Learning Objectives

After completing this tutorial, you will be able to:

- Find and download data from the USGS Earth Explorer Website
- Filter data by cloud cover to find datasets with the least amount of clouds for a study area.

What you need

You will need a computer with internet access to complete this lesson and the data for week 6 / 7 of the course.

{% include/data_subsets/course_earth_analytics/_data-week6-7.md %}

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

In the last lesson, we learned to **c**oncatenate elements into a vector using the `c` function, e.g. `x <- c("A", "B", "C")` creates a vector `x` with three elements. Furthermore, we can extend that vector again using `c`, e.g. `y <- c(x, "D")` creates a vector `y` with four elements.

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:

```
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" "***"
```

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"
```

{: .challenge} "‘

The function environment is self-contained

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.

Named Variables and the Scope of Variables

- Functions can accept arguments explicitly assigned to a variable name in in the function call `functionName(variable = value)`, as well as arguments by order:

```
input_1 = 20
mySum <- function(input_1, input_2 = 10) {
  output <- input_1 + input_2
  return(output)
}
```

1. Given the above code was run, which value does `mySum(input_1 = 1, 3)` produce?
    1. 4
    2. 11
    3. 23

4. 30

2. If `mySum(3)` returns 13, why does `mySum(input_2 = 3)` return an error? {: .challenge}

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:

```
# 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 }

Functions to Create Graphs

Write a function called `analyze` that takes a filename as a argument and displays the three graphs produced in the previous lesson (average, min and max inflammation over time). `analyze("data/inflammation-01.csv")` should produce the graphs already shown, while `analyze("data/inflammation-02.csv")` should produce corresponding graphs for the second data set. Be sure to document your function with comments.

Solution

```
analyze <- function(filename) {
  # Plots the average, min, and max inflammation over time.
  # Input is character string of a csv file.
  dat <- read.csv(file = filename, header = FALSE)
  avg_day_inflammation <- apply(dat, 2, mean)
  plot(avg_day_inflammation)
  max_day_inflammation <- apply(dat, 2, max)
  plot(max_day_inflammation)
  min_day_inflammation <- apply(dat, 2, min)
  plot(min_day_inflammation)
}
```

{: .r} {: .solution} {: .challenge}

Rescaling

Write a function `rescale` that takes a vector as input and returns a corresponding vector of values scaled to lie in the range 0 to 1. (If L and H are the lowest and highest values in the original vector, then the replacement for a value v should be $(v-L)/(H-L)$.) Be sure to document your function with comments.

Test that your `rescale` function is working properly using `min`, `max`, and `plot`.

Solution

```
rescale <- function(v) {
  # Rescales a vector, v, to lie in the range 0 to 1.
  L <- min(v)
  H <- max(v)
  result <- (v - L) / (H - L)
  return(result)
}
```

{: .r} {: .solution} {: .challenge}

Default function arguments

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 `read.csv` without naming them:

```
dat <- read.csv("data/inflammation-01.csv", FALSE)
```

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

```
dat <- read.csv(header = FALSE, file = "data/inflammation-01.csv")
dat <- read.csv(FALSE, "data/inflammation-01.csv")
```

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

The key change is that the second argument is now written `desired = 0` instead of just `desired`. 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)
```

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

```
more_data <- 5 + test_data
more_data
center(more_data)
```

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()
# one argument
display(55)
# two arguments
display(55, 66)
# three arguments
display (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)
```

Matching Arguments

To be precise, R has three ways that arguments are 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. {: .callout}

With that in hand, let's look at the help for `read.csv()`:

```
?read.csv
```

There's a lot of information there, but the most important part is the first couple of lines:

```
read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

This tells us that `read.csv()` has one argument, `file`, that doesn't have a default value, and six others that do. Now we understand why the following gives an error:

```
dat <- read.csv(FALSE, "data/inflammation-01.csv")
```

It fails because `FALSE` is assigned to `file` and the filename is assigned to the argument `header`.

A Function with Default Argument Values

Rewrite the `rescale` function so that it scales a vector to lie between 0 and 1 by default, but will allow the caller to specify lower and upper bounds if they want. Compare your implementation to your neighbor's: do the two functions always behave the same way?

Solution

```
rescale <- function(v, lower = 0, upper = 1) {
  # Rescales a vector, v, to lie in the range lower to upper.
  L <- min(v)
  H <- max(v)
  result <- (v - L) / (H - L) * (upper - lower) + lower
  return(result)
}
```

{: .r} {: .solution} {: .challenge}