3 - Functions
Introduction
Defining a Function
# boiling point of water
Composing Functions
Nesting Function Calls
Function Exercises
Exercise 1
Exercise 2

3 - Functions

Introduction

Functions allow you to automate common tasks in a more powerful and general way than repeating the same code manually. Writing a function offers several advantages:

1. Can reduce repetitive operations to a single command.
2. Make useful code from one program reusable across many other programs.
3. As requirements change, you only need to update code in one place (inside the function), instead of many.
    a. You eliminate the chance of making incidental mistakes when you copy and paste (e.g., updating a variable name in one place, but forgetting to in another).
4. Makes code more readable and understandable.

Defining a Function

We start by defining a function name (say "my_function") and the input parameter(s) that the user will feed to the function. Then we define the operation(s) that take place in the "body" of the function within curly braces ({}). Finally, we assign the result (or output) of the function in the "return" statement.

```
# notice that my_function below is a variable to the left of <-,
# and is being assigned the output of the function called 'function'
#
my_function <- function(input_param1, input_param2, …) {
      # perform operations involving input_param1, input_param2, …
      # …
      return(result_of_above_operations)
}
```

Now let’s see this process with an example. We are going to define a function fahrenheit_to_celsius that converts a temperature from Fahrenheit to Celsius:

```
# fahrenheit_to_celsius: convert fahrenheit to celsius temperature
# input1 temp_F
# return temp_C
#
fahrenheit_to_celsius <- function(temp_F) {
      temp_C <- (temp_F - 32) * 5 / 9
      return(temp_C)

      # note: above two lines can be shortened to just
      #            return((temp_F - 32) * 5 / 9)
}
```

We define the fahrenheit_to_celsius function by assigning it the output of function. The list of input parameter names (a.k.a., arguments) names are contained within parentheses. Next, the body of the function – the statements that are executed when it runs – is contained within curly braces ({}). The statements in the body are indented by some number of spaces, which makes the code easier to read but does not affect how the code operates.

When we call the function, the values we pass to it are assigned to the input parameter variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

Let’s try running our function. Calling our own function is no different from calling any other function:

```
# freezing point of water
fahrenheit_to_celsius(32)

# output
[1] 0
```

# boiling point of water

```
fahrenheit_to_celsius(212)

# output
[1] 100

Q: Why would we bother to write and use this function, if we could just write (32 - 32) * 5 / 9
   and (212 - 32) * 5 / 9 in our R programs?

A: ?
```

Now that we’ve seen how to turn Fahrenheit into Celsius, it should be easy to turn Celsius into Kelvin temperature:

```
# celsius_to_kelvin: convert celsius to kelvin temperature
# input1 temp_C
# return temp_K
#
```

```
celsius_to_kelvin <- function(temp_C) {
      temp_K <- temp_C + 273.15
      return(temp_K)
}

# freezing point of water in Kelvin
celsius_to_kelvin(0)

# output
[1] 273.15

# Q: Why bother to write this simple function if we can just add 273.15 to a celsius temperature?

# A: ?
```

Composing Functions

What about converting Fahrenheit to Kelvin? We could write out the formula, but we don’t need to. Instead, we can "compose" (combine) the two functions we have already created in this new function:

```
# fahrenheit_to_kelvin: convert fahrenheit to kelvin temperature
# input1 temp_F
# return temp_K
#
fahrenheit_to_kelvin <- function(temp_F) {
      #
      # function composition happening here
      #
      temp_C <- fahrenheit_to_celsius(temp_F)
      temp_K <- celsius_to_kelvin(temp_C)

      return(temp_K)
}

# freezing point of water in Kelvin
fahrenheit_to_kelvin(32.0)

# output
[1] 273.15
```

This is a first taste of how larger programs are built: we define basic operations, then combine them into larger reusable chunks (functions) to get the effect we want. Real-life functions will usually be larger than the ones shown here—typically half a dozen to a few dozen lines—but they should not be much longer than that. A very long function becomes incomprehensible. As a rule of thumb, functions should be created to do just one thing, and do it reliably.

