Intro to Programming with R for Political Scientists

Session 6: Writing Functions

Markus Freitag

Geschwister Scholl Institute of Political Science, LMU

19.07.2021

Overview

1. Intro + R-Studio and (Git)Hub
2. Base R & Tidyverse Basics
3. Data Wrangling I
4. Data Wrangling II
5. Data Viz
6. **Writing Functions**

Workflow

- Navigate to `Session Scripts` and open `Session_6_script.R`.
- You will see a pre-formatted Script with all the steps I do on the slides.
- Explore as you follow.
- If you have a second monitor, great! If not, split your screen.

Functions and Functional Programming in R

A Primer on Functional Programming

Even though not all functions are **pure**[1], R is a kind of a functional programming language as it has **first-class** functions.

> In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have first-class status. Some of the “rights and privileges” of first-class elements are:
>
> - They may be named by variables.
> - They may be passed as arguments to procedures.
> - They may be returned as the results of procedures.
> - They may be included in data structures. (**Abelson et al. 1996**: 102)

Because of these features, R "**lends itself to a style of problem solving centred on functions.**"

[1] **NOTE:** Not all functions are **pure** as some functions output changes even with stable input (e.g. `rnorm()`).

Recap: Anatomy of a Function

- We have already written a very basic function on our own (session 2) and used a ton of base and user-written ones.
- **Recall:** Functions are objects. In math: a function maps elements from one set to another.
- Input x, rule f, output $f(x)$. This is pretty much the same in programming.
- A function in R consists of three components: an **argument list**, a **body**, and an environment.

Syntax:

```
name <- function(COMMA-SEPARATED LIST OF ARGUMENTS) {
  FUNCTION BODY / OPERATIONS
  return(THE VALUES OR OBJECTS WE WANT OUR FUNCTION TO RETURN)
}
```

Note: `return()` is not strictly necessary because R returns the last object you defined within the function by default. But making explicit what gets returned is **good style**.

Function Environment

- A function in R consists of three components: an argument list, a body, and an **environment**.

> When a function is called, a new environment (called the evaluation environment) is created, [...]. This new environment is initially populated with the unevaluated arguments to the function; as evaluation proceeds, local variables are created within it. (**R Language Definition**)

- So what happens is that instead of in your global environment, objects you create within a function are local and only the output defined by `return` gets passed on.

- **Lexical Scoping**/How functions find values: The first place where functions "look" for values of named objects is inside the function. It only then turns to the parent environments. The function environment is fresh every call.

An Example

```
fun <- function(x = 3) { # we can specify a default value
  intermediate <- (2 * x + 3) / sqrt(3)
  output <- data.frame(input = x, output = intermediate)
  return(output) # good practice
}

fun()
```

```
##   input   output
## 1     3 5.196152
```

```
fun(2)
```

```
##   input   output
## 1     2 4.041452
```

An Example

Scoping:

```
x = 7

fun()
```

```
##   input   output
## 1     3 5.196152
```

Control Flow

> In computer science, **control flow** (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

- There are two basic tools to control the flow of our code, especially in functions: **conditions/choices** and **loops**.
- **Loops**: Often, we want function to iterate over inputs. Most programming languages use `for` loops for this. In R, `for` loops iterate over elements in vectors:

```
for (element in vector) DO SOMETHING
```

- **Choices**:

```
if (CONDITION) IS TRUE DO SOMETHING else DO SOMETHING ELSE
```

Control Flow: Examples

```
fun("hello")
```

```
## Error in 2 * x: non-numeric argument to binary operator
```

```
fun <- function(x = 3) {

  if (is.numeric(x)) {

  intermediate <- (2 * x + 3) / sqrt(3)
  output <- data.frame(input = x, output = intermediate)
  return(output)

  } else {

  stop("The input you provide has to be numeric.")

  }
}
```

Control Flow: Examples

```
fun("hello")
```

```
## Error in fun("hello"): The input you provide has to be numeric.
```

`if` operates only on scalar booleans (i.e. a single `TRUE/FALSE`)...

```
set.seed(666)

x <- rnorm(10)

if (x < 0) {
  "negative"
} else {
  "positive"
}
```

```
## [1] "positive"
```

Mostly used in function building/when a scalar condition is needed/sufficient (see previous slide).

