Writing Functions

Dr Andrew J. Stewart

E: drandrewjstewart@gmail.com
T: @ajstewart_lang
G: ajstewartlang

R is a functional language

This means that every command you type in R, is actually a call to a function. You have already used in-built functions like `sum()`, as well as functions from packages (e.g., the function `summarise()` from `{dplyr}`).

One of the powerful things about functional languages it that you can create your own functions that you can call with one command.

Very often, you'll find yourself repeating the same chunk of code to complete a particular task (e.g., you might want to plot a bunch of visualisations for a large dataset where everything stays the same from one visualisation to the next, but maybe you have different variables on the x- and y-axis in each plot). If you're using the same code over and over again, you probably want to turn that code into a function.

The basis of a function

There are three components to a function in R:

- The `body()`, the code inside a function.
- The `formals()`, the list of arguments that control how you call the function.
- The `environment()`, the “map” of the location of a function’s variables.

Creating a new function

Let’s write a function that takes a number, and adds 5 to it.

```
add_five <- function(x) x + 5
```

So, we can then run the following line:

```
add_five(1)
```

We get the answer 6.

Creating a new function

We can examine the body, formals, and environment of the function we created as follows:

```
> body(add_five)
x + 5
> formals(add_five)
$x
> environment(add_five)
<environment: R_GlobalEnv>
```

Functions can call themselves

```
factorial_of_a_number <- function(x) {
  if (x == 1) {
    return(x)
  } else {
    return(x * factorial_of_a_number(x - 1))
  }
}

> factorial_of_a_number(4)
[1] 24
```

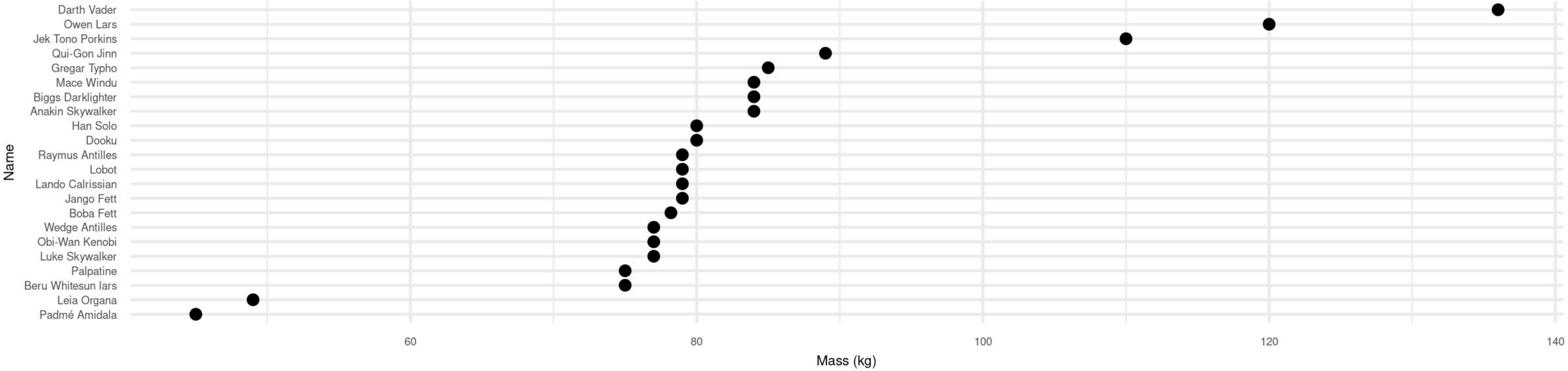
A function to produce a plot...

```r
my_plot_of_humans <- function(my_tibble, my_x, my_y, my_x_label, my_y_label) {
    ggplot(data = filter(my_tibble, species == "Human" & !is.na(eval(my_y))),
        aes(x = fct_reorder(eval(my_x), eval(my_y)), y = eval(my_y))) +
    geom_point() +
    coord_flip() +
    labs(x = my_x_label, y = my_y_label) +
    theme_minimal() +
    theme(text = element_text(size = 5))
}
```

I’ve written the above function to work with the `starwars` dataset in the `tidyverse` that will display a visualisation of characters in the dataset that are human - as parameters the function takes a tibble, a variable for the x-axis, a variable for the y-axis, and labels for the x and y-axes. I can call this function to produce a plot of heights and names, mass and names etc.