Nesting Function Calls

The above fahrenheit_to_kelvin function example showed the output of fahrenheit_to_celsius assigned to temp_C, which is then passed to celsius_to_kelvin to get the final result. It is also possible to perform this calculation in one line of code, by "nesting" one function call inside another:

```
# freezing point of water in Fahrenheit
celsius_to_kelvin(fahrenheit_to_celsius(32.0))

# output
[1] 273.15
```

Here, we call the fahrenheit_to_celsius function to convert 32.0 from Fahrenheit to Celsius, and immediately pass the value returned from fahrenheit_to_celsius to celsius_to_kelvin to convert from Celsius to Kelvin.

We can also rewrite the fahrenheit_to_kelvin function using function nesting:

```
fahrenheit_to_kelvin <- function(temp_F) {
       #
       # function composition happening here
       #
       return(celsius_to_kelvin(fahrenheit_to_celsius(temp_F)))
}
```

This is convenient, but you should be careful not to nest too many function calls at once - it can become confusing and difficult to read.

Function Exercises

Exercise 1

We combine elements into a vector using the c (combine) function. For example,

```
x <- c("A", "B", "C")
```

creates a vector x with three elements. We can extend that vector again using c, e.g.,

```
y <- c(x, "D")
```

to create a vector y with four elements ("A", "B", "C", "D").

Write a function called 'highlight' that takes two vectors as input arguments, *content* and *wrapper*, and returns a new vector that has the wrapper vector at the beginning and end of the content. The 'highlight' function would be used as follows:

```
# slogan will be passed to highlight as the input param content
slogan <- c("IT'S", "FOR", "YOUR", "HEALTH")  # vector of strings
```

```
# clown will be passed to highlight as the input param wrapper
clown <- "*:o)"  # single string vector

highlight(slogan, clown)

# desired output
*:o)   IT'S FOR YOUR HEALTH   *:o)
```

A partially correct function definition is shown below:

```
# function highlight: takes some text content and puts markers around it for highlighting
# input1: content
# input2: wrapper
# return: wrapped content
#
# To get correct output, you'll have to use the paste function before you return(answer).
# See ?paste in the R console or find external help. Alternatively, replace the 'return'
# statement with a call to the 'cat' function. See ?cat in the R console.
#
highlight <- function(content, wrapper) {
      answer <- c(wrapper, content, wrapper)
      return(answer)
}
```

Note: In R, functions can accept arguments explicitly assigned to a variable name in the function call, i.e., functionName(variable = value), as well as arguments by order. So, the following ways to call the above highlight function are also valid:

```
highlight(content = slogan, wrapper = clown)
highlight(content = slogan, clown)
highlight(slogan, wrapper = clown)
highlight(wrapper = clown, content = slogan)  # order of inputs is reversed, but output unchanged
highlight(wr = clown, co = slogan)            # partial input parameter names also work
highlight(p1 = clown, p2 = slogan)            # this won't work
highlight('FYIFV', '..!..')                   # can use literal string arguments
highlight(content = 'FYIFV', wrap = '..!..')
```

Exercise 2

To get the index of an element in a vector, we can use the which function. For example:

```
v <- c(TRUE, FALSE, TRUE, FALSE)
which(v)
[1] 1 3      # returns the indices of vector v where the TRUE values are

which(state.abb == 'NJ')
[1] 30       # returns the index of vector state.abb where the value is 'NJ'
```

Write a function called 'state_area' that inputs a two letter state abbreviation and returns the area of the state in square miles and square kilometers. The 'state_area' function would be used as follows:

```
print state_area('NJ')
[1] 7836 20295            # first number = square miles, second number = square kilometers

print state_area('wy')    # note: input state abbreviation can be mixed case
[1] 97914 253598
```

The start of the function definition is shown below:

```
# function state_area: returns state area in square miles and kilometers
# input1: (string) two letter state abbreviation; mixed case allowed
# return: vector containing state area in square miles, square kilometers
#
state_area <- function(state_abb) {

}
```

- You'll need to use the state.abb and state.area built-in data sets in R.
- Conversion formula: $km^2 = mi^2 / 0.38610$
- Other functions that you'll find useful to build the state_area function are the following:
    - toupper()   (see ?toupper in R)
    - round()   (see ?round in R)