STAT 209
Making Repetitive Tasks Modular

July 15, 2021

Colin Reimer Dawson

The Plan

- Intro to Functions
- Writing Functions in R
- Intro to Loops
- Compact alternatives to loops in R

Outline

Repeated Similar Code

Ever find yourself doing this sort of thing?

```
myData %>%
    ggplot(aes(x = variable1, y = variable2)) +
    geom_point() + geom_smooth() +
    xlab("Label for Variable 1") + ylab("Label for Variable 2")
myData %>%
    ggplot(aes(x = variable3, y = variable4)) +
    geom_point() + geom_smooth() +
    xlab("Label for Variable 3") + ylab("Label for Variable 4")
myOtherData %>%
    ggplot(aes(x = variable1, y = variable2)) +
    geom_point() + geom_smooth() +
    xlab("Label for Variable 1") + ylab("Label for Variable 2")
```

Repeated Similar Code

Ever find yourself doing this sort of thing?

```
myData %>%
    ggplot(aes(x = variable1, y = variable2)) +
    geom_point() + geom_smooth() +
    xlab("Label for Variable 1") + ylab("Label for Variable 2")
myData %>%
    ggplot(aes(x = variable3, y = variable4)) +
    geom_point() + geom_smooth() +
    xlab("Label for Variable 3") + ylab("Label for Variable 4")
myOtherData %>%
    ggplot(aes(x = variable1, y = variable2)) +
    geom_point() + geom_smooth() +
    xlab("Label for Variable 1") + ylab("Label for Variable 2")
```

There are only a few things changing here:

1. the dataset
2. the variable names
3. the axis labels

A Plot "Template"

Inputs:

1. Dataset name (`dataset`)
2. "x" variable name (`xvar`)
3. "y" variable name (`yvar`)
4. "x" variable axis label (`xlabel`)
5. "y" variable axis label (`ylabel`)

Procedure:

1. Make a `ggplot` with `dataset`
2. Define an aesthetic: `aes(x = xvar, y = yvar)`
3. Add `geom_point()` and `geom_smooth()`
4. Add labels via `xlab(xlabel)` and `ylab(ylabel)`

R Syntax for "Template" (Function) Definition

```
my_function_name <- function( # the word function is literal
    input1,                   # the argument names should describe their role
    input2,
    input3 = "some default value",
    input4 = "another default value")
{
    ...                       # the dots just represent "some code"
    someResult <- someExpression
    return(someResult)       # return() needs parens
}
```

- In this function, `input1` and `input2` are **required**, `input3` and `input4` are **optional** (will use **default values** if omitted)
- The **return value** is typically an object defined within the function body (here, it's `some_result`)

Using the Function

```
## Here's that definition again, for reference
my_function_name <- function(
  input1, input2, input3 = "default3", input4 = "default4") {...}
```

```
## (a) Set input1 = x, input2 = y
##     leaving input3 = "default3", input4 = "default4"
my_function_name(x, y)
## (b) Or override defaults for input3 and input4
my_function_name(x, y, "override3", "override4")
## (c) Can give arguments in any order if we give the names
##     (input4 gets its default value)
my_function_name(input3 = "overrride3", input1 = x, input2 = y)
## (d) Or a combination of names and positions
##     (input3 gets the default value)
my_function_name(x, input2 = y, input4 = "override4")
## (e) We can even use it with a pipe (same as (d):
##     input1 = x, input2 = y, input3 = "default3"
##     input4 = "overrideValue")
x %>% my_function_name(y, input4 = "override4")
```

Concrete Example

Demo in RStudio: “Writing Functions Demo”

Global Variables

For "quick-and-dirty" special-purpose functions, we may sometimes refer to specific datasets or variable names inside a function, rather than passing them as arguments:

```
n_richest_countries <- function(n = 6)
{
    AllCountries %>%
        select(Country, Population, GDP) %>%
        arrange(desc(GDP)) %>%
        head(n = n)
}
```

We can define the function without having loaded the data, but...

