

Probability - Tidyverse 2¹


Reducing duplication - writing functions and introduction to purrr

Introduction to Quantitative Social Science

Xiaolong Yang

University of Tokyo

June 28, 2022

¹Sources: R for Data Science; Advanced R by Hadley Wickham 

Today's Game Plan

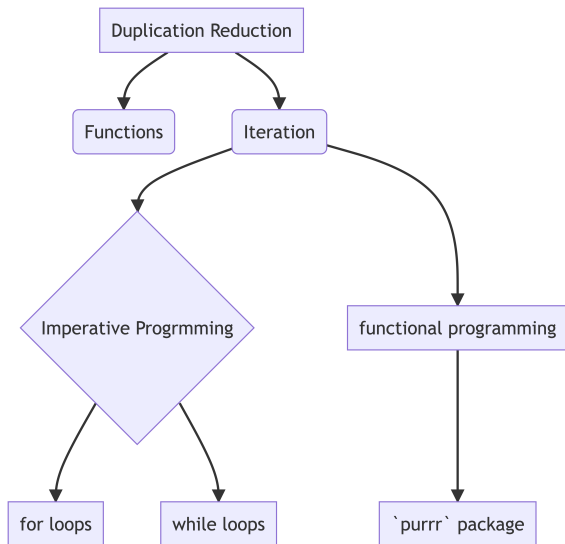
- 1 reducing duplication: **functions**
- 2 purrr package
 - purrr: `map_df()` (introduced in Chapter 6: Probability (sections 6.3-6.4))

i Today's in-class assignment: `intrade-prob`

Section 1

Functions

Landscape of Duplication Reduction in R



Why writing functions?

What does this code chunk do?

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /  
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$b <- (df$b - min(df$b, na.rm = TRUE)) /  
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$c <- (df$c - min(df$c, na.rm = TRUE)) /  
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))  
df$d <- (df$d - min(df$d, na.rm = TRUE)) /  
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Why writing functions?

- re-scaling each column to have a range from 0 - 1

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /  
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$b <- (df$b - min(df$b, na.rm = TRUE)) /  
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$c <- (df$c - min(df$c, na.rm = TRUE)) /  
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))  
df$d <- (df$d - min(df$d, na.rm = TRUE)) /  
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Why writing functions?

- re-scaling each column to have a range from 0 - 1
- **But there is a mistake!!**

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /  
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$b <- (df$b - min(df$b, na.rm = TRUE)) /  
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$c <- (df$c - min(df$c, na.rm = TRUE)) /  
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))  
df$d <- (df$d - min(df$d, na.rm = TRUE)) /  
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Why writing functions?

- re-scaling each column to have a range from 0 - 1
- **But there is a mistake!!**
 - `df$b`: **did not change a to b**

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /  
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$b <- (df$b - min(df$b, na.rm = TRUE)) /  
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))  
df$c <- (df$c - min(df$c, na.rm = TRUE)) /  
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))  
df$d <- (df$d - min(df$d, na.rm = TRUE)) /  
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```


What if we have a function?

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Advantages of functions over copy-paste

- 1 easier to see the intent of your code: eyes on **difference** not **similarity**
- 2 easier to respond to changes in requirements
- 3 fewer bugs (i.e. updating a variable name in one place, but not in another).

You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code).

3 key steps to create a function

- 1 pick a name for the function

```
square
```

3 key steps to create a function

- 1 pick a name for the function
- 2 list the inputs, or arguments, to the function inside function

```
square <- function(x) {}
```

3 key steps to create a function

- 1 pick a name for the function
- 2 list the inputs, or arguments, to the function inside function
- 3 place the code you have developed in body of the function

It's easier to start with working code and turn it into a function; it's harder to create a function and then try to make it work

```
square <- function(x) {  
  x^2  
}
```

```
square(13)
```

```
[1] 169
```

Function arguments

- data arguments: supplies the data to compute on
- details arguments: supplies arguments that control the details of the computation
- `lm()` as an example

💡 `View()` allows you to check the source code of a function

```
function (formula, data, subset, weights, na.action, method =  
  model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok =  
  contrasts = NULL, offset, ...)  
{ ...  
  ...  
  ...  
}
```

Conditional execution

- an if statement allows you to conditionally execute code

```
if (condition) {  
  # code executed when condition is TRUE  
} else {  
  # code executed when condition is FALSE  
}
```

Multiple conditions

- chain multiple if statements together

```
if (condition 1) {  
  # do this if TRUE  
} else if (condition 2) {  
  # do that if TRUE  
} else {  
  # do something else if FALSE  
}
```


Good practices

Functions are for humans and computers

- `if` and `function` should always be followed by `{}`
 - `{`: never go on its own line
 - `}`: always go on its own line (unless followed by `else`)

```
# Good
if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

```
# Bad
if (y == 0) {
  log(x)
} # with else in this case
else {
  y ^ x
}
```

Functions are for humans and computers

- function names: verbs
- argument names: nouns
- use inline code to explain the “why”
 - avoid the “what” or “how”

```
# Good  
input_select()  
input_checkbox()
```

```
# Bad  
f()  
my_awesome_function()
```

Section 2

Brief introduction to purrr



Figure 1: purrr

- R as a functional programming (FP) language
- purrr provides complete and consistent tools for working with **functions** and different data types (**vectors**) → enhances R's efficiency
 - the family of `map()` function → replace many for loops with succinct code

i [purrr cheatsheet](#)

purrr package: `map_df()`

- transforms the input by applying a function to each variable of a data frame or tibble (**each element of a list or atomic vector**)
- returns a data frame/tibble
- arguments
 - `.x` = a data frame/tibble (list or atomic vector)
 - `.f` = a function

```
map_df(.x, .f)
```

purrr package: map_df()

Example 1

```
map_df(FLVoters, class)
```

```
# A tibble: 1 x 6
```

surname	county	VTD	age	gender	race
<chr>	<chr>	<chr>	<chr>	<chr>	<chr>

```
1 character integer integer integer character character
```

```
class(FLVoters)
```

```
[1] "data.frame"
```

purrr package: map_df()

Example 2

```
FLVoters %>%
```

```
  map(unique) %>%
```

```
  map_df(length)
```

```
# A tibble: 1 x 6
```

```
  surname county   VTD   age gender  race
  <int>   <int> <int> <int> <int> <int>
1    6240     67  1053   82     3     7
```

Summary

What we learnt

- how to write functions
 - how to communicate functions by following good practices
- `purrr (map_df())`

Future Game Plan

- data types in R: vector
- reducing duplication: **iteration**
- new functions in **Chapter 7: Uncertainty**