Writing functions

CRI R Workshop

Why, and what?

Sometimes you may want to perform the same analysis multiple times, possibly on several sets of similarly formatted data, or you may be working with a data set that is updated periodically. Re-doing your analysis manually is time-consuming and sets you up to make mistakes. Editing code to re-run an analysis might work, but is inconvenient. Enter functions!

Functions allow you to repeat the same code using different sets of input, using a single command.

Writing functions

Functions have several parts:

- A name, so that you can call the function (as in `mean`, `sum`, or `c`)
- Arguments (technically optional)
- A body of code, which may specify what to return

The general structure of a function looks like this:

```
functionName <- function(argument1, argument2) {
  body of code that refers to argument1 and argument2
  return(result of function)
}
```

Note that the body of the function is defined within curly braces.

Let's start by creating a very simple function to learn how this works. We will define a function called `addone` to add 1 to the value that we pass to it, and then return this value.

```
addone <- function(x){
  y <- x + 1
  return(y)
}
addone(3)
```

```
## [1] 4
```

In this case, `addone` is the name of our function, which takes one argument, `x`, calculates the value of `y` as `x` + 1, and then returns `y`. When we call this function, the value that we pass to it is assigned to the argument `x`, which becomes a variable `x` within the body of the function.

This variable has the value we passed to it only within the body of the function. I.e., in the `addone` example, `y` = `x` + 1 only within the function `addone`, and calling `addone` does not define a variable called `y` in the rest of our R session. Similarly, if we use a function to do something with other pieces of data, R will make a copy of the data to work with inside the function, keeping the original data safe. In addition, every time you call a function, it starts with a clean environment; there is no memory of the last time you ran the function, the values you passed to it, and the variables inside it.

To re-phrase:

- Modifying data within a function will not affect this data in the rest of your R session
- Anything created within a function is only available outside the function if it is returned by the function
- Every time you call a function, the function environment starts with a clean slate, with no memory of values from previous calls

We can return a value from within the function by using `return`. However, this is optional - if we don't specify something to `return`, R will automatically return the result of the command on the last line of the function. For example, `addone` would have worked similarly if we had defined it as follows:

```
addone2 <- function(x){
  x + 1
}
addone2(3)
```

```
## [1] 4
```

However, it is good practice to be explicit about what the function is returning, to make it easier to understand what your function does.

You can only include one `return` command in a function, since returning a result stops the action of the function. However, this is not a limitation, since you can return multiple elements by containing them all within a vector, data frame, or list, which can then be returned.

You can pass any kind of input to a function, as long as it makes sense once the function is evaluated. So, for example, try calling `addone` with a numeric vector as input. Did it work? Now try calling `addone` with a character as input. What happens?

You can also pass a defined variable to a function, which gives the same output as passing the value of the variable directly. What output do you expect from the following?

```
x <- 25
addone(x)
```

```
## [1] 26
```

```
addone(25)
```

```
## [1] 26
```

As with defining vectors or other variables, it's important to avoid confusion by giving your function a name that doesn't already have a definition. So, it's a bad idea to name your function `c` or `mean` or `sum`. You can check whether a variable name has been defined by using the function `exist`, with a character string as input:

```
exists("addone")  # Should exist because we just defined it
```

```
## [1] TRUE
```

```
exists("otherFunctionName")  # Should not exist, unless you have defined it on your own computer!
```

```
## [1] FALSE
```

Another example: `if` statements within functions

You can combine functions with everything else that you have learned so far. For example, you can put an `if` statement into a function. Let's try this using an example from the lesson on `if` statements: an `if` statement that indicates whether a number is even or odd. The original code required us to specify the value of a variable first, and then run the line of code with the `if` statement. If we instead write a function, we can re-use this line of code much more simply.

Here is the original code:

```
a <- 21  # Set value of x
ifelse(test = a %% 2 == 0, yes = "even", no = "odd") # Check if x is even or odd
```

```
## [1] "odd"
```

Here is the equivalent function:

```
checkEvenOdd <- function(a) {
    # Original if statement; set result to a variable
        outcome <- ifelse(test = a %% 2 == 0, yes = "even", no = "odd")
    # Return the result of the if statement
        return(outcome)
}
```

Now we can run this function directly for any value of our choosing. We are essentially passing the function the value of `a` to run through the `if` statement.

```
checkEvenOdd(321)
```

```
## [1] "odd"
```

```r
checkEvenOdd(400)
```

```
## [1] "even"
```

Of course, you may never need a function that simply checks whether a value is even or odd. However, this approach can simplify other tasks as well. For example, you could use a function to rename a column, or calculate a new variable from other variables, in many different data frames. You can also use functions to do much more involved tasks, such as plotting calibration data to a pdf file, calculating calibration equations using linear or logarithmic regression, and converting raw measurement data to the variable of interest. Or, as another example, you could run and plot a multivariate analysis on multiple datasets. Functions will allow you to do these analyses with multiple data sets, in the same way each time. This is not only convenient, it also facilitates reproducible analyses.