Global Variables

For "quick-and-dirty" special-purpose functions, we may sometimes refer to specific datasets or variable names inside a function, rather than passing them as arguments:

```
n_richest_countries <- function(n = 6)
{
    AllCountries %>%
        select(Country, Population, GDP) %>%
        arrange(desc(GDP)) %>%
        head(n = n)
}
```

We can define the function without having loaded the data, but...

```
n_richest_countries(n = 3)
```

```
    Error in select(., Country, Population, GDP): object 'AllCountries'
not found
```

We'll get an error if we try to call it

Global Variables

It works if we load the data first and then call it...

```
data(AllCountries, package = "Lock5Data")
n_richest_countries(n = 3)
```

```
          Country Population    GDP
     1 Luxembourg      0.543 110665
     2     Norway      5.080 100898
     3      Qatar      2.169  93714
```

Global Variables

It works if we load the data first and then call it...

```
data(AllCountries, package = "Lock5Data")
n_richest_countries(n = 3)
```

```
          Country Population    GDP
     1 Luxembourg      0.543 110665
     2     Norway      5.080 100898
     3      Qatar      2.169  93714
```

But it's not great coding practice to write functions that depend on external variables unless we're only using them in a single context

Passing Variable Names to `tidyverse` functions

We unfortunately can't (directly) pass variable names to most `tidyverse` functions via arguments to custom functions, since they interpret variable names in a special way

```
scatterplot_with_smooth <- function(dataset, xvar, yvar)
{
    dataset %>% ggplot(aes(x = xvar, y = yvar)) +
        geom_point() + geom_smooth()
}
```

The definition runs fine...

Passing Variable Names to `tidyverse` functions

We unfortunately can't (directly) pass variable names to most `tidyverse` functions via arguments to custom functions, since they interpret variable names in a special way

```
scatterplot_with_smooth <- function(dataset, xvar, yvar)
{
    dataset %>% ggplot(aes(x = xvar, y = yvar)) +
        geom_point() + geom_smooth()
}
```

The definition runs fine...

```
AllCountries %>%
    scatterplot_with_smooth(
        xvar = BirthRate,
        yvar = LifeExpectancy)

    Error in FUN(X[[i]], ...):  object 'BirthRate' not found
```

Passing Variable Names to `tidyverse` functions

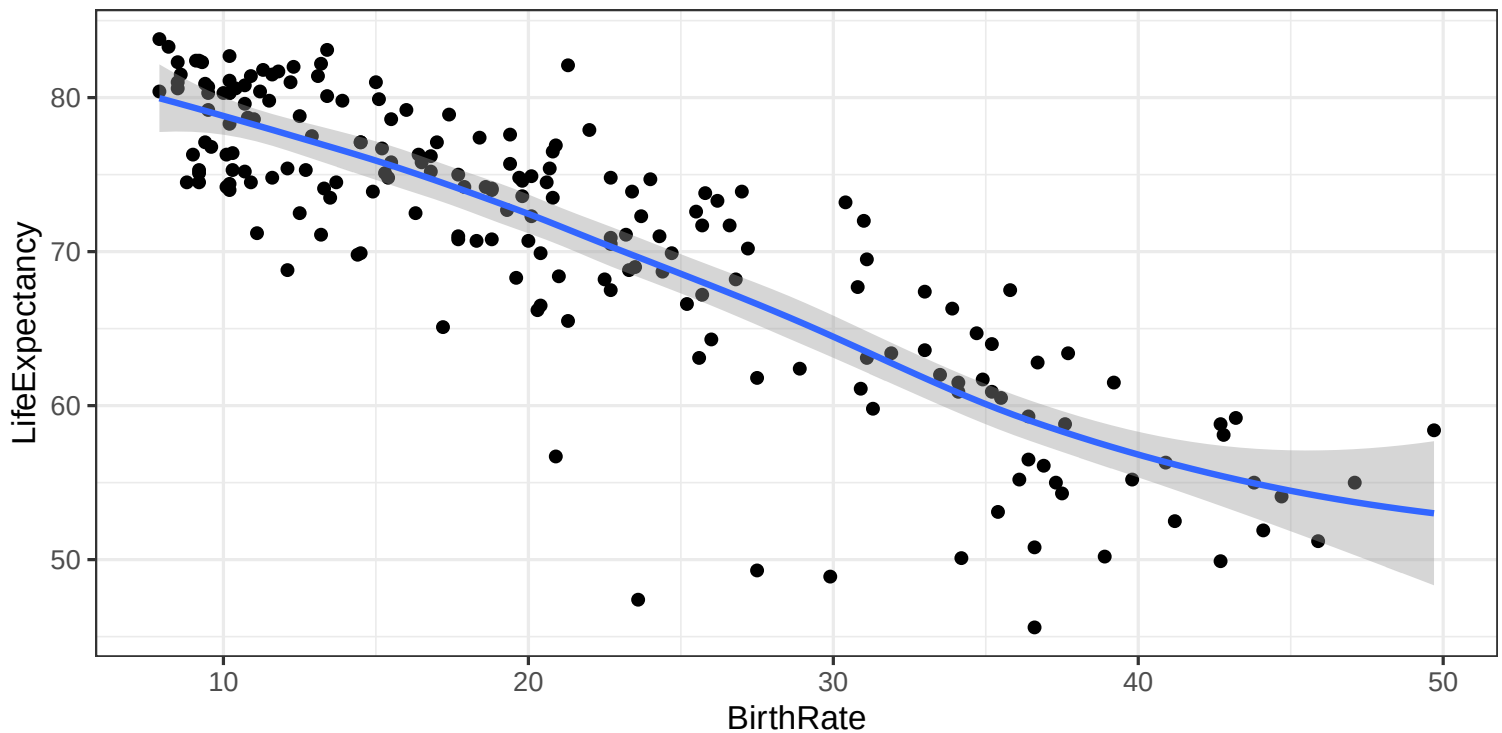
There is a special syntax to convert strings into variable names as the `tidyverse` uses them.

```
scatterplot_with_smooth <- function(dataset, xvar, yvar)
{
    xvar_sym <- ensym(xvar) # the variable names need to be converted
    yvar_sym <- ensym(yvar) # to 'symbols' internally with ensym()
    dataset %>%
        ## and then 'dereferenced' in tidyverse funs
        ## by preceding the symbol with !!
        ggplot(aes(x = !!xvar_sym, y = !!yvar_sym)) +
        geom_point() +
        geom_smooth()
}
```

