) : argument ``threshold" is missing, with no default

So do we always have to supply a value for all the arguments in a function we create? The answer is no, if some arguments were given default values when the function was defined.

For example consider the `dnorm` function. When we use `dnorm(2)` we are calculating the probability density evaluated at $X = 2$, where $X \sim N(0, 1)$. The mean and sd have **default** values of 0 and 1 so we didn't need to specify them.

3 A style guide for functions

Wickham (Wickham 2015) suggests several conventions associated with functions. There are:

- `{` should never appear on its own in a line
- `{` should always be followed by a new line
- `}` should always appear on its own line unless it is followed by `else` (I'll mention `else` in later chapters)
- code inside `{ }` should be indented by 2 spaces
- if the arguments of a function go over more than one line, each line or arguments should be indented to match the first

Luckily, if you create functions in *RStudio* then most of the above is implemented automatically.

4 Exercises A

1. The following is poorly formatted and incorrect code (it won't run). Correct the code and and format it according to the style guidelines (you may need to revisit the Style Guide in Chapter 2). It is supposed to return the highest value of a vector raised to a given power.

```
highest. value_to.a_power= function (x ,gamma )
{
  y <- max{x}^ gamma
  return(x)}
```

- Try your function on the vector `x <- c(2, 6, -7)` with `gamma = 2`
- Use your function to calculate the maximum of the square roots of 100 realisations of a random variable with a unif(0, 10) distribution
- Modify your function so that it returns the largest absolute value raised to a power `gamma` and check it on `x <- c(2, 6, -7)` with `gamma = 2`

2. Write a function to return all values of a vector that are more than 2 or less than -3
3. Write a function to return the **second** smallest value in a vector `x` (hint: you may find the `sort` function useful). Check your function on some vectors of your choice (perhaps use `rnorm` to create some vectors of numbers).
4. Write a function to return the n^{th} smallest value in a vector `x` (where n is user-supplied). Check your function on some vectors you create.
5. (A harder question)

- Generate 50 co-ordinates (x, y) with $x \in [0, 1]$ and $y \in [0, 1]$ where each point in the square is equally likely to be sampled (think about a suitable probability distribution to use).

- Plot the points using the `plot` function.
- Write a function that returns the co-ordinate in the square that is nearest (measured using Euclidean distance) to a user specified co-ordinate (hint: you may find the function `which.min` helpful).
- Try your function out on your 50 data points
- Add, in green, the user-specified co-ordinate to your plot using the `points` function.
- Add, in red, the co-ordinate nearest to the user-specified co-ordinate.

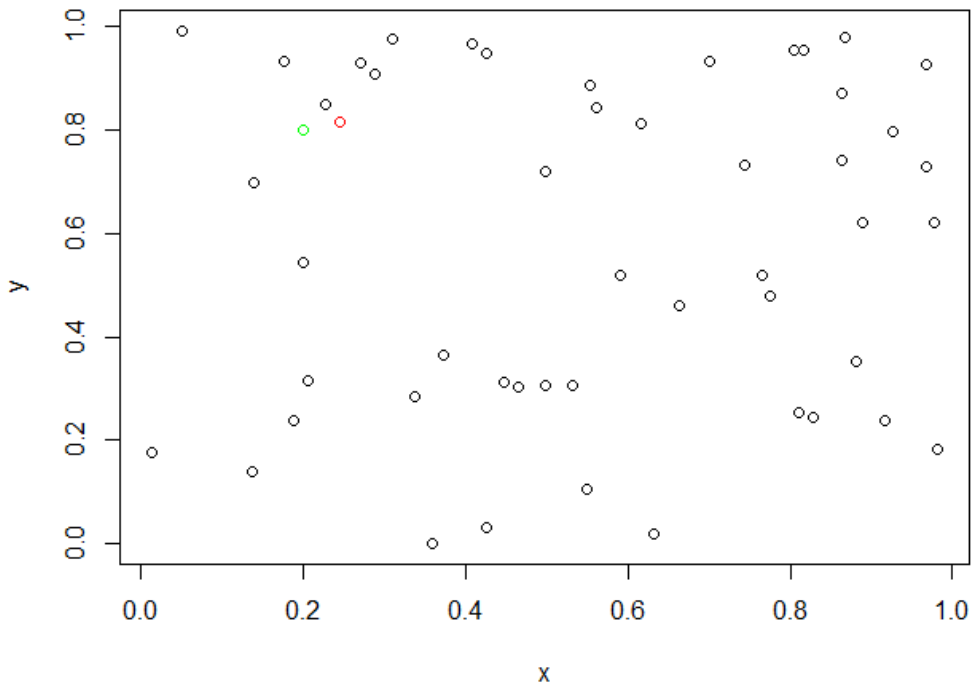

Figure 4.1: A plot showing some uniformly distributed points over a unit square with the point in red nearest to the point in green

You should end up with a figure similar to Figure 4.1.

