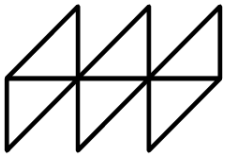

Barcelona Summer School of Demography

Module 1. Introduction to R

3. Functions

Tim Riffe
tim.riffe@ehu.eus

5 July 2023

Contents

1 Functions

We've thus far been writing code that consists in data being read in, doing stuff to it using functions (so far mostly from the tidyverse), assigning new objects from time to time, and plotting using `ggplot2` functions. Think of the smallish tidy pipelines used for each dataset on Tuesday. That type of coding is common in research practice, and we refer to that kind of code as `R` scripts. It's often once-off code, and that's just fine. But what if . . . what if you wanted to re-use the code a lot? What if you had a method of your own, your or simply own way of doing things that you want to be able to repeat? Then do you really want to type all that code over and over in the future, have to remember it? What if you did that— always retyping the same scripts repeatedly — and then decided to change how to do it? Maybe you fix a bug, make it more robust or faster, or simply have a better way? Then do you have to go and hunt down all the times you wrote that code to fix them too? That would be horrible. It's not a good use of anyone's time, and it's also fragile. We can do better. We can write functions.

2 The anatomy of a function

Here's an oldy but goody:

```
hello <- function(your_name){
    paste0("Hello ", your_name, ". Have a nice day!")
}
hello("Tim")
```

```
## [1] "Hello Tim. Have a nice day!"
```

1. `hello` is the name of the new function, which we know because
2. `<- function(){}` is being assigned to it, which creates functions
3. The thing in the parentheses `(your_name)` is the name of an argument to be used
4. The part inside `{ }` is the body of the function, presumably making use of `your_name`.
5. In this particular case we're using the function `paste0()`, which concatenates text, including our argument `your_name`.

Those are the parts of a function. The function can have more arguments, which could be any kind of data of any size and shape, and the stuff that happens inside the function (inside `{ }`) should know what to do with the arguments. It should produce a result, and return it. For our little function above, it returns a character string (the last thing evaluated). In our case it printed to console because we didn't assign it to anything. We can be more explicit about that by telling it what to return:

```
# Identical in this case
hello <- function(your_name){
    out <- paste0("Hello ", your_name, ". Have a nice day!")
    return(out)
}
hello("Tim")
```

```
## [1] "Hello Tim. Have a nice day!"
```

So here's an outline:

```
function_name <- function(arguments){
    # do stuff with arguments
    return(result)
}
```

Functions can be very large. Let's look at some:

```
lm # type into console with no parentheses to see the guts of a function
```

3 Modularity

As you can see in our cheap example, functions can use other functions, as long they can be found. I think we should start straight away with an exercise. Below I give a bunch of equations for lifetables. Start with a variable `mx`, a.k.a. $m(x)$, or $\mu(x)$ if you can pretend it's continuous. We'll also need a helper-variable `ax` that tells us something about mortality within age intervals- This is one we usually assume something for, and we can be lazy and just say it's always 0.5 for single age data.

If the mortality rate is constant in the interval, then we have some relationships that we can program as little functions. I think it's always a good idea to write functions with test data, so how about you use this as your test `mx` and `ax`:

```
omega <- 110 # last age
x     <- 0:omega
a     <- 0.00022
b     <- .07
mx    <- a * exp(x * b)
ax    <- rep(.5, length(mx))
```

A note before we start making little functions: Please try to do this on your own. Solutions will be provided in the session script, available separately after the session.

