Introduction to functions in R - Efficient scientific programming

Learning Objectives

After completing this tutorial, you will be able to:

- Create a new function in R
- Understand the concept of a function argument.

What you need

You will need a computer with internet access to complete this lesson.

In this lesson, we'll learn how to write a function so that we can repeat several operations with a single command.

Defining a Function

Let's start by defining a function `fahr_to_kelvin` that converts temperatures from Fahrenheit to Kelvin:

```
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

We define `fahr_to_kelvin` by assigning it to the output of `function`. The list of argument names are contained within parentheses - in this case (temp) is the only argument that this function requires.

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 two 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 those 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.

Automatic Returns

In R, it is not necessary to include the return statement. R automatically returns whichever variable is on the last line of the body of the function. Since we are just learning, we will explicitly define the return statement. {: .notice}

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

```
# freezing point of water
fahr_to_kelvin(32)
## [1] 273.15
# boiling point of water
fahr_to_kelvin(212)
## [1] 373.15
```

We've successfully called the function that we defined, and we have access to the value that we returned.

Composing Functions

Now that we've seen how to turn Fahrenheit into Kelvin, it's easy to turn Kelvin into Celsius:

```
kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}

# absolute zero in Celsius
kelvin_to_celsius(0)
## [1] -273.15
```

What about converting Fahrenheit to Celsius? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

```
fahr_to_celsius <- function(temp) {
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}

# freezing point of water in Celsius
fahr_to_celsius(32.0)
## [1] 0
```

This is our first taste of how larger programs are built: we define basic operations, then combine them in ever-larger chunks 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 shouldn't ever be much longer than that, or the next person who reads it won't be able to understand what's going on.

Chaining Functions

This example showed the output of `fahr_to_kelvin` assigned to `temp_k`, which is then passed to `kelvin_to_celsius` to get the final result. It is also possible to perform this calculation in one line of code, by "chaining" functions together, like so:

```
# freezing point of water in Celsius
kelvin_to_celsius(fahr_to_kelvin(32.0))
## [1] 0
```

Create a Function

We can use the `c` function to **c**oncatenate elements. We have done this when we classified rasters. For example

`x <- c("A", "B", "C")` creates a vector `x` with three elements. Furthermore, we can extend that vector again using `c`.

For example,

`y <- c(x, "D")` creates a vector `y` with four elements.

Optional challenge

Write a function called `fence` that takes two vectors as arguments, called original`and`wrapper‘, and returns a new vector that has the wrapper vector at the beginning and end of the original. An example of how the function should run is below.

```
 best_practice <- c("Write", "programs", "for", "people", "not", "computers")
 asterisk <- "***"  # R interprets a variable with a single value as a vector
                    # with one element.
 fence(best_practice, asterisk)
## [1] "***"       "Write"     "programs"  "for"       "people"    "not"
## [7] "computers" "***"
```

Optional challenge 2

If the variable `v` refers to a vector, then `v[1]` is the vector’s first element and `v[length(v)]` is its last (the function `length` returns the number of elements in a vector). Write a function called `outside` that returns a vector made up of just the first and last elements of its input:

```
 dry_principle <- c("Don't", "repeat", "yourself", "or", "others")
 outside(dry_principle)
## [1] "Don't"  "others"
```

The function environment is self-contained

THere are many benefits to using functions in your code including it will make your code simpler and easier to read. However another significant benefit is that it will save memory and keep your coding environment cleaner. Let’s discuss that next.

The function environment is self-contained. What this means is that when you run a function, it does not create intermediate variables in your global environment.

For example, above we created a function called `fahr_to_censius`. Within that function, we see it creates two variables:

1. `temp_k`
2. `result`

Run the function below. Then call it `fahr_to_celsius(15)`. Look closely at your global environment in `R`. Do you see the variables temp_k or result?

```
fahr_to_celsius <- function(temp) {
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}

fahr_to_celsius(15)
## [1] -9.444444
```

When we run the function above, it creates a distinct environment where it runs the steps required to complete the tasks specified in the function. However, the variables defined by the intermediate steps are not retained in your global environment. Each variable that you create consumes memory on your computer, so this saves memory. It also reduces the “clutter” associated with too many variables.

Documentation

We have one more task first, though: we should write some documentation for our function to remind ourselves later what it's for and how to use it.

A common way to put documentation in software is to add comments like this:

```r
# define function
fahr_to_celsius <- function(temp) {
  # function that converts temperature in farenheit to celcius
  # input: temperature in degrees F
  # output: temperature in censius
  temp_k <- fahr_to_kelvin(temp)
  result <- kelvin_to_celsius(temp_k)
  return(result)
}
```

Writing Documentation

Formal documentation for R functions is written in separate `.Rd` using a markup language similar to LaTeX. You see the result of this documentation when you look at the help file for a given function, e.g. `?read.csv`. The roxygen2 package allows R coders to write documentation alongside the function code and then process it into the appropriate `.Rd` files. You will want to switch to this more formal method of writing documentation when you start writing more complicated R projects.

{: .notice }