Passing Variable Names to `tidyverse` functions

Now we can pass variable names like we would normally.

```
AllCountries %>%
    scatterplot_with_smooth(
        xvar = BirthRate,
        yvar = LifeExpectancy)
```

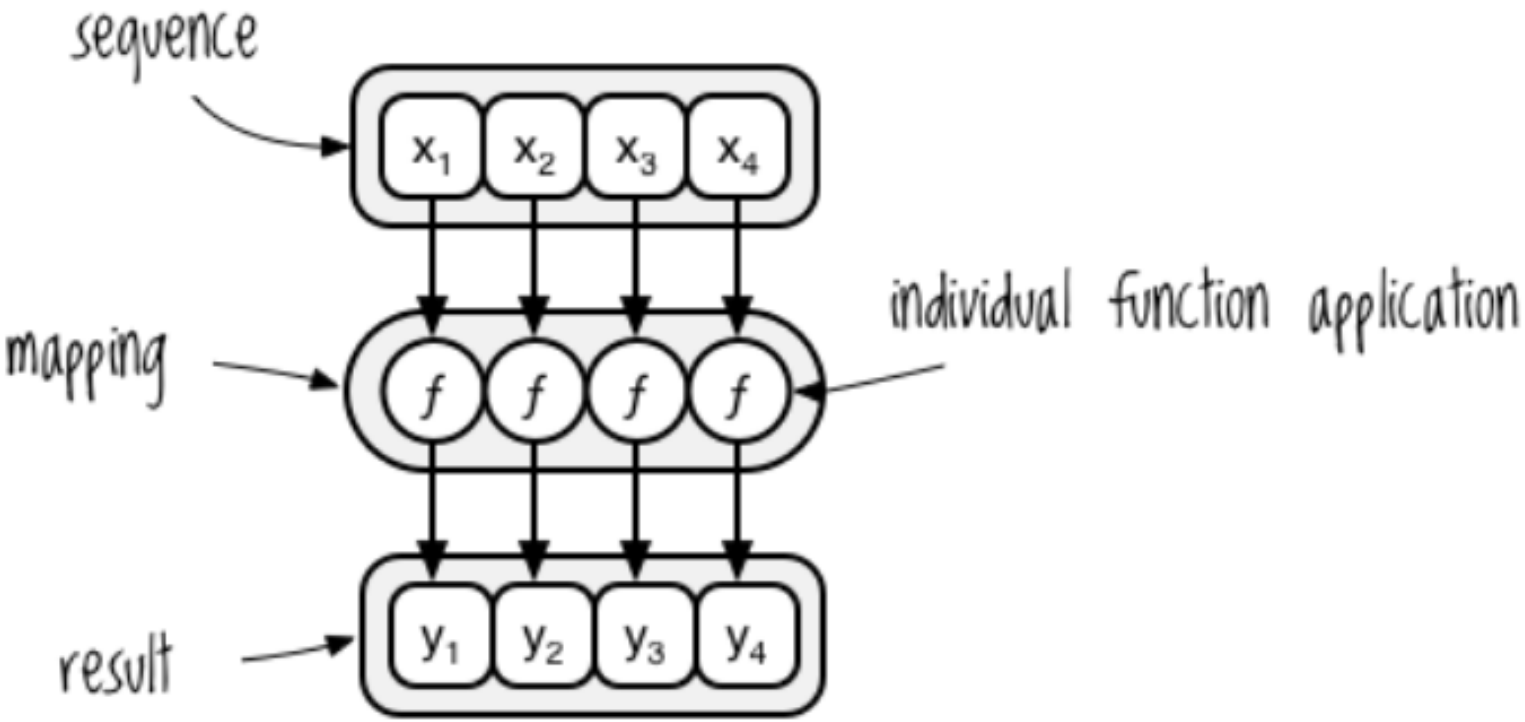


Outline

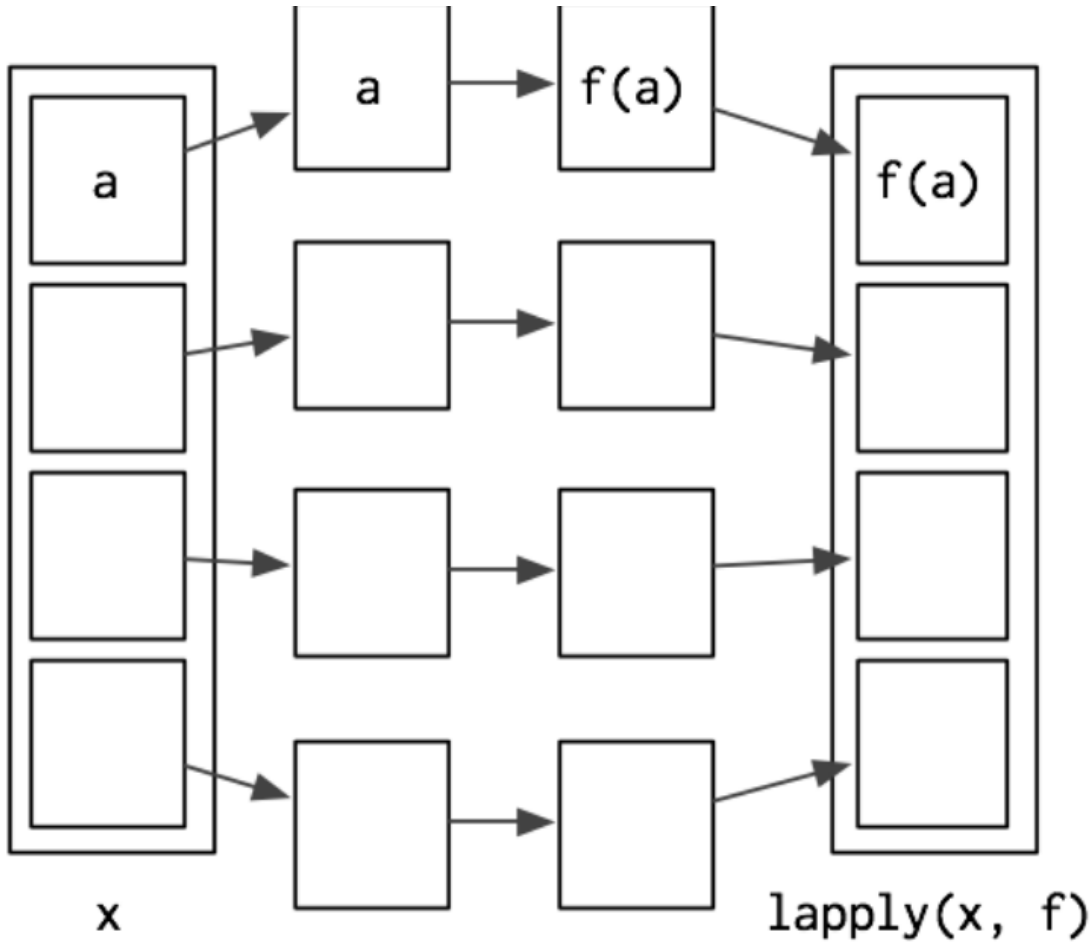
Iteration

- Sometimes you have
  - a function `yourFunction(someValue)` that returns something
  - a list of `someValues` you want to apply `yourFunction` to
- and you want to produce a list or table that has all the results collected in one place
- Could just copy and paste, but
  - the list might be long
  - this is a nightmare to make changes to
  - consolidating the results would be cumbersome