Control Flow: Examples

Combining the scalar `if` with a `for` loop to "grow" an object telling us whether each element of `x` is `positive` or `negative` iteratively:

```
nested <- function(z = NULL) { # setting default to NULL

  y <- vector(mode = "character", length = 10) # initialise, e.g., an empty vector; preallocate memory
# y <- NULL This works too, but R has to grow, copy, paste and allocate memory at each step. This is slow.

  for (i in 1:length(z)) { # We can reference the element with whatever name, i is just convention.

    if (z[i] < 0) {
      y[i] <- "negative"
    } else {
      y[i] <- "positive"
    }
  }

  return(y)

}

nested(z = x)
```

```
##  [1] "positive" "positive" "negative" "positive" "negative" "positive"
##  [7] "negative" "negative" "negative" "negative"
```

Control Flow: Examples

We already got introduced to **vectorized** versions of if statements: `if_else()`/`ifelse()` and the more general `case_when()`:

```
vectorized <- function(z = NULL) {

  ifelse(x < 0, "negative", "positive")

}

vectorized(z = x)
```

```
##  [1] "positive" "positive" "negative" "positive" "negative" "positive"
##  [7] "negative" "negative" "negative" "negative"
```

That's much easier (and faster)!

Vectorization

- We already got introduced to vectorized versions of if statements: `if_else()` / `ifelse()` and the more general `case_when()`.

What is meant by "vectorized"?

- You often read the following: it means that the function is applied on every element of a vector/list at **once**.

Vectorization

```
fun <- function(x = 3) {
  intermediate <- (2 * x + 3) / sqrt(3)
  output <- data.frame(input = x, output = intermediate)
  return(output)
}

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

```
##   input   output
## 1     1 2.886751
## 2     2 4.041452
## 3     3 5.196152
```

Vectorization

- We already got introduced to vectorized versions of if statements: `if_else()` / `ifelse()` and the more general `case_when()`.

What is meant by "vectorized"?

- You often read the following: it means that the function is applied on every element of an vector/list at **once**.
- Except it's not literally doing that...

Technical Fine Point: At the core, everything you type in R is a vector. **Actually**, in vectorized functions, the vector gets just passed on to the compiled code of a lower-level language (e.g. C++) which itself consists of... loops.

Do I need to explicitly write a loop?

- I like loops[2], but in R they are sometimes slow **if not done properly** (see sec. Ch. of the **R Inferno**). What makes loops slow is the stuff you do inside of them.
- There are a few ways out:
  - Vectorising the stuff inside. E.g., if you can replace a nested `for` loop containing `if` statements with `ifelse()`, do it. **Recursion**) (`if`) is also slower than loops in R.
  - Another way is to use **functionals**: functions that take a vector and other functions as arguments.
  - For instance, with `lapply()` from the `*apply()` family or the equivalent but optimized tidyverse alternative `purrr::map()`, we can make **applying a function to every element of a vector and returning a list** often a one-liner.

[2] **Comment:** That's why I also like Julia for some (simulation) tasks. I can get away more easy with writing **fugly loops**.

Do I need to explicitly write a loop?

Why do these **functionals** work well?

They...

- automatically pre-allocate memory (we can do that manually too, see Ex. a few slides ago!).
- make use of compiled C code.

NOTE: They are not really vectorized functions (or faster) but make **syntax cleaner** and make it easier to not screw up.

The use of functionals to avoid looping is related to/a core topic of Hadley-Style **functional programming**.

Functionals

The syntax of the `*apply()` family and `purrr::map()` is virtually identical.

We can also specify the data structure we want to get with `map_*`:

```
is_positive <- function(z) {
    if (z < 0) {
      "negative"
    } else {
      "positive"
    }
}

purrr::map_chr(x, is_positive)
```

```
##  [1] "positive" "positive" "negative" "positive" "negative" "positive"
##  [7] "negative" "negative" "negative" "negative"
```

Functionals

```
purrr::map_df(penguins, mean, na.rm = TRUE)
```

```
## # A tibble: 1 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g   sex
##     <dbl>  <dbl>          <dbl>         <dbl>             <dbl>       <dbl> <dbl>
## 1      NA     NA           43.9          17.2              201.       4202.    NA
## # ... with 1 more variable: year <dbl>
```

```
penguins %>%
  summarise(across(.cols = everything(), mean, na.rm = TRUE))
```

```
## # A tibble: 1 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g   sex
##     <dbl>  <dbl>          <dbl>         <dbl>             <dbl>       <dbl> <dbl>
## 1      NA     NA           43.9          17.2              201.       4202.    NA
## # ... with 1 more variable: year <dbl>
```

Functionals

Functionals can take anonymous functions...

