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. What output do you expect from the following?

```
a <- 25
addone(a)
```

```
## [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
```

Challenge

- Write a function that returns the absolute value of a number without using the `abs` function.
- Write a sentence that takes two arguments, divides one by the other, and then prints out a sentence with the result.
- Write a function that takes a vector as its argument, prints a warning message if the vector includes `NA`s, and returns a list with the mean, minimum, and maximum of the vector.

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`.

Other 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, 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 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. Read in this file, then write the following functions:

- Write 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
- Write a function that counts how many sites have richness greater than 40, and returns a sentence with the answer.
- Write 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.

Challenge solutions

- Here is one solution for a function that checks whether any variables are non-numeric and returns observations with missing data:

```
Inverts <- read.csv(file="../Data/Inverts.csv", stringsAsFactors=FALSE, header=TRUE)

checkData <- function(dataset){
  # Print output based on whether any variables are not numeric
  if (any(!is.numeric(dataset$Richness), !is.numeric(dataset$TOC), !is.numeric(dataset$CurrentVariabili
    print("All variables are numeric")
    }
  # Which observations have NA values for one of the measured variables?
    print("The following observations (if any) have NA values for at least one of the measured variables
    dataset %>%
      filter(is.na(Richness) | is.na(TOC) | is.na(CurrentVariability) | is.na(MeanTemperature) | is.na(
```

```
}

checkData(Inverts)
```

```
## [1] "All variables are numeric"
## [1] "The following observations (if any) have NA values for at least one of the measured variables:"

##   Site Richness  TOC CurrentVariability MeanTemperature   Type Country
## 1   20       69 0.89               47.5              NA Forest  Canada
```

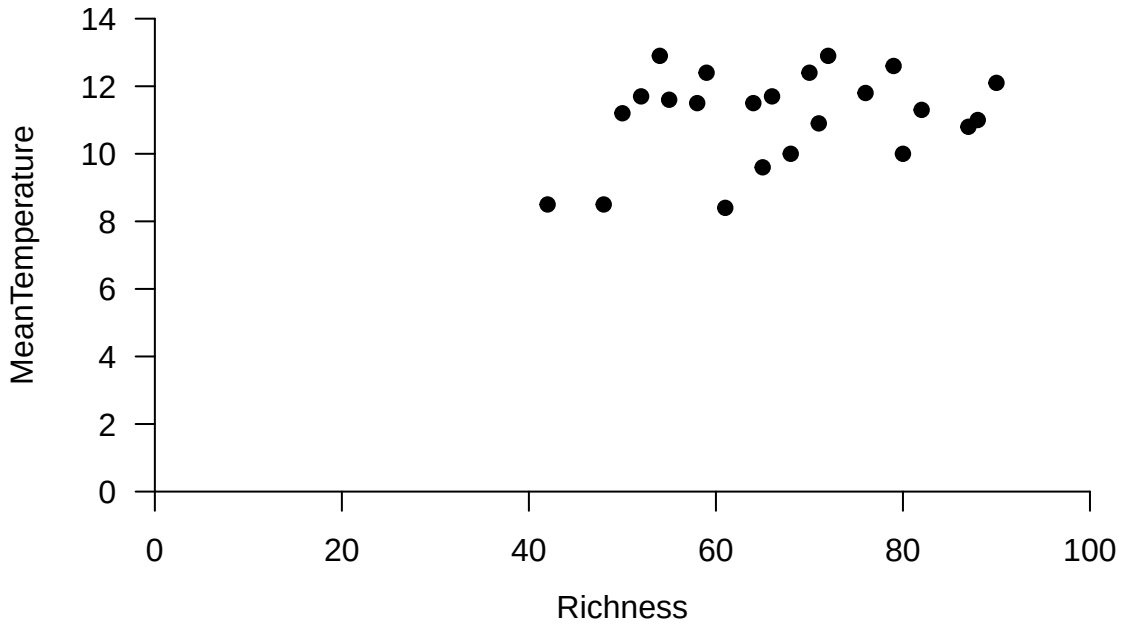
- Here is an option for a function that counts how many sites have richness greater than 40.

```
richnessThreshold <- function(dataset){
  # Count how many sites have species richness greater than 40
    richSites <- dataset %>%
      filter(Richness > 40) %>%
      tally()
  return(paste("There are ",richSites," sites that have richness greater than 40.", sep="") )
}

richnessThreshold(Inverts)
```