5 Default values for function arguments

If we want to give a default value to a function argument, you can easily do this in the function declaration. To make `threshold` take the default value of 0 in our `remove_low_values` function we do so as follows

```
remove_low_values <- function(x, threshold = 0){
  low_values_indices <- which(x < threshold)
  return(x[-low_values_indices])
}
```

We can now just supply a numeric vector to `remove_low_values` without specifying the `threshold` value

```
remove_low_values(x = c(1, -6, 9, 224, 67, 3, 89))
```

```
## [1]   1   9 224  67   3  89
```

If we want a different threshold we just specify its value in the function call

```
remove_low_values(x = c(1, -6, 9, 224, 67, 3, 89), threshold = 10)
```

```
## [1] 224  67  89
```

For existing functions in R we can see which arguments have defaults and what these values are in the help file for the function

```
help(dnorm)
```

We see that `x` has no default value whilst `mean` and `sd` have defaults of 0 and 1 (which is the standard normal distribution).

You don't have to use the argument names in the function call. R will just take the first supplied argument to match the first argument in the function definition, the second supplied argument to match the second argument in the function definition etc. So

```
remove_low_values(x = c(1, -6, 9, 224, 67, 3, 89), threshold = 10)
```

```
## [1] 224  67  89
```

could have been simplified as

```
remove_low_values(c(1, -6, 9, 224, 67, 3, 89), 10)
```

```
## [1] 224  67  89
```

Generally it is recommended to use the argument names in the functions calls for two reasons:

1. you are less likely to make mistakes: for example in `dnorm` using the argument names reminds you to use the standard deviation, rather than the variance since the argument name is `sd` ;
2. if the reader is not very familiar with the function, it makes it clear what the arguments are.

> Try this
>
> Further modify your `calculate_proportion_in_interval_from_normal` function (created in an earlier *Try this* exercise) so that, by default, it simulates 100 realisations and the normal distribution has a mean of 2 and a standard deviation of 1.2; run the code without specifying these three arguments to check it runs.

6 Writing quicker code for basic functions

Consider our code to square the input value

```
square_it <- function(x){
  squared_value <- x ^ 2
  return(squared_value)
}
```

We can simplify this function definition in two ways:

1. When there is only one line of code we can choose **not** to use `{}` to contain the function code (so the code can be written as a single line)
2. you can choose **not** to use the `return` function in your final line of code. R will automatically return the last line provided the final line of code **does not** assign the value to anything (i.e. not contain `<-`).

So we can simplify our function as

```
square_it <- function(x)  x ^ 2
```

Note that we didn't assign `x ^ 2` to an object. If we did this and didn't return anything it wouldn't work. So the code

```
square_it <- function(x) squared_value <- x ^ 2
```

doesn't return anything when the function is called

```
square_it(5)
```

which is not what we want. If you find this confusing just use the `return` function.

> Try this
>
> Modify the `remove_low_values` function code so that it
>
> - **doesn't** have a `return` statement.
> - only has one line of code and no `{}`
>
> Check your new function on some simulated data

7 Automatic code formatting

The `tidy_source` function in the `formatR` package automatically tidies the code in a *.R* script file. Consider the following poorly formatted code

```
remove_low_values_bad_format <- function(x, threshold)
  {
  low_values_indices = which (x<threshold)
  return(x[-low_values_indices] )}
```

If this code is in a script file called *bad_function_format.R* then

```
tidy_source(source = "bad_function_format.R")
```

will tidy the code in the script file and print a better formatted version. You can then copy and paste the improved code into the script file. It isn't perfect; which formatting errors has it not picked up here? So you still need to check the code provided, but is a good initial check.

8 The `extract` function for creating functions

RStudio has a tool that can help to create functions from lines of code in a script file. Suppose we had the following lines of code in a script file

```
x <- c(1, -6, 9, 224, 67, 3, 89)
threshold = 50
low_values_indices = which(x < threshold)
x[-low_values_indices]
```

```
## [1] 224  67  89
```

We want to make the last two lines of code into a function. To do this highlight the last two lines in the script file and then click *Code* → *Extract Function*. Supply a function name (I used `remove_low_values_via_extract`) and you get the following

```
remove_low_values_via_extract <- function(x, threshold) {
  low_values_indices = which(x < threshold)
  x[-low_values_indices]
}
```

Note that `extract` will modify your code so use *Ctrl + z* to undo if it doesn't look correct. For beginners, this is a good way to construct functions. Write the individual lines of code in a script file putting arguments first followed by the lines of code to go into the function, then use the `extract` function to put it into a function. Then check it!

> Try this
>
> Write some individual lines of code (**not** a function) to return the number of elements of a vector that are less than a threshold. Use the `extract` function to make it into a function. Check it.

9 Returning multiple objects from a function

So far our functions have returned a single object (via the `return` function or as the last line of code in the function). How do we allow our function to return **multiple** objects. Since these objects could be of different types (numeric, character, logical etc) it makes sense to return a list (since lists allow collections of objects of different type).