3.1 conditional probability of death in an age interval: $q(x)$

$$q(x) = \frac{m(x)}{1 + (1 - a(x)) \cdot m(x)}$$

Write a function called `mxax_to_qx()`, taking arguments `mx` and `ax` and returning a variable `qx`. Note, `mx` and `ax` should be the same length (one element per age class). It's common to declare that the very last value of $q(x)$ is 1, i.e. that no one survives beyond that age.

```
# mxax_to_qx <-
```

3.2 conditional survival $p(x)$

$$p(x) = 1 - q(x)$$

3.3 survivorship, $l(x)$

Survivorship (a.k.a the survivor curve) is the cumulative product of $p(x)$. Tip: there is a function called `cumprod()`.

$$l(x) = \prod_{i=0}^{x-1} p(i)$$

3.4 death distribution, $d(x)$

The lifetable death distribution, $d(x)$ is the probability at birth of dying in a given age.

$$d(x) = l(x) \cdot q(x)$$

Or you could also think of it as the decrement of $l(x)$

$$d(x) = l(x) - l(x+1)$$

Can you make a function `lxqx_to_dx()`? What about `mxax_to_dx()` (make it use the previous functions!), or just `lx_to_dx()`? For this last one, note it's common if ω is at a very high age so that you can assume that no one survives beyond that age.

3.5 Lifetable exposure, $L(x)$

What I call lifetable exposure is supposed to mean something like the total lifetable person-years lived between age x and $x+1$.

$$L(x) = l(x) - (1 - a(x)) \cdot d(x)$$

write `lxax_to_dx()`

3.6 Total remaining survivorship, $T(x)$

$$T(x) = \sum_{i=x}^{\omega} L(i)$$

Try to write `Lx_to_Tx()`. This is tricky since we haven't talked about loops, but you can do it by creatively combining `rev()` (flip a vector backwards) and `cumsum()`.

3.7 Life expectancy, $e(x)$

The average length of life remaining at each age in the lifetable.

$$e(x) = \frac{T(x)}{l(x)}$$

Can you write a function `lxTx_to_ex()`? What about `mxax_to_ex()`? If you do the second one, try to make it modular, i.e. use the little functions written earlier.

4 The real exercise

By now we have a nice collection of lifetable transformation functions. As is, these are little utility functions. Note, if you have a `data.frame` with `mx` and `dx` as columns, you can make a whole lifetable with a single call to `mutate()`!

```
data.frame(mx=mx, ax=ax) %>%
mutate(LT,
       Age = 0:n(),
       qx = maxax_to(qx),
       ...)
```

Now turn that into a lifetable function, you've now made a collection of small functions that exhibit modularity. That's the goal! But sometimes things get complicated inside functions. The trick is to find units of code that seem generalizable, and turn them into functions.

Let's take a break to look at some functions in the `DemoTools` package, here. We'll see some loops being used there, and I might narrate them, but we'll get more into that tomorrow. A nice and under-appreciated way to learn to code is to read it, so let's just do that a while.

5 Exercises

Exercise 1.3.1:

Read in the dataset `Data/hmd.csv.gz`:

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to b
```

```
HMD <- read_csv("Data/hmd.csv.gz")
```

```
## Rows: 1038072 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): country, sex
## dbl (4): year, age, mx, ax
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

This dataset is the entire HMD, where there is a lifetable subset for each unique combination of `country`, `sex`, and `year`. Complete the lifetable for the entire dataset **using your new function in a tidy pipeline**.

Exercise 1.3.2:

$e(x)^\dagger$, pronounced e-dagger is the average year of life lost due to death in the lifetable, and one way to calculate it looks like this:

$$e(x)^\dagger = \sum_{i=x}^{\omega} \frac{d(i)}{l(x)} e(x)$$

That is to say, this is a function patterned by age $e(x)^\dagger$. If you want to calculate this for each age then you'll probably want to write a loop, which we've not done yet. But you can calculate it for age 0 without a loop, in which case:

$$e(0)^\dagger = \sum_{0}^{\omega} d(x)e(x)$$

Write a function that does this, and calculate $e(0)^\dagger$ for each Country, Sex, and Year. Also save $e(0)$, i.e. the value of $e(x)$ where `Age == 0`. Are you going to do this with `mutate()` or with `summarize()`? Can you plot the relationship between $e(0)$ and $e(0)^\dagger$? Make a scatterplot of this. If it's super overlapped try transparency. Maybe add a `geom_smooth()` on top of it, your choice of method.