‹ (../01-starting-with-data/index.html)

Programming with R (../)

› (../03-loops-R/ind

Creating Functions

Overview

Teaching: 30 min
Exercises: 0 min
Questions

- How do I make a function?
- How can I test my functions?
- How should I document my code?

Objectives

- Define a function that takes arguments.
- Return a value from a function.
- Test a function.
- Set default values for function arguments.
- Explain why we should divide programs into small, single-purpose functions.

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 some simple statistics. But we have twelve files to check, and may have more 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.

Defining a Function

You can write your own functions in order to make repetitive operations using a single command. Let's start by defining your function "my_function" and the input parameter(s) that the user will feed to the function. Afterwards you will define the operation that you desire to program in the body of the function within curly braces ({}). Finally, you need to assign the result (or output) of your function in the return statement.

Now let's see this process with an example. We are going to define a function `fahrenheit_to_celsius` that converts temperatures from Fahrenheit to Celsius (https://en.wikipedia.org/wiki/Temperature_conversion_formulas#Fahrenheit):

R

```
fahrenheit_to_celsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5 / 9
  return(temp_C)
}
```

We define `fahrenheit_to_celsius` by assigning it to the output of `function` . The list of argument names are contained within parentheses. Next, the body (../reference.html#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, 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 (../reference.html#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. While in the learning phase, 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:

R

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

Output

```
[1] 0
```

R

```
# boiling point of water
fahrenheit_to_celsius(212)
```

Output

```
[1] 100
```

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 Celsius, it's easy to turn Celsius into Kelvin:

R

```
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
```

What about converting Fahrenheit to Kelvin? We could write out the formula, but we don't need to. Instead, we can compose (../reference.html#function-composition) the two functions we have already created:

R

```
fahrenheit_to_kelvin <- function(temp_F) {
  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 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.

📌 Nesting Function Calls

This 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, like so:

R

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

Output

```
[1] 273.15
```

Here, we call `fahrenheit_to_celsius` 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. Our conversion from Fahrenheit to Kelvin is done, all in one go!

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!

Create a Function

In the last lesson, we learned to **c**ombine 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 `highlight` that takes two vectors as arguments, called `content` and `wrapper`, and returns a new vector that has the wrapper vector at the beginning and end of the content:

R

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

Output

```
[1] "***"       "Write"     "programs"  "for"       "people"    "not"
[7] "computers" "***"
```

Solution

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 `edges` that returns a vector made up of just the first and last elements of its input:

R

```
dry_principle <- c("Don't", "repeat", "yourself", "or", "others")
edges(dry_principle)
```

Output

```
[1] "Don't"  "others"
```

Solution

The Call Stack

For a deeper understanding of how functions work, you'll need to learn how they create their own environments and call other functions. Function calls are managed via the call stack. For more details on the call stack, have a look at the supplementary material (../14-supp-call-stack/).

Named Variables and the Scope of Variables

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

R

```
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?

Solution

Testing, Error Handling, 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:

R

```
center <- function(data, midpoint) {
  new_data <- (data - mean(data)) + midpoint
  return(new_data)
}
```

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:

R

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

Output

```
[1] 0 0 0 0
```

R

```
center(z, 3)
```

Output

```
[1] 3 3 3 3
```

That looks right, so let's try center on our real data. We'll center the inflammation data from day 4 around 0:

R

```
dat <- read.csv(file = "data/inflammation-01.csv", header = FALSE)
centered <- center(dat[, 4], 0)
head(centered)
```

Output

```
[1]  1.25 -0.75  1.25 -1.75  1.25  0.25
```

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:

R

```
# original mean
mean(dat[, 4])
```

Output

```
[1] 1.75
```

R

```
# centered mean
mean(centered)
```

Output

```
[1] 0
```

That seems right: the original mean was about 1.75 and the mean of the centered data is 0. We can even go further and check that the standard deviation hasn't changed:

R

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

Output

```
[1] 1.067628
```

R

```
# centered standard deviation
sd(centered)
```

Output

```
[1] 1.067628
```

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:

R

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

Output

```
[1] 0
```

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`:

R

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

Output

```
[1] TRUE
```

It's still possible that our function is wrong, but it seems unlikely enough that we should probably get back to doing our analysis. However, there are two other important tasks to consider: 1) we should ensure our function can provide informative errors when needed, and 2) we should write some documentation (../reference.html#documentation) for our function to remind ourselves later what it's for and how to use it.

Error Handling

What happens if we have missing data (NA values) in the `data` argument we provide to `center`?

R

```
# new data object and set one value in column 4 to NA
datNA <- dat
datNA[10,4] <- NA

# returns all NA values
center(datNA[,4], 0)
```

Output

```
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[51] NA NA NA NA NA NA NA NA NA NA
```

This is likely not the behavior we want, and is caused by the `mean` function returning NA when the `na.rm=TRUE` is not provided. We may wish to not consider NA values in our `center` function. We can provide the `na.rm=TRUE` argument and solve this issue.