```
purrr::map_df(penguins, function(x) is.numeric(x))
```

```
## # A tibble: 1 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex
##   <lgl>   <lgl>  <lgl>          <lgl>         <lgl>             <lgl>       <lgl>
## 1 FALSE   FALSE  TRUE           TRUE          TRUE              TRUE        FALSE
## # ... with 1 more variable: year <lgl>
```

Anonymous functions are also quite useful in combination with dplyr...

```
penguins %>%
  select(function(x) is.numeric(x)) %>%
  head(1)
```

```
## # A tibble: 1 x 5
##   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g  year
##            <dbl>         <dbl>             <int>       <int> <int>
## 1           39.1          18.7               181        3750  2007
```

Functionals

There are also a few useful shorthands to make this less verbose...

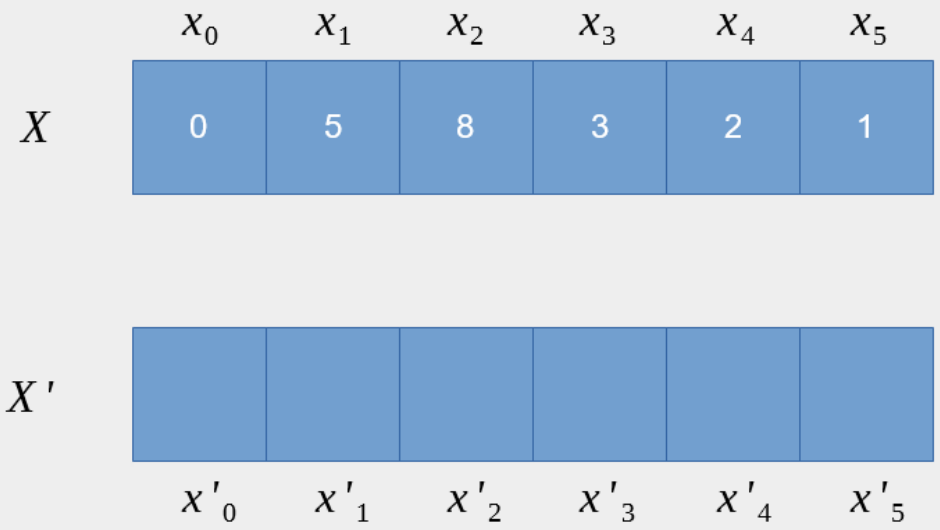
```
purrr::map_df(penguins, ~ is.numeric(..1))
```

