Functions Explained

If we only had one data set to analyze, it would probably be faster to load the file into a spreadsheet and use that to plot simple statistics. However, the our data may be updated periodically, and we may want to pull in that new information later and re-run our analysis again. We may also obtain similar data from a different source in the future.

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

## What is a function?

Functions gather a sequence of operations into a whole, preserving it for ongoing use. Functions provide:

* a name we can remember and invoke it by
* relief from the need to remember the individual operations
* a defined set of inputs and expected outputs
* rich connections to the larger programming environment

As the basic building block of most programming languages, user-defined functions constitute “programming” as much as any single abstraction can. If you have written a function, you are a computer programmer.

## Defining a function

Let’s open a new R script file in the functions/ directory and call it functions-lesson.R.

Let’s define 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. Next, the [body](%7B%7B%20page.root%20%7D%7D/reference/#function-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. This makes the code easier to read but does not affect how the code operates.

It is useful to think of creating functions like writing a cookbook. First you define the “ingredients” that your function needs. In this case, we only need one ingredient to use our function: “temp”. After we list our ingredients, we then say what we will do with them, in this case, we are taking our ingredient and applying a set of mathmatical operators to it.

When we call the function, the values we pass to it as arguments are assigned to those variables so that we can use them inside the function. Inside the function, we use a [return statement](%7B%7B%20page.root%20%7D%7D/reference/#return-statement) to send a result back to whoever asked for it.

## Tip

One feature unique to R is that the return statement is not required. R automatically returns whichever variable is on the last line of the body of the function. But for clarity, we will explicitly define the return statement.

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

## Challenge 1

Write a function called kelvin\_to\_celsius() that takes a temperature in Kelvin and returns that temperature in Celsius.

Hint: To convert from Kelvin to Celsius you subtract 273.15

## Solution to challenge 1

Write a function called kelvin\_to\_celsius that takes a temperature in Kelvin and returns that temperature in Celsius

kelvin\_to\_celsius <- function(temp) {  
 celsius <- temp - 273.15  
 return(celsius)  
}

## Combining functions

The real power of functions comes from mixing, matching and combining them into ever-larger chunks to get the effect we want.

Let’s define two functions that will convert temperature from Fahrenheit to Kelvin, and Kelvin to Celsius:

fahr\_to\_kelvin <- function(temp) {  
 kelvin <- ((temp - 32) \* (5 / 9)) + 273.15  
 return(kelvin)  
}  
  
kelvin\_to\_celsius <- function(temp) {  
 celsius <- temp - 273.15  
 return(celsius)  
}

## Challenge 2

Define the function to convert directly from Fahrenheit to Celsius, by reusing the two functions above (or using your own functions if you prefer).

## Solution to challenge 2

Define the function to convert directly from Fahrenheit to Celsius, by reusing these two functions above

fahr\_to\_celsius <- function(temp) {  
 temp\_k <- fahr\_to\_kelvin(temp)  
 result <- kelvin\_to\_celsius(temp\_k)  
 return(result)  
}

## Defensive Programming

Now that we’ve begun to appreciate how writing functions provides an efficient way to make R code re-usable and modular, we should note that it is important to ensure that functions only work in their intended use-cases. Checking function parameters is related to the concept of *defensive programming*. Defensive programming encourages us to frequently check conditions and throw an error if something is wrong. These checks are referred to as assertion statements because we want to assert some condition is TRUE before proceeding. They make it easier to debug because they give us a better idea of where the errors originate.

### Checking conditions with stopifnot()

Let’s start by re-examining fahr\_to\_kelvin(), our function for converting temperatures from Fahrenheit to Kelvin. It was defined like so:

fahr\_to\_kelvin <- function(temp) {  
 kelvin <- ((temp - 32) \* (5 / 9)) + 273.15  
 return(kelvin)  
}

For this function to work as intended, the argument temp must be a numeric value; otherwise, the mathematical procedure for converting between the two temperature scales will not work. Luckily R provides the convenience function stopifnot(). We can list as many requirements that should evaluate to TRUE; stopifnot() throws an error if it finds one that is FALSE. Listing these conditions also serves a secondary purpose as extra documentation for the function.

