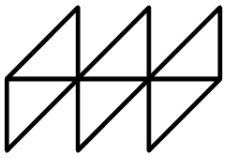

Barcelona Summer School of Demography

Module 1. Introduction to R

4. Programming tools

Tim Riffe
riffe@demogr.mpg.de

Francisco Villavicencio
fvillav1@jhu.edu

4 July 2018

Contents

1 Introduction

In this lecture we will learn some basic programming tools. This may look rather foreign if you haven't seen code like this before. If you have, then you can know that conditional programming (`if`, `ifelse()`) works similar to anywhere else, as do `for` loops. I'll try to pull off the feat of introducing these concepts and making them relevant to you in a single session. FYI, you see Francisco Villavicencio in the authors above, because for this session I've recycled some of the material from 2018 BSSD. This particular session has a much longer history since this is how intro to `R` was taught for a long time, but I've reprofiled it some to match our modus operandi: that is, when we learn a new concept, we will aim to combine it with others that we've already learned.

2 Conditional execution `if`

Conditional execution is performed in R using the command/function `if`. It works like this:

1. One first **tests a condition**.
2. **Only if** this condition is `TRUE`, the following code is executed.
3. If `FALSE`, then the following code chunk gets skipped.

The main syntax is:

```
if ( condition == TRUE ) {
    do.this
    }
```

Let's start with a very simple example

```
x <- rnorm(10)
if (mean(x) > 0){
  print("I'm feeling lucky")
}
```

Only if the condition is `TRUE`, does `R` print the result. The function `print()` is used to explicitly show an R-object in the console, like a character-string. The message `I'm feeling lucky` may or may not print to the console. It depends on the random numbers you got. In our case, we got

```
mean(x)
```

```
## [1] -0.2518674
```

I'll give more ad hoc examples of this basic idea in class.

The part that goes in the parenthesis after `if` just needs to evaluate to a single value of `TRUE` in order to execute the code afterwards.

```
if (mean(x) > 0) {
  print("I'm feeling lucky")
  print(x)
}
```

The curly brackets delimit the code that the `if` condition applies to, in much the same way es they demarcate the insides of a function. You can also place various lines of code inside the curly brackets.

In case you want to do something different if the condition is not met, you can add a second `else` clause. This would give an alternative code chunk to execute if the condition evaluates to `FALSE`:

```
if (mean(x) > 0){
  # Note the indentation in the brackets,
  #  which is optional, but increases legibility
  print("I'm feeling lucky")
} else {
  print("Today is just not my day")
}
```

```
## [1] "Today is just not my day"
```

You can, of course, have more than one condition using the operators `&` (and) and `|` (or).

```
random1 <- rnorm(1)
random2 <- rnorm(1)
if (random1 > 0 & random2 > 0) {
    print("Both random numbers are positive")
} else {
    print("At least one number is not positive")
}
```

```
## [1] "At least one number is not positive"
```

Effectively,

```
random1; random2
```

```
## [1] -0.1882012
```

```
## [1] 1.278219
```

We often use `if` statements to make code able to flexibly handle different cases. If the code is simple, you can also evaluate many such conditions at once using `ifelse()`, For instance,

```
education <- c("Secondary", "Tertiary", "Secondary",
               "Secondary", "Tertiary", "Primary")
ifelse(education == "Tertiary", 1, 0)
```

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

Things can also get complicated by nesting `if` or `ifelse()` statements:

```
ifelse(education == "Tertiary", 2,
    ifelse(education == "Secondary", 1, 0))
```

```
## [1] 1 2 1 1 2 0
```

Sometimes we use `ifelse()` to recode variables if there aren't too many unique values to recode. Otherwise, use `case_when()` from the `tidyverse`, which is a better suited to this situation. This example comes from the `case_when()` documentation. NB, I remember playing this game as a kid. The game works like this: You sit in a circle and take turns counting in sequence, if the number you're on is evenly divisible by five, you say "fizz" instead of the number, and if it's evenly divisible by seven you say "buzz", and if it's divisble by both you say "fizz buzz". It has to be fast, but if anyone makes a mistake everyone has to start over.

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.1.0       v purrr   0.3.2
## v tibble  2.1.1       v dplyr   0.8.0.1
## v tidyr   0.8.3       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
x <- 1:50
case_when(
  x %% 35 == 0 ~ "fizz buzz",  # look we're using modulo again!! %%
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)       # this is the last 'else' instruction
)
```