```
## # A tibble: 1 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex
##   <lgl>   <lgl>  <lgl>          <lgl>         <lgl>             <lgl>       <lgl>
## 1 FALSE   FALSE  TRUE           TRUE          TRUE              TRUE        FALSE
## # ... with 1 more variable: year <lgl>
```

You can use `..1` (or `.x` and `.y`) and so forth for multiple-argument functions.

Functionals

$$X' = map(X, f) \quad f(x) = x + 1$$



Source: **Wikipedia**. Maps are common in pretty in programming languages.

Examples: Functional Programming and ggplot

Combining map() and ggplot()

We have already learned facets as a way to make a multi-panel plot. But what if we just want multiple plots? Copying can be tedious.

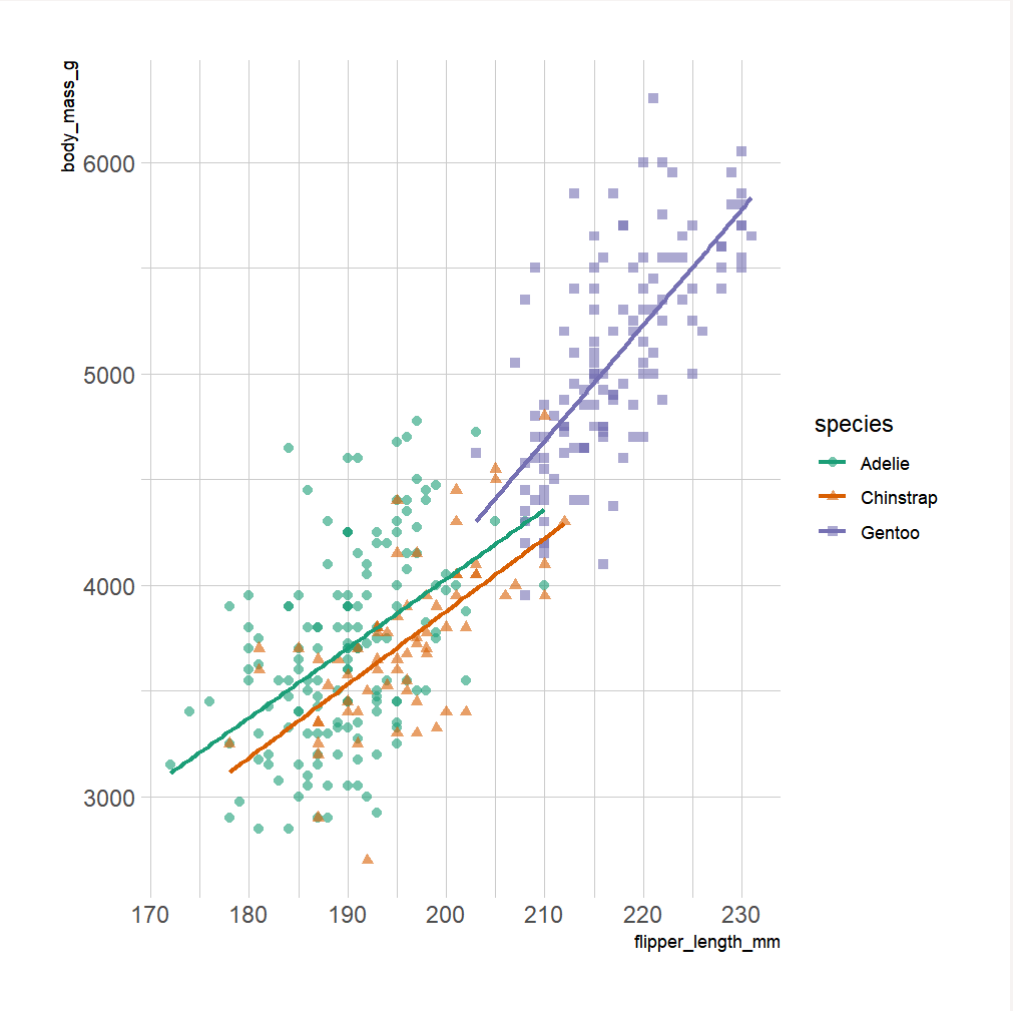
Witness the power of the functional approach...

```
penguin_scatter <- function(df, x, y, color = NULL, shape = NULL) {

  ggplot(df, aes_string(x = x, y = y, color = color, shape = shape)) +
    geom_point(size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    scale_colour_brewer(palette = "Dark2") +
    theme_ipsum()

}

penguin_scatter(penguins,
  x = "flipper_length_mm",
  y = "body_mass_g",
  color = "species",
  shape = "species"
)
```



Combining map() and ggplot()

Getting a tibble of all distinct combinations of names of numeric variables (not gonna lie, took some time to cook it up - specifying it manually would've probably been easier)...

```
combs <- penguins %>%
  select(function(x) is.numeric(x), -year) %>% # get only numeric cols, except year
  names() %>% # get a character vector of column names
  tibble(y = ., x = .) %>%  # put it into two tibble columns
  tidyr::expand(x, y) %>% # all combinations of a variable in a data set
  transmute(y = pmin(y, x), x = pmax(y, x)) %>% # some "sorting" shenanigans to get distinct combs
  distinct() %>% # cleaning stuff that did not get picked up in the step above
  filter(y != x) # "
combs
```

```
## # A tibble: 6 x 2
##   y              x
##   <chr>          <chr>
## 1 bill_depth_mm  bill_length_mm
## 2 bill_depth_mm  body_mass_g
## 3 bill_length_mm body_mass_g
## 4 bill_depth_mm  flipper_length_mm
## 5 bill_length_mm flipper_length_mm
## 6 body_mass_g    flipper_length_mm
```

Combining map() and ggplot()

Letting the magic happen with `pmap()`.

> **Function Description:** These functions are variants of map() that iterate over multiple arguments simultaneously. They are parallel in the sense that each input is processed in parallel with the others, not in the sense of multicore computing. They share the same notion of "parallel" as base::pmax() and base::pmin(). map2() and walk2() are specialised for the two argument case; pmap() and pwalk() allow you to provide any number of arguments in a list. Note that a data frame is a very important special case, in which case pmap() and pwalk() apply the function .f to each row.