Let’s try out defensive programming with stopifnot() by adding assertions to check the input to our function fahr\_to\_kelvin().

We want to assert the following: temp is a numeric vector. We may do that like so:

fahr\_to\_kelvin <- function(temp) {  
 stopifnot(is.numeric(temp))  
 kelvin <- ((temp - 32) \* (5 / 9)) + 273.15  
 return(kelvin)  
}

It still works when given proper input.

# freezing point of water  
fahr\_to\_kelvin(temp = 32)

## [1] 273.15

But fails instantly if given improper input.

# Metric is a factor instead of numeric  
fahr\_to\_kelvin(temp = as.factor(32))

## Error in fahr\_to\_kelvin(temp = as.factor(32)): is.numeric(temp) is not TRUE

## Challenge 3

Use defensive programming to ensure that our fahr\_to\_celsius() function throws an error immediately if the argument temp is specified inappropriately.

## Solution to challenge 3

Extend our previous definition of the function by adding in an explicit call to stopifnot(). Since fahr\_to\_celsius() is a composition of two other functions, checking inside here makes adding checks to the two component functions redundant.

fahr\_to\_celsius <- function(temp) {  
 stopifnot(is.numeric(temp))  
 temp\_k <- fahr\_to\_kelvin(temp)  
 result <- kelvin\_to\_celsius(temp\_k)  
 return(result)  
}

### Testing and Documenting

Once we start putting things in functions so that we can re-use them, we need to start testing that those functions are working correctly. To see how to do this, let’s write a function to center a dataset around a particular midpoint:

center <- function(data, midpoint) {  
 new\_data <- (data - mean(data)) + midpoint  
}

We could test this on our actual data, but since we don’t know what the values ought to be, it will be hard to tell if the result was correct. Instead, let’s create a vector of 0s and then center that around 3. This will make it simple to see if our function is working as expected:

z <- c(0, 0, 0, 0)  
z

## [1] 0 0 0 0

center(z, 3)

That looks right, so let’s try center on our real data. We’ll center the mtcars data on mpg around 0:

dat <- mtcars  
centered <- center(dat[, "mpg"], 0)  
head(centered)

## [1] 0.909375 0.909375 2.709375 1.309375 -1.390625 -1.990625

It’s hard to tell from the default output whether the result is correct, but there are a few simple tests that will reassure us:

# original min  
min(dat[, 4])

## [1] 52

# original mean  
mean(dat[, 4])

## [1] 146.6875

# original max  
max(dat[, 4])

## [1] 335

# centered min  
min(centered)

## [1] -9.690625

# centered mean  
mean(centered)

## [1] 4.440892e-16

# centered max  
max(centered)

## [1] 13.80937

That seems almost right: the original mean was about 146.69, so the lower bound from zero is now about -146.69. The mean of the centered data is 4.440892110^{-16}. We can even go further and check that the standard deviation hasn’t changed:

# original standard deviation  
sd(dat[, 4])

## [1] 68.56287

# centered standard deviation  
sd(centered)

## [1] 6.026948

Those values look the same, but we probably wouldn’t notice if they were different in the sixth decimal place. Let’s do this instead:

# difference in standard deviations before and after  
sd(dat[, 4]) - sd(centered)

## [1] 62.53592

Sometimes, a very small difference can be detected due to rounding at very low decimal places. R has a useful function for comparing two objects allowing for rounding errors, all.equal:

all.equal(sd(dat[, 4]), sd(centered))

## [1] "Mean relative difference: 0.912096"

It’s still possible that our function is wrong, but it seems unlikely enough that we should probably get back to doing our analysis. We have one more task first, though: we should write some [documentation](%7B%7B%20page.root%20%7D%7D/reference.html#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](%7B%7B%20page.root%20%7D%7D/reference.html#comment) like this:

center <- function(data, midpoint) {  
 # return a new vector containing the original data centered around the  
 # midpoint.  
 # Example: center(c(1, 2, 3), 0) => c(-1, 0, 1)  
 new\_data <- (data - mean(data)) + midpoint  
 return(new\_data)  
}

## Tip: Testing and documenting

It’s important to both test functions and document them: Documentation helps you, and others, understand what the purpose of your function is, and how to use it, and its important to make sure that your function actually does what you think.