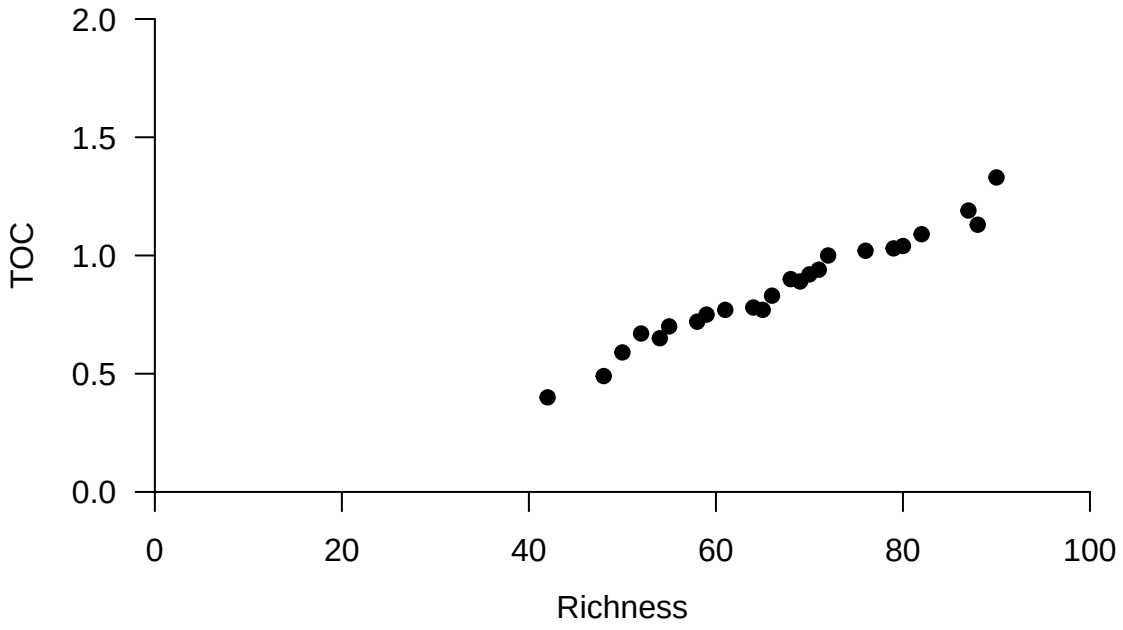
```
## [1] "There are 24 sites that have richness greater than 40."
```

- Here is an example of a function that will plot a variable in the Inverts data set against species richness.

```
plotVsRichness <- function(dataset, yvariable){
  # Find y-axis limits to be able to start at 0 (not necessary, but helps with plot aesthetics)
    ymax <- dataset %>%
              select(get(yvariable)) %>%
              unlist() %>%  # Take out of list form
              max(na.rm=T) %>%   # Find maximum value
                # Round up to the nearest two:
                        `/`(2) %>%  # Divide by two, using the division inline function as a normal fun
              ceiling %>%  # Ceiling rounds up to the nearest whole number
                        `*`(2)  # Multiply by two, using the * inline function as a normal function via
  # Make plot
    plot(get(yvariable) ~ Richness, data=dataset, ann=FALSE, axes=FALSE, pch=19, ylim=c(0,ymax), xlim=c
    # Add axes and labels
      axis(1, pos=0) # Only if the plot is set to start at 0
        mtext(side=1,line=2,"Richness")
      axis(2, pos=0, las=1)
        mtext(side=2, line=2, yvariable)
}

plotVsRichness(Inverts, "MeanTemperature")  # Plot mean temperature against richness
```




```
plotVsRichness(Inverts, "TOC")  # Plot total organic carbon against richness
```



Resources

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