```
plots <- pmap(combs, function(x,y) penguin_scatter(penguins,
  x = x,
  y = y,
  color = "species",
  shape = "species"
))
```

Combining map() and ggplot()

```
patchwork::wrap_plots(plots) # wrap_plots makes it easy to take a list of plots and add them into one composition, along with layout specifications.
```

Normalize Writing Functions

- On your way towards a proficient R user, you should try to integrate writing functions in your workflow. Mastering functions (and OOP) also makes switching between languages easier!

When to write a function?

- Whenever you copy some block of code a few times, there surely is a way to save some lines with a function. This makes your code much more readable!
- A typical application case in statistics is simulations.
- If you want to plot multiple things, functions become extremely handy!

Stylistic Rules

- To make your functions readable for your future self, name them and their arguments properly. Specificity > brevity.
- Comment more complex functions extensively or you will be doomed!

You wrote a neat function you need a lot in your daily workflow? Consider **packaging it**!

A Brief Aside on Debugging in R-Studio

While you should be relatively well equipped to understand some errors or warnings that R throws, there will often be situations where you simply have no clue.

When writing a function, avoid this by writing several small functions instead of a single large one.

- Most of the time, you can get away by googling errors and consulting, e.g. **stackoverflow**.
- Or, you can use `errorist` to automatically search errors and warnings when they arise.

You can also use **R-Studios debugging tool** to peek under the hood of a function, execute it line by line, and spot the error.

I mostly enter the debugging mode with:

```
debugonce(your_function)
your_function()
```

Some Words on Linting

When writing functions, keeping your code clean is especially important for legibility.

While the first `session_script` contains some information on structuring scripts, we did not touch on code so far.

Linting means stylistically standardizing code/ a software that tells you which lines of code are badly formatted/have errors. Other languages have fix conventions, R does not. But see the **tidyverse style guide**.

While Rstudio has some auto code formatting and code and snippet suggestions, it does not natively provide code linting.

My Tipp is to use `styler`, an Rstudio addin.

Let's take a look...

Session 5: Problem Set

If you want to master functions there is only one way...

Get your hands dirty!

A (Scientific) Worflow with R: A Few Pointers

Workflow

Ideally, we want every single piece of our (scientific) work to be reproducible.

This includes:

- Having clean scripts that are readable, and purpose specific (i.e., one for data wrangling, one for the analysis, etc.)
- Using things like `R projects` and the `here` package.
- Using a version control system (e.g., Git) to track changes to scripts and data.
- Making the text documents fully reproducible, e.g., using Rmarkdown rather than word or LaTeX.
- Keeping all your package and software dependencies stable (`renv` and **docker** containers).

Workflow

This is a long list, but after this course, we could check almost any item on it.

Except, the last point. Unfortunately, this is perhaps the most demading thing to do.

While using `renv` is fairly **simple**, I would recommend **combining it with docker** only in larger projects.

I am, however, pretty sure, that in 3-5 years, submitting replication materials will have to be done via a Docker container by default.

Pointers:

Here is a collection of useful links for further reading.

- **Rmarkdown**: See, e.g., **this** and **this** book. But I recommend just learning by doing, and consulting the books or google on the fly.
- For a nice and super easy to use "template" for paper drafts or submission-ready papers, see the new `reproducr` package by Julia Schulte-Cloos (GSI).
- Git(Hub): For learning Git, I highly recommend **getting a bit more familiar with the shell first** (or **here**). After this, go, e.g., on **this** site and practice/learn Git with a simulator. See also **here**, **here**, **here**, **here**.
- For using docker and R, take, e.g., **this**, **this**, or **this** article as starting point.

Other Pointers

- Data Viz: for a very exhaustive list of pointers, see **here**.

The great **CRAN Task View** collections. A few examples:

- **Machine Learning**
- **Econometrics**
- **Bayesian Inference**

Q & A