Suppose we wanted to write a function that takes two vectors of strings, and returns the strings that appear in both vectors

```
find_intersection <- function(string1, string2){
  in_common <- intersect(string1, string2)
  return(in_common)
}

first_string <- c("oil", "inequality", "climate", "Brexit", "Trump", "data", "sanction
s", "fear")
second_string <- c("environment", "respect", "freedom", "data", "climate", "inclusion"
)
find_intersection(string1 = first_string, string2 = second_string)
```

```
## [1] "climate" "data"
```

What if instead we wanted to return those strings in the first vector but **not** in the second, and those in the second but **not** in the first

```
find_unique_by_string <- function(string1, string2){
  unique_in_first <- setdiff(string1, string2)
  unique_in_second <- setdiff(string2, string1)
  return(list(unique_in_first, unique_in_second))
}
```

Our function now returns a list of objects. We can access them using the `[[ ]]` list subsetting operator

```
unique_by_string <- find_unique_by_string(string1 = first_string, string2 = second_str
ing)
(unique_words_string1 <- unique_by_string[[1]])
```

```
## [1] "oil"        "inequality" "Brexit"     "Trump"      "sanctions"
## [6] "fear"
```

```
(unique_words_string2 <- unique_by_string[[2]])
```

```
## [1] "environment" "respect"     "freedom"     "inclusion"
```

10 Using the 'pipe' with the magrittr function

The pipe, `%>%` , comes from the `magrittr` package. If the `magrittr` package isn't installed on your computer then install it now. Also make sure that `magrittr` is loaded into your search path. Check using `search()` .

What is the pipe? It is a way of making your code easier to read and understand. It is often used when you apply a series of functions to some object. For example, suppose we have a set of numbers and we want to find the mean of those values that are at least 5. We could use our `remove_low_values` function from Section 2.4. to remove values less than 5 as follows (here we will print all steps to show the output).

```
(x <- seq(2, 10, by = 0.5))
```

```
##  [1]  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  8.5
## [15]  9.0  9.5 10.0
```

```
(nums_at_least_five <- remove_low_values(x, threshold = 5))
```

```
##  [1]  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

```
mean(nums_at_least_five)
```

```
## [1] 7.5
```

This code is fine but we have to create an object called `nums_at_least_five` that we aren't really interested in but have to give a name to so that we can use it as input to the `mean` function. We could avoid this by combining the last two lines

```
mean(remove_low_values(x, threshold = 5))
```

```
## [1] 7.5
```

but if you start to apply too many functions, code like the above becomes hard to follow. This is where the pipe may help. You use the pipe as follows

```
library(magrittr)
x %>% remove_low_values(threshold = 5) %>% mean()
```

```
## [1] 7.5
```

You can think of it as putting the `x` values into a pipe (`%>%`) which feeds it into the `remove_low_values` function (with a threshold of 5). The output of the first pipe is then put into the second pipe (`%>%`) which feeds it through the `mean` function.

Notice that in the first pipe we don't need to specify `x` as an argument to the `remove_low_values` function. It is assumed to be the first argument. Using the pipe makes it clear what the series of funtions being applied is, and we don't need to assign the intermediate output to an object (so we don't clutter our Global Environment).

> Try this
>
> Change the first element of the x vector to an NA. Run the pipe above on x. What happens? Modify the pipe code so that it will ignore the NA value and return a number (hint: look at the help for the mean function). You should still use pipes to do this.

11 Exercises B

1. Modify the `highest. value_to.a_power` function (Q1 of Exercises A) so that it returns both the minimum, median and maximum of the numbers raised to some user-specified power

Run the function on 100 realisations of a random variable with any appropriate distribution. Extract the minimum, median and maximum values and assign them each to a suitably named object.

2. Further modify the function in Question 4 of Exercises A so that it returns the nth smallest value in a vector where n is user supplied but is 3 by default. Check your function when you don't supply a value of n in the function call.
3. A quadratic form based on an $n \times n$ real symmetric matrix A is given by $q(\boldsymbol{x}) = \boldsymbol{x}^T A \boldsymbol{x}$ where $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$ is a vector in $\mathbb{R}^n$. Write a function (called `quadratic_form`) to calculate the quadratic form for

- a user-supplied vector $\boldsymbol{x} \in \mathbb{R}^n$
- a user-supplied $n \times n$ real symmetric matrix A with a default of the 2×2 identity matrix, the `diag` function might help here

Hint: the matrix transpose function ($\boldsymbol{x}^T$) in R is `t` . Check your function works

4. (A hard question) An $n \times n$ real symmetric matrix A is said to be positive definite if the quadratic form $\boldsymbol{x}^T A \boldsymbol{x} > 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$.

Write some code to

- simulate lots of 2-dimensional co-ordinates
- calculate the quadratic form for each co-ordinate

and hence assess whether the matrix $A = \begin{pmatrix} 4 & 5 \\ 3 & 4 \end{pmatrix}$ could be positive definite.

Hint: You may find the `replicate` function useful here (see `?replicate`). It may also help to write a function that calls `quadratic_form` within it

5. Rewrite your code in Q4 so that it makes use of the pipe operator `%>%` .

References

Wickham, H. 2015. *Advanced R*. CRC Press.