R

```
center <- function(data, midpoint) {
  new_data <- (data - mean(data, na.rm=TRUE)) + midpoint
  return(new_data)
}

center(datNA[,4], 0)
```

Output

```
 [1]  1.2542373 -0.7457627  1.2542373 -1.7457627  1.2542373  0.2542373
 [7]  0.2542373  0.2542373  1.2542373         NA -1.7457627 -1.7457627
[13] -0.7457627 -1.7457627 -0.7457627 -1.7457627 -1.7457627 -0.7457627
[19] -0.7457627 -1.7457627  1.2542373  1.2542373  1.2542373 -0.7457627
[25] -0.7457627 -0.7457627  0.2542373 -0.7457627  0.2542373 -0.7457627
[31] -1.7457627  1.2542373  0.2542373 -0.7457627  0.2542373  1.2542373
[37]  0.2542373  0.2542373  1.2542373  1.2542373  0.2542373  1.2542373
[43]  1.2542373  1.2542373  1.2542373  0.2542373  1.2542373  1.2542373
[49]  1.2542373  0.2542373 -0.7457627  0.2542373  0.2542373 -0.7457627
[55] -0.7457627  1.2542373  0.2542373 -0.7457627 -0.7457627 -1.7457627
```

However, what happens if the user were to accidentally hand this function a `factor` or `character` vector?

R

```
datNA[,1] <- as.factor(datNA[,1])
datNA[,2] <- as.character(datNA[,2])

center(datNA[,1], 0)
```

Warning

```
Warning in mean.default(data, na.rm = TRUE): argument is not numeric or logical:
returning NA
```

Warning

```
Warning in Ops.factor(data, mean(data, na.rm = TRUE)): '-' not meaningful for
factors
```

Output

```
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[51] NA NA NA NA NA NA NA NA NA NA
```

R

```
center(datNA[,2], 0)
```

Warning

```
Warning in mean.default(data, na.rm = TRUE): argument is not numeric or logical:
returning NA
```

Error

```
Error in data - mean(data, na.rm = TRUE): non-numeric argument to binary operator
```

Both of these attempts result in errors. Luckily, the errors are quite informative. In other cases, we may need to add in error handling using the `warning` and `stop` functions.

For instance, the `center` function only works on numeric vectors. Recognizing this and adding warnings and errors provides feedback to the user and makes sure the output of the function is what the user wanted.

Documentation

A common way to put documentation in software is to add comments (../reference.html#comment) like this:

R

```
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)
}
```

📌 Writing Documentation

Formal documentation for R functions is written in separate `.Rd` using a markup language similar to LaTeX (https://www.latex-project.org/). You see the result of this documentation when you look at the help file for a given function, e.g. `?read.csv`. The roxygen2 (https://cran.r-project.org/package=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.

Functions to Create Graphs

Write a function called `analyze` that takes a filename as an argument and displays the three graphs produced in the previous lesson (../01-starting-with-data/) (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

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

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 `read.csv` without naming them:

R

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

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

R

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

Error

```
Error in read.table(file = file, header = header, sep = sep, quote = quote, : 'file' must be a char
acter string or connection
```

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

R

```
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:

R

```
test_data <- c(0, 0, 0, 0)
center(test_data, 3)
```

Output

```
[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`:

R

```
more_data <- 5 + test_data
more_data
```

Output

```
[1] 5 5 5 5
```

R

```
center(more_data)
```

Output

```
[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

R

```
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()
```

Output

```
a b c
1 2 3
```

R

```
# one argument
display(55)
```

Output

```
 a  b  c
55  2  3
```

R

```
# two arguments
display(55, 66)
```

Output

```
 a  b  c
55 66  3
```

R

```
# three arguments
display(55, 66, 77)
```

Output

```
 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:

R

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

Output

```
 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.

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

R

```
?read.csv
```

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

R

```
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:

R

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

Error

```
Error in read.table(file = file, header = header, sep = sep, quote = quote, : 'file' must be a char
acter string or connection
```

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 your two implementations produce the same results when both are given the same input vector and parameters?

Solution

Key Points

- Define a function using `name <- function(...args...) {...body...}` .
- Call a function using `name(...values...)` .
- R looks for variables in the current stack frame before looking for them at the top level.
- Use `help(thing)` to view help for something.
- Put comments at the beginning of functions to provide help for that function.
- Annotate your code!
- Specify default values for arguments when defining a function using `name = value` in the argument list.
- Arguments can be passed by matching based on name, by position, or by omitting them (in which case the default value is used).

(../01-starting-with-data/index.html)

(../03-loops-R/ind