Chapter 6: Programming in R

Mark Andrews
Psychology Department, Nottingham Trent University

✉ mark.andrews@ntu.ac.uk

Introduction

In this chapter, we aim to provide an introduction to some of the major programming features of R.

- We'll begin with *functions* both because they can be very simple to use and because of the major role functions play in programming in R generally.
- We will then consider *conditionals*, which allow us to execute different blocks of code depending on whether certain conditions are true.
- We will then consider *iterations*, also known as *loops*.
- This will lead on to *functionals*, which can often take the place of traditional loops in R.

Note, however, that there is more to programming in R than what we cover here.

Functions

Let’s say we have a vector of probabilities

$$p_1, p_2 \ldots p_n$$

and we want to calculate

$$\log_b \left(\frac{p_i}{1 - p_i} \right) \quad \text{for } i \in 1 \ldots n.$$

We can define a function to implement this as follows:

```
log_odds <- function(p, b){
  log(p/(1-p), base = b)
}
```

Function input arguments

Notice that in the function definition, we stated that it takes two input arguments `p` and `b`, and that the code in the body explicitly operates on `p` and `b`. As we see in the following code, we may also explicitly indicate which variables are mapped to `p` and `b`.

```r
probs <- c(0.1, 0.25, 0.5, 0.75)
log_base = 2
log_odds(p=probs, b=log_base)
#> [1] -3.169925 -1.584963  0.000000  1.584963
```

When using explicit assignment like this, the order of the arguments no longer matters. Thus, we can write the above code as follows.

```r
log_odds(b=log_base, p=probs)
#> [1] -3.169925 -1.584963  0.000000  1.584963
```

Anonymous functions

Consider the following function:

```r
f <- function(x, g){
  g(sum(x))
}
```

Here, `f` takes an object `x` and a function `g` and calls `g(sum(x))` and returns the result. We can pass in any existing R function we wish as the value of `g`, as in the following examples.

```r
x <- c(0.1, 1.1, 2.7)
f(x, log10)
#> [1] 0.5910646
```

However, we don't have to assign a name to our custom function and pass in that name. We can instead just pass in the unnamed, or *anonymous*, function itself as in the following examples.

```r
f(x, function(x) x^2)
#> [1] 15.21
```

Anonymous functions are widely used in R.

Conditionals

Conditionals allows us to execute some code based on whether some condition is true or not. Consider the following simple example.

```r
library(tidyverse)

# Make a data frame
data_df <- tibble(x = rnorm(10),
                  y = rnorm(10))


write_data <- TRUE

if (write_data) {
  write_csv(data_df, 'tmp_data.csv')
}
```

Here, we write `data_df` to a `.csv` file if and only if `write_data` is true.

if . . . else statments

Sometimes, we want to execute one code block if the condition is true and execute an alternative code block if it is false. To do this, we use an *if . . . else* statement, as in the following example.

```
use_new_data <- TRUE

if (use_new_data){
  data_df <- read_csv('data_new.csv')
} else {
  data_df <- read_csv('data_old.csv')
}
```

As we can see, if `use_new_data` is true, we read in the data from `data_new.csv`, and otherwise we read the data in from `data_old.csv`.

`for` loops

In R, there are two types of iterations or *loops*, which we informally refer to as *`for` loops* and *`while` loops*.

Let's say we had the following vector of 1000 elements to which we wished to apply the `relu` function.

```
N <- 1000
x <- seq(-0.1, 1.1, length.out = N)

relu <- function(x){
  if (x < 0) { 0 } else { x }
}
```

We can create a for loop as follows.

```
y <- numeric(N)
for (i in 1:N) {
  y[i] <- relu(x[i])
}
```

For each value of `i` from `1` to `N`, we execute `y[i] <- relu(x[i])`.

`while` loops

Unlike for loops, which iterate through each value in a sequence of values, while loops continuously execute the code body while a condition remains true.

As a very simple example of a while loop, let's say we wish to find the largest value of k such that $2^k \leq 10^6$.

```
k <- 0
while (2^(k+1) <= 10^6) {
  k <- k + 1
}
k
#> [1] 19
```

`lapply`

Functionals are functions that take a function as input and return a vector. There are many functionals in the base R language. Here, we will look at one of the most widely used: `lapply`

The `lapply` functional takes a vector and a function, applies the function to each element in the vector and returns a new list.

```r
x <- c(-1, 0, 1)
y <- lapply(x, relu)
y
#> [[1]]
#> [1] 0
#>
#> [[2]]
#> [1] 0
#>
#> [[3]]
#> [1] 1
```

Functionals with `purrr`

The `purrr` package in the `tidyverse` provides functionals like `lapply`. We can load `purrr` with `library(purrr)`, but it is also loaded by `library(tidyverse)`.

One of the main tools in `purrr` is `map` and its variants. It is very similar to `lapply`.

```
y <- map(x, relu)
y
#> [[1]]
#> [1] 0
#>
#> [[2]]
#> [1] 0
#>
#> [[3]]
#> [1] 1
```