```
##  [1] "1"         "2"         "3"         "4"         "fizz"
##  [6] "6"         "buzz"      "8"         "9"         "fizz"
## [11] "11"        "12"        "13"        "buzz"      "fizz"
## [16] "16"        "17"        "18"        "19"        "fizz"
## [21] "buzz"      "22"        "23"        "24"        "fizz"
## [26] "26"        "27"        "buzz"      "29"        "fizz"
## [31] "31"        "32"        "33"        "34"        "fizz buzz"
## [36] "36"        "37"        "38"        "39"        "fizz"
## [41] "41"        "buzz"      "43"        "44"        "fizz"
## [46] "46"        "47"        "48"        "buzz"      "fizz"
```

Did you see what happened there? Aside from the logical testing, what kind of output did we produce? x was an integer vector, and what we got back was a character vector. Inside vectors we can't mix data types, so `R` has to coerce to the lowest (human readable!) common denominator, which is a character string.

Other friends for logical testing are `all()` and `any()`

```
(x <- rpois(10,10))
```

```
##  [1] 10 10 10 15  9  7 11  4  5 10
```

```
all(x > 0)
```

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

```
all(x > 5)
```

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

```
any(x >= 14) # at least one TRUE
```

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

NOTE

Bear in mind that `if` conditions can consist in any statement that has `TRUE` or `FALSE` as a result. The code is only executed if the answer is `TRUE`, unless an `else` condition is included.

3 Repetitive execution: `for`-loops

There are several commands available in `R` for repetitive execution, and now we will focus on `for` loops, i.e. iteration. In fact we did this without pointing it out in our tidy pipelines, for example in using `mutate()` after `group_by()`. The kind of `for`-loop construction used in `R` is common in other languages. It is often considered to be the most legible kind of iteration for this reason.

The basic form looks like this:

```
for (i in values) {
  execute.this
}
```

The parameter `i` assumes the value of each element of `values` in succession. In the first iteration, `i` takes on the first value of `values` in the first loop and executes `execute.this`. In the second run, the second value is taken from `values` and so on until the last value is taken from `values` and `execute.this` is executed the last time. You can name `i` anything you want, and `values` can be any valid vector (not necessarily integer numbers). The code chunk in curly brackets can be big or small.

Here are some examples for clarification:

```
countries <- c("AUT", "BEL", "DEU", "NLD", "GBR", "USA")
for (i in countries) {
  print(i)
}
```

```
## [1] "AUT"
## [1] "BEL"
## [1] "DEU"
## [1] "NLD"
## [1] "GBR"
## [1] "USA"
```

```
for (i in 2:4) {
  print(countries[i+1])
}
```

```
## [1] "DEU"
## [1] "NLD"
## [1] "GBR"
```

Let's say you have a vector of random numbers, and you want to compute the cumulative sum. A `for`-loop could be used to perform this task.

```
vec <- sort(runif(20))
vec
```

```
##  [1] 0.0009376239 0.0056905705 0.0846863578 0.1386488331 0.2153829345
##  [6] 0.2332514268 0.2873370240 0.3119830529 0.3602479086 0.4131935267
## [11] 0.4751363744 0.6155368378 0.6534250402 0.7135027938 0.7649292876
## [16] 0.7912686267 0.7985647339 0.8176022812 0.8725729766 0.9210091215
```

```
# Create an object with the first value of vec
cumSum <- rep(0, 20)
# Store the cumulative sum
for (i in 1:length(vec)) {
  cumSum[i] <- sum(vec[1:i])
}
cumSum
```

```
##  [1] 0.0009376239 0.0066281944 0.0913145521 0.2299633853 0.4453463198
```

```
##  [6] 0.6785977466 0.9659347706 1.2779178235 1.6381657321 2.0513592588
## [11] 2.5264956332 3.1420324710 3.7954575112 4.5089603050 5.2738895926
## [16] 6.0651582193 6.8637229532 7.6813252345 8.5538982111 9.4749073326
```

Just so you get the idea... You've seen the `cumsum()` function already, and it's actually much more efficient.

```
cumsum(vec)
```

```
##  [1] 0.0009376239 0.0066281944 0.0913145521 0.2299633853 0.4453463198
##  [6] 0.6785977466 0.9659347706 1.2779178235 1.6381657321 2.0513592588
## [11] 2.5264956332 3.1420324710 3.7954575112 4.5089603050 5.2738895926
## [16] 6.0651582193 6.8637229532 7.6813252345 8.5538982111 9.4749073326
```

4 Combining `if` and `for`-loops

The next function finds the Collatz number of a positive integer, due to the still-unproven Collatz conjecture, which (paraphrasing here) states that all positive integers can be reduced to 1 by following this iterative process: if the present iteration of the number is even, then divide by two, otherwise multiply by 3 and add 1 to it. Repeat as necessary until the number is reduced to 1. The number of steps it takes to get there is the integer's Collatz number. These have funny patterns... that are hard to see without visualizing them... Let's turn those words into a function together in class.