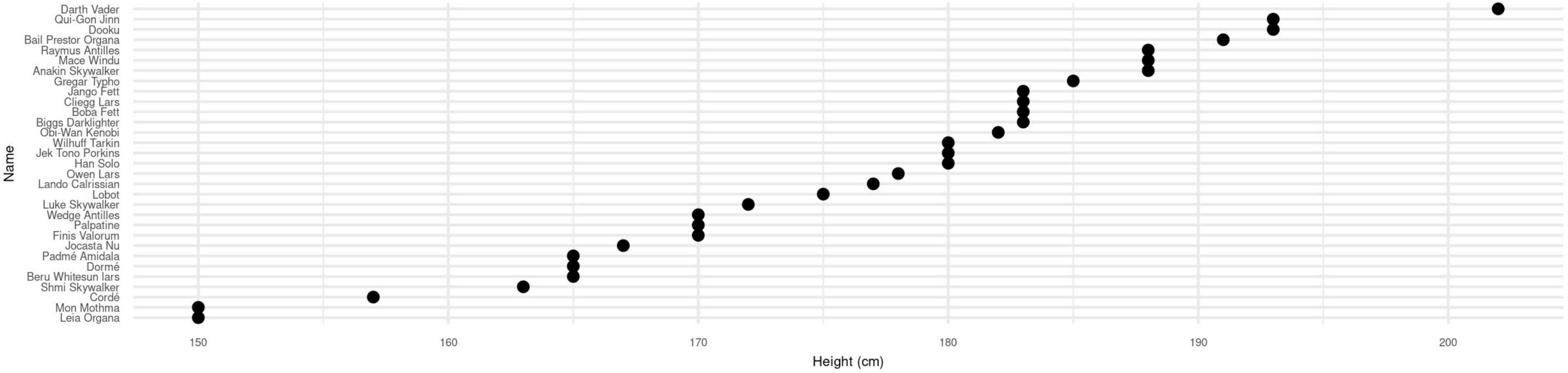
Calling the function to plot the mass variable

```
my_plot_of_humans(my_tibble = starwars,
                  my_x = quote(name),
                  my_y = quote(mass),
                  my_x_label = "Name",
                  my_y_label = "Mass (kg)")
```



Calling the function to plot the height variable

```
my_plot_of_humans(my_tibble = starwars,
              my_x = quote(name),
              my_y = quote(height),
              my_x_label = "Name",
              my_y_label = "Height (cm)")
```



You could write your entire script as a set of functions

If you’re reading in data, tidying it, visualising it, modelling it, and conducting follow up analyses you could write your entire script as a set of functions - one function for each step (with the data as the first parameter in each function). You could save your script containing all your functions as `my_functions.R`

You can then write a new “master” script which contains the following:

```
source(“my_functions.R”)

read_in_data(location) %>%
    tidy_my_data() %>%
    visualise_my_data() %>%
    model_my_data() %>%
    follow_up_tests()
```

Purrr for functional programming

The purrr packages contains functions (such as `map_df()` and `map())` which allow you to apply the same function over (e.g.) columns in a tibble, lists etc. For example, I create a function to square the values of a variable - and with `map_df()` I map this function over all columns in a tibble that are numeric:

```
square_the_values <- function(x) {
  x <- x * x
}

starwars %>%
  select_if(is.numeric) %>%
  map_df(square_the_values)
```

Purrr for functional programming

```
# A tibble: 87 x 3
   height  mass birth_year
    <int> <dbl>      <dbl>
 1  29584  5929       361
 2  27889  5625     12544
 3   9216  1024      1089
 4  40804 18496      1756.
 5  22500  2401       361
 6  31684 14400      2704
 7  27225  5625      2209
 8   9409  1024        NA
 9  33489  7056       576
10  33124  5929      3249
# … with 77 more rows
```