Mapping


sequence
x_1 x_2 x_3 x_4
mapping
f f f f
individual function application
result
y_1 y_2 y_3 y_4

The `lapply()` function



Working with lists

- `lapply()` returns a **list**
- Items in a list can be of **any types** (even all different)
- We can use `unlist()` to convert a list into a **vector**, provided the elements are the **same type**
- The `bind_rows()` function **flattens** a list of **data frames** (or **tibbles**) into one big data frame...
    - as long as they all have the same variables (columns)

Example: `lapply()` to get group means

```
mean_of_subset <- function(dataset, group_var, group_value, result_var) {
    group_var_sym <- ensym(group_var)
    result_var_sym <- ensym(result_var)
    dataset %>%
        filter((!!group_var_sym) == group_value) %>%
        summarize(
            group      = group_value,
            group_mean = mean(!!result_var_sym, na.rm = TRUE))
}
```

Example: `lapply()` to get group means

```
cars <- c("corolla", "civic", "camry")
lapply(
    X           = cars,
    FUN         = mean_of_subset,
    dataset     = mpg,
    group_var   = model,
    result_var  = hwy) %>%
  bind_rows()
```

```
# A tibble: 3 x 2
  group   group_mean
  <chr>        <dbl>
1 corolla       34
2 civic         32.6
3 camry         28.3
```

The `do()` function

- Similar to `summarize()` in that it works over a collection of data frames
- Useful with `group_by()`

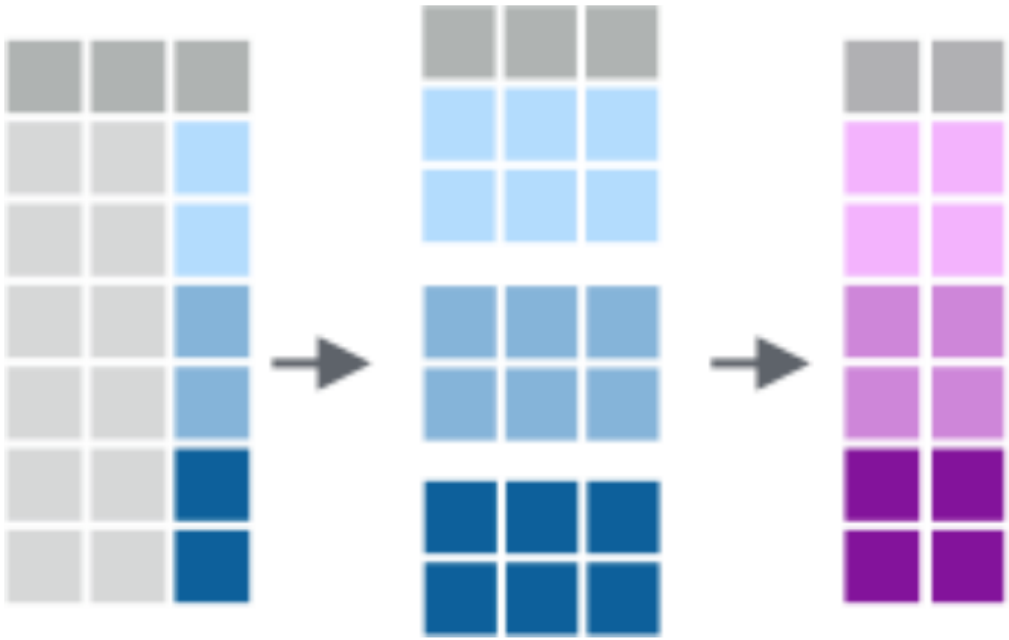
`group_by()` + `summarize()`

Group Cases

Use **group_by()** to created a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

group_by() + do()



```
mtcars %>%
  group_by(cyl) %>%
  do(func(.))
```

Demo in RStudio: “Writing Functions Demo”