When you first start out, your workflow will probably look a lot like this:

1. Write a function
2. Comment parts of the function to document its behaviour
3. Load in the source file
4. Experiment with it in the console to make sure it behaves as you expect
5. Make any necessary bug fixes
6. Rinse and repeat.

Formal documentation for functions, written in separate .Rd files, gets turned into the documentation you see in help files. The [roxygen2](http://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html) 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.

Formal automated tests can be written using the [testthat](http://r-pkgs.had.co.nz/tests.html) package.

## Loading functions from a separate file for cleaner code

If you’ve been writing these functions down into a separate R script (a good idea!), you can load in the functions into our R session by using the source() function:

source("functions/functions-lesson.R")

## Tip: Pass by value

Functions in R almost always make copies of the data to operate on inside of a function body. When we modify dat inside the function we are modifying the copy of the mtcars dataset stored in dat, not the original variable we gave as the first argument.

This is called “pass-by-value” and it makes writing code much safer: you can always be sure that whatever changes you make within the body of the function, stay inside the body of the function.

## Tip: Function scope

Another important concept is scoping: any variables (or functions!) you create or modify inside the body of a function only exist for the lifetime of the function’s execution. When we call center(), the variable dat only exists inside the body of the function. Even if we have variables of the same name in our interactive R session, they are not modified in any way when executing a function.

## Defining Defaults

We have passed arguments to functions in two ways: directly, as in dim(dat), and by name, as in read.csv(file = "data/inflammation-01.csv", header = FALSE). In fact, we can pass the arguments to functions without naming them:

dat <- round(3.14, 2)

However, the position of the arguments matters if they are not named.

dat <- round(2, 3.14)

To understand what’s going on, and make our own functions easier to use, let’s re-define our center function like this:

center <- function(data, midpoint = 0) {  
 # return a new vector containing the original data centered around the  
 # midpoint (0 by default).  
 # Example: center(c(1, 2, 3), 0) => c(-1, 0, 1)  
 new\_data <- (data - mean(data)) + midpoint  
 return(new\_data)  
}

The key change is that the second argument is now written midpoint = 0 instead of just midpoint. If we call the function with two arguments, it works as it did before:

test\_data <- c(0, 0, 0, 0)  
center(test\_data, 3)

## [1] 3 3 3 3

But we can also now call center() with just one argument, in which case midpoint is automatically assigned the default value of 0:

more\_data <- 5 + test\_data  
more\_data

## [1] 5 5 5 5

center(more\_data)

## [1] 0 0 0 0

This is handy: if we usually want a function to work one way, but occasionally need it to do something else, we can allow people to pass an argument when they need to but provide a default to make the normal case easier.

The example below shows how R matches values to arguments

display <- function(a = 1, b = 2, c = 3) {  
 result <- c(a, b, c)  
 names(result) <- c("a", "b", "c") # This names each element of the vector  
 return(result)  
}  
  
# no arguments  
display()

## a b c   
## 1 2 3

# one argument  
display(55)

## a b c   
## 55 2 3

# two arguments  
display(55, 66)

## a b c   
## 55 66 3

# three arguments  
display(55, 66, 77)

## a b c   
## 55 66 77

As this example shows, arguments are matched from left to right, and any that haven’t been given a value explicitly get their default value. We can override this behavior by naming the value as we pass it in:

# only setting the value of c  
display(c = 77)

## a b c   
## 1 2 77

## Matching Arguments

To be precise, R has three ways that arguments supplied by you are matched to the *formal arguments* of the function definition:

1. by complete name,
2. by partial name (matching on initial *n* characters of the argument name), and
3. by position.

Arguments are matched in the manner outlined above in *that order*: by complete name, then by partial matching of names, and finally by position.

## Tip

R has some unique aspects that can be exploited when performing more complicated operations. We will not be writing anything that requires knowledge of these more advanced concepts. In the future when you are comfortable writing functions in R, you can learn more by reading the [R Language Manual](http://cran.r-project.org/doc/manuals/r-release/R-lang.html#Environment-objects) or this [chapter](http://adv-r.had.co.nz/Environments.html) from [Advanced R Programming](http://adv-r.had.co.nz/) by Hadley Wickham.