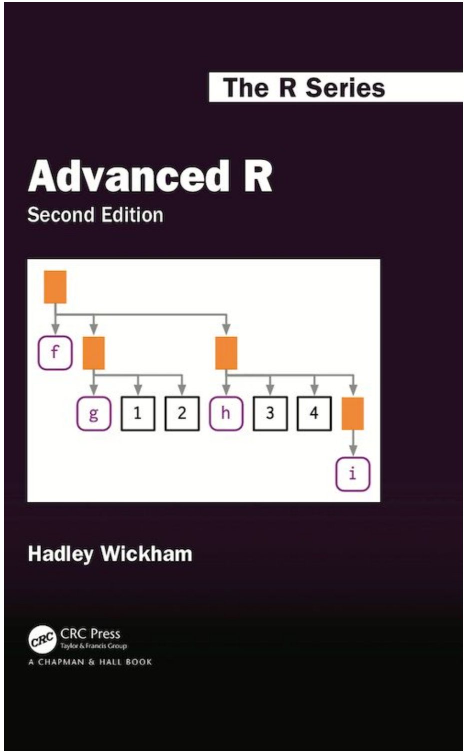
Writing anonymous functions

For simple functions, it can be easier to include the body of the function in the map() call without first assigning it to a function name. You can use the ~ operator to do this as follows:

```
starwars %>%
  select_if(is.numeric) %>%
  map_df(~(.x <- .x * .x))
```

which will achieve exactly the same result as we produced when the function was named.

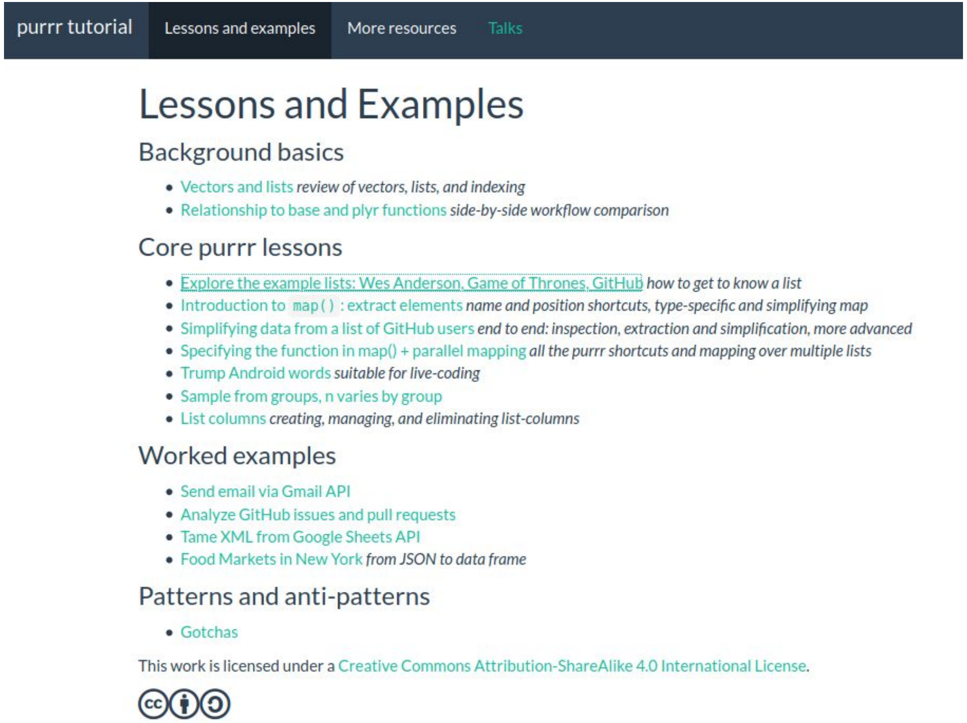
More on writing functions…



https://adv-r.hadley.nz/

More on purrr and functional programming...

https://jennybc.github.io/purrr-tutorial/



Lessons and Examples

Background basics

- Vectors and lists *review of vectors, lists, and indexing*
- Relationship to base and plyr functions *side-by-side workflow comparison*

Core purrr lessons

- Explore the example lists: Wes Anderson, Game of Thrones, GitHub *how to get to know a list*
- Introduction to `map()`: extract elements *name and position shortcuts, type-specific and simplifying map*
- Simplifying data from a list of GitHub users *end to end: inspection, extraction and simplification, more advanced*
- Specifying the function in map() + parallel mapping *all the purrr shortcuts and mapping over multiple lists*
- Trump Android words *suitable for live-coding*
- Sample from groups, n varies by group
- List columns *creating, managing, and eliminating list-columns*

Worked examples

- Send email via Gmail API
- Analyze GitHub issues and pull requests
- Tame XML from Google Sheets API
- Food Markets in New York *from JSON to data frame*

Patterns and anti-patterns

- Gotchas