Some notes on writing functions

As with other parts of your code, it's important to include comments in your functions to describe what they do. This is so important! This will help other people - as well as yourself in the future - understand what the function does and how to use it.

It's also essential to test your function to make sure that it does what you think it does. Try feeding it simple inputs and make sure that the output makes sense. As you write your function, it is also ok to test pieces of the syntax in the console, which will help you figure out how to craft the function - but make sure that you put all the final components into the function itself. It may also help to clear the environment before testing the function, to make sure that you haven't accidentally made it dependent on a variable that exists in your current R session but might not exist in future R sessions.

As a last tip, it often helps to keep your functions short and modular. This can make it easier to read your code and understand what your function is doing. You can always call a function within another function, which can help with organization, and also allows you to easily re-use pieces in other parts of your code.

Challenge

- Write a function that takes two arguments and adds them together.
- Write a function that converts a weight in kg to a weight in g.
- Write a function that converts a temperature in Fahrenheit to a temperature in Celsius.
- Write a function that returns the absolute value of a number without using the `abs` function.
- Dig deep and remember this from algebra class: When you have a quadratic equation of the form $ax^2 + bx + c = 0$, you can find its roots using the quadratic formula: $x = (-b \pm \text{sqrt}(b^2 - 4*ac)\ )/(2*a)$. Write a function to find both values of x given a, b, and c.
- Write a function that takes two arguments, divides one by the other, and then prints out a sentence with the result. (Hint: Use the `paste` function!)
- Write a function that takes a vector as its argument, prints a warning message if the vector includes `NA`s, and returns a vector with the mean, minimum, and maximum of the vector (i.e., with the NA values removed).

Setting default values

When you write a function, you can specify default values for specific arguments, which allows the function to run whether or not there are inputs for those arguments. For example, in the `plot` function, the default is to use `pch` = 19, which plots open circles for points, but you can specify a different `pch` value when you call the function. Similarly, the `log` function has `base` as an argument, but if you don't specify a base, it will default to e (`exp(1)`), and compute the natural log.

Defaults are defined by setting arguments equal to specific values in the initial function definition. To demonstrate, let's write a function that prints out a sentence about your favorite color.

```
printFavColor <- function(colorname = "aquamarine") {
  paste("My favorite color is ", colorname, ".", sep="")
}
```

If we pass this function a `colorname`, it will include it in the output.

```
printFavColor("turquoise")
```

```
## [1] "My favorite color is turquoise."
```

However, if we don't give it any input, it will use the default value.

```
printFavColor()
```

```
## [1] "My favorite color is aquamarine."
```

This can be useful when you want to have an option to specify a parameter, but only if it is an exception. For example, you might want to write a function that plots your data and log-transforms the y-axis only when you specify it in the function's arguments. Or, if you're using a function to calculate a calibration equation from a calibration curve, you might want to default to using a linear regression, but be able to use a logarithmic regression instead when a certain argument is equal to `TRUE`.

Challenge: Using functions with data frames

Say you have data from multiple years on stream invertebrate species richness, along with other stream characteristics: total organic carbon (TOC, percent C), current variability (cm/s), and mean temperature (degrees C). One year of this data is in the *Inverts.csv* file that you downloaded earlier. Your goal is to write a script that contains functions that can be used repeatedly to analyze multiple years of data.

Set up your script as if it were a part of your own analysis workflow: Begin the script with commented lines with the file name, your name, the date, and a description of the script, and also comment liberally throughout the script to indicate what your code does!

Your script should read in *Inverts.csv*, then contain the following functions:

- A data checking function that does the following:
  - Checks whether there are any variables that are not numeric, and prints a statement reporting the conclusion
  - Returns observations for sites that have missing values (`NA`) for any measurement variables

- A function that counts how many sites have richness greater than 40, and returns a sentence with the answer.
- A function that plots any variable in the Inverts data set against Richness. You can use `get` in the form `get("variable name")` to call a variable using its name as a character string, which may help with passing variable names into a function. If you're feeling more adventurous, set the plot's y-limits to go from 0 to the maximum of the variable being plotted.

Resources

- Software Carpentry lesson on functions
- programiz tutorial on functions in R
- R-bloggers article on writing and debugging functions