Functions

Completed by Leon Woltermann

Contents

NB: The worksheet has beed developed and prepared by Lincoln Mullen. Source: Lincoln A. Mullen, *Computational Historical Thinking: With Applications in R (2018–)*: http://dh-r.lincolnmullen.com.

The best way to learn R or computational history is to practice. These worksheets contain a series of questions designed to teach you about R or different computational methods. The worksheets are R Markdown documents that include text and code together. The places where you are expected to answer questions are marked like this.

`(@) Can you make a plot from this dataset?`

Beneath each question is a space to either create a code block or write an answer.

Aims of this worksheet

After completing this worksheet you should be able to use functions and write your own.

You may find the chapter on functions from Hadley Wickham's *Advanced R* book helpful.

Explanation of functions

Functions are one of the most powerful pieces of R. A function is basically a way of taking input and producing output. For instance, we can take the numbers 1, 2, and 3 (the input) run them through the `sum()` function and produce 6 (the output).

```
sum(c(1, 2, 3))
```

```
## [1] 6
```

There are two reasons that is powerful. First, many functions in R are *pure*. That means that given the same inputs, they always produce the same outputs. When you add those numbers together, nothing will ever make you get a different answer than 6: not the packages that you've loaded, not the variables in the global environment, not the waxing and waning of the moon. That means that you can *reason* about pure functions, since their output is predictable and stable. Furthermore, it means that you can always substitute the output for the input plus the function (like a variable in algebra). For example:

```
sum(c(1, 2, 3)) < 10
```

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

```
x <- sum(c(1, 2, 3))
x < 10
```

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

The second reason that this is powerful is that functions are thoughts. The function `sum()` instantiates the idea of summing up values. Functions are like legos. They are small and insignificant in themselves, but because of the way that they connect to each other, you can use a lot of small ideas to build up a big idea in a rigorous, stable way.

Function calls

A function call has two basic parts: the name of the function, and the arguments. The name of the function comes before the parentheses which marks a function call.

```
paste(c("First", "Second", "Third"), collapse = " -> ")
```

```
## [1] "First -> Second -> Third"
```

(1) What function is being called above? What are the two arguments to the function? What do those arguments do? > The name of the function is paste. The two arguments are the vector and collapse. The function transforms the vector of three strings into one string. the argument collapse defines the string that is in between every string which is joined together from the vector.

Function definition

A function definition has three parts: the name of the function (which is the same as the name of any variable), the arguments to the function, and the body of the function that does the work. Let's define a basic function. It will be called `hello`; it will take a person's name as an input, and it will produce the output "Hello, person's name."

```
hello <- function(person) {
  paste("Hello,", person)
}
hello("Leon")
```

```
## [1] "Hello, Leon"
```

(2) What is the function name? What is the name of the single argument to the function? What is the body of the function and how is it delimited? What output does it produce? > The name of the function is hello. The name of the single argument is person. The body of the function is the function paste with the string "hello" and the input argument. The output is a pasted string of the "hello" and the input argument.

Let's make a more complicated function. This one will take some arbitrary greeting as well as a person's name, and greet that person using that name.

```
greet <- function(person, greeting = "Hello") {
  paste(greeting, person)
}
greet("Leon", greeting = "Howdy")
```

```
## [1] "Howdy Leon"
```

```
greet("Leon")
```

```
## [1] "Hello Leon"
```

Notice that we made the `greeting =` argument have a default value. Now we can call it with an explicit value (e.g., "Howdy") and the function will use that. Or we can not give any value to the `greeting =` argument, and the function will use "Hello" automatically.

(3) Write a function that greets someone with an arbitrary greeting, as in the example above, but then adds an argument `pleasantry =` which lets you specify something pleasant to say to the person. Make sure the `pleasantry =` argument has a default value.

```r
greet <- function(person, greeting = "Hello", pleasantry = "How are you?") {
  paste(greeting, person, pleasantry)
}
greet("Leon")
```

```
## [1] "Hello Leon How are you?"
```

(4) Between 1582 and 1752, England and its colonies were on the Julian calendar while most of the rest of Europe was on the Gregorian calendar. Simplifying a complex historical problem with representing dates quite a bit, we can say that dates before 1752 were "Old Style" and in 1752 and after were "New Style." Write a function called `check_date_style()` which takes an arbitrary vector of years and returns either "Old Style" or "New Style" as appropriate. Run the function on the vector below. Hint: You will need to use the `ifelse()` function in the body of your function. Look it up with `?ifelse`.

```r
years <- c(1758, 1752, 1756, 1740, 1746, 1755, 1749, 1749, 1753, 1759,
           1758, 1752, 1759, 1749, 1756, 1757, 1750, 1744, 1750, 1748)

check_date_style <- function(years) {
  ifelse(years > 1752, "New Style", "Old Style")
}

check_date_style(years)
```

```
##  [1] "New Style" "Old Style" "New Style" "Old Style" "Old Style" "New Style"
##  [7] "Old Style" "Old Style" "New Style" "New Style" "New Style" "Old Style"
## [13] "New Style" "Old Style" "New Style" "New Style" "Old Style" "Old Style"
## [19] "Old Style" "Old Style"
```

(5) One common operation to perform on data is to take data with an arbitrary range (i.e., minimum and maximum value) and scale (or normalize) it so that it has a range of 0 to 1. For instance, if I have a vector `c(1, 6, 2)` that might be normalized to `c(0.0, 1.0, 0.2)`. The formula to do that transformation is

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Below is the boilerplate for a function that does that kind of rescaling. Flesh out the function's body so that the last line of this chunk returns `TRUE` (i.e., so that the rescaled input matches the expected output). You will need to use some base R functions that we have already explored such as `max()`.

```r
rescale <- function(x) {
 return((x-min(x))/(max(x)-min(x)))
}

input <- c(-6, 3, 1, 2, 10, 3)
output <- c(0, 0.5625, 0.4375, 0.5000, 1.0, 0.5625)

all(output == rescale(input))
```

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

(6) Hard question: Can you rewrite that function as `rescale2` so that it maps the domain (the input) to an arbitrary range (the output)? In other words, can you scale the values to a range other than `0` and `1`?