```
Collatz <- function(...){
    ...
}
```

Exercises

Exercise 4.1 The Fibonacci numbers are a sequence of integers whose first two elements are 1's and, afterwards, each additional element is the sum of the two preceding ones: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Formally,

$$F_1 = F_2 = 1 \quad \text{and} \quad F_n = F_{n-2} + F_{n-1} \quad \text{for} \quad n > 2\,.$$

- Using a `for`-loop, compute the first 30 Fibonacci numbers.
- It is known that the limit of the quotient of two subsequent Fibonacci numbers gives the famous **golden ratio** or **golden number** φ,

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \varphi = 1.6180339887...\,,$$

  which is observed in several natural phenomena. Using the first 30 Fibonacci numbers, create a vector with the quotients F_{n+1}/F_n for the different values of n. (Hint: It is a very simple operation, just think how to manipulate and do calculations with vectors in R.) Check that the limit is effectively close to the golden number φ (last numbers of your vector F_{n+1}/F_n).

- Finally, plot the vector F_{n+1}/F_n and visualize how the quotients oscillate at the beginning but converge rapidly to φ.

Exercise 4.2 This exercise presents a typical programming exercise: Computing square root of a number using Newton's method. We can define the square-root function as:

$$\sqrt{x} = y \quad \text{such that} \quad y \geq 0 \quad \text{and} \quad y^2 = x \, .$$

How does one compute square roots? For instance, Wikipedia lists 14 different ways, but we will exercise our programming skills with only one of them which uses Newton's method of successive approximations. As first step, we need an **initial guess** $\tilde{y}$ for the value of the square root of a number x. Then, the method works as follows:

1. First, we compute **quotient** between x and our guess $\tilde{y}$, $q = x/\tilde{y}$.
2. Next, we compute the **average** between $\tilde{y}$ and q.
3. This average becomes the **new guess** $\tilde{y}$ for the following step.

By updating the guess, we get closer to the actual square root. For example, we can compute the square root of $x = 3$, supposing that our initial guess is $\tilde{y} = 2$. Hence, one has basically 3 steps which have to repeat in order to update the *guess* and the subsequent outcomes, as shown in the following table. Compare the number of iterations you need for each of the three routines, by testing different initial guesses.

Guess: $\tilde{y}$	Quotient: $\frac{x}{\tilde{y}}$	Average: $\frac{\text{Quotient}+\tilde{y}}{2} = \text{new.guess}$
2	$\frac{3}{2} = 1.5$	$\frac{1.5+2}{2} = 1.75$
1.75	$\frac{3}{1.75} = 1.7143$	$\frac{1.7143+1.75}{2} = 1.7322$
1.7322	$\frac{3}{1.7322} = 1.7319$	$\frac{1.7319+1.7322}{2} = 1.7321$
1.7321	...	...

Exercise 4.3

Stable population theory has played an important role in demographic methods and demographic thought. Happy to expound if anyone is interested, but all you need to know for this problem is that there is a quantity that we call the *intrinsic growth rate* (a.k.a. Lotka's r, Lotka's parameter, the Malthusian parameter). If you have a fertility schedule and a mortality schedule, and if you assume that they remain unchanged, then it doesn't matter what the population structure looks like when you start: there is only one stable population structure that it will head towards and it is uniquely determined by fertility and mortality (migration is assumed zero). Further, the population size each year either grows or shrinks at a constant rate, and that's the intrinsic growth rate, r. Coale (1955) (better Coale (1957)) came up with an **iterative** procedure to find it. There are other ways, but this one is pretty neat. Here's a link if you want to see the paper: https://www.jstor.org/stable/2172513

The things you need: 1. $L(x)$, which you can grab from any lifetable 2. $f(x)^f$ ASFR for girl births (strong assumption, don't get me started) 3. x a vector of ages. Make sure all three of these things are the same length (i.e. $f(x)^f$ in pre- and post-fertile ages is just 0)

Steps to follow:

1. calculate

$$R(0) = \sum L(x) f(x)^f$$

2. assume a parameter $T = 29$
3. calculate a guess at r, the first of several, call it r^i, the i^{th} guess:

$$r^i = \frac{log(R(0))}{T}$$

4. Now in a for-loop we update r^i in two steps.
   i) calculate a residual
   $$\delta = \sum e^{(-r^i x)} f(x)^f L(x) - 1$$
   ii) update r^i using
   $$r^{i+1} = r^i + \frac{\delta}{T - \frac{\delta}{r^i}}$$
5. Repeat step 4 until δ is teeny tiny, i.e. until

$$1 = \sum e^{(-r^i x)} f(x)^f L(x)$$

Usually this takes less than 10 iterations, but you can let it go for more than that.

When you get the loop written, wrap it in a function whose arguments are $f(x)^f$, $L(x)$, and x, and which returns r. I'll think of a way for you to scale this up into a pipeline.

References

Coale, Ansley J. 1955. "The Calculation of Approximate Intrinsic Rates." *Population Index*, 94–97.

———. 1957. "A New Method for Calculating Lotka's R–the Intrinsic Rate of Growth in a Stable Population." *Population Studies*, 92–94.