Week 11 - Functions and Iteration

Alex Lishinski

October 25, 2021

Welcome!

Welcome to *week 11*!

Record the meeting

A recap of last week (on modeling)

- A model is a simplification (and a summary) of your way
- A common type of model is a regression model, aka a linear regression model (or a linear model)

This code represents the regression of `hp` upon `mpg`:

```
lm(mpg ~ hp, data = my_data)
```

This code often corresponds to the underlying mathematical/statistical equation:

$$\text{mpg} = \alpha + \beta_1(\text{hp}) + \epsilon$$

A recap of last week (on modeling)

- A linear model can help you learn how one or more ***independent variables*** (or, ***x*** variables) relate to one ***dependent variable*** (a ***y*** variable)
- It is common to specify a number of models, and to compare them (commonly based on how much of the variation in the dependent variable is accounted for by relationships with the ***x*** variable(s))
- the `lm()` function is a great tool for specifying relatively simple linear regression models, but it can be extended in a number of powerful ways
- R has a number of tools for interpreting and presenting the output of models

Checking-in on final projects

- We will hear more details about your plans on Thursday
- And, generally, please consider a) the scope of your project and b) the level of detail in your plans
- I want you to do a small(er) number of things well

Topics for today

Record the meeting

A. Functions
B. Iteration (or, *applying* functions)
C. More shiny - Reactivity

A. Functions

A function is a collection of code that:

- takes one or more inputs (most commonly in R, data!)
- and produces one or more forms of output (often, your data---transformed!)

You *already* use functions all of the time:

```
mpg
```

```
## # A tibble: 234 × 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto… f        18    29 p     comp…
##  2 audi         a4           1.8  1999     4 manu… f        21    29 p     comp…
##  3 audi         a4           2    2008     4 manu… f        20    31 p     comp…
##  4 audi         a4           2    2008     4 auto… f        21    30 p     comp…
##  5 audi         a4           2.8  1999     6 auto… f        16    26 p     comp…
##  6 audi         a4           2.8  1999     6 manu… f        18    26 p     comp…
##  7 audi         a4           3.1  2008     6 auto… f        18    27 p     comp…
##  8 audi         a4 quattro   1.8  1999     4 manu… 4        18    26 p     comp…
##  9 audi         a4 quattro   1.8  1999     4 auto… 4        16    25 p     comp…
## 10 audi         a4 quattro   2    2008     4 manu… 4        20    28 p     comp…
## # … with 224 more rows
```

A. Functions

A function is a collection of code that:

- takes one or more inputs (most commonly in R, data!)
- and produces one or more forms of output (often, your data---transformed!)

You *already* use functions all of the time:

```
print(mpg)
```

```
## # A tibble: 234 × 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto… f        18    29 p     comp…
##  2 audi         a4           1.8  1999     4 manu… f        21    29 p     comp…
##  3 audi         a4           2    2008     4 manu… f        20    31 p     comp…
##  4 audi         a4           2    2008     4 auto… f        21    30 p     comp…
##  5 audi         a4           2.8  1999     6 auto… f        16    26 p     comp…
##  6 audi         a4           2.8  1999     6 manu… f        18    26 p     comp…
##  7 audi         a4           3.1  2008     6 auto… f        18    27 p     comp…
##  8 audi         a4 quattro   1.8  1999     4 manu… 4        18    26 p     comp…
##  9 audi         a4 quattro   1.8  1999     4 auto… 4        16    25 p     comp…
## 10 audi         a4 quattro   2    2008     4 manu… 4        20    28 p     comp…
## # … with 224 more rows
```

A. Functions

`select()` is a function

```
mpg %>%
  select(model, displ, cyl)
```

```
## # A tibble: 234 × 3
##    model      displ   cyl
##    <chr>      <dbl> <int>
##  1 a4           1.8     4
##  2 a4           1.8     4
##  3 a4           2       4
##  4 a4           2       4
##  5 a4           2.8     6
##  6 a4           2.8     6
##  7 a4           3.1     6
##  8 a4 quattro   1.8     4
##  9 a4 quattro   1.8     4
## 10 a4 quattro   2       4
## # … with 224 more rows
```

A. Functions

`select()` and `filter()` are functions

```
mpg %>%
  select(model, displ, cyl) %>%
  filter(cyl == 6)
```

```
## # A tibble: 79 × 3
##    model      displ   cyl
##    <chr>      <dbl> <int>
##  1 a4           2.8     6
##  2 a4           2.8     6
##  3 a4           3.1     6
##  4 a4 quattro   2.8     6
##  5 a4 quattro   2.8     6
##  6 a4 quattro   3.1     6
##  7 a4 quattro   3.1     6
##  8 a6 quattro   2.8     6
##  9 a6 quattro   3.1     6
## 10 malibu       3.1     6
## # … with 69 more rows
```

A. Functions

There are all kinds of functions:

```
mpg %>%
  summarize(mean_cty = mean(cty),
            mean_hwy = mean(hwy))
```

```
## # A tibble: 1 × 2
##   mean_cty mean_hwy
##      <dbl>    <dbl>
## 1     16.9     23.4
```

```
ten_mpg_values <- c(13, 63, 23, 43, 23, 33, 42, 34, 21, 23)
mean(ten_mpg_values)
```

```
## [1] 31.8
```

A. Functions

You can write your own functions!

Suppose we want to write a function to *standardize* the `cty` variable to have $M = 0$ and $SD = 1$; presently, its M and SD are:

```
mpg %>%
  summarize(mean_cty = mean(cty),
            sd_cty = sd(cty))
```

```
## # A tibble: 1 × 2
##   mean_cty sd_cty
##      <dbl>  <dbl>
## 1     16.9   4.26
```

A. Functions

(In this case, there *is* an existing function, but it has some quirks, including, importantly, returning a *matrix*, rather than a vector/"a column")

So, let's write our own. Here is a template, with two "blanks" represented by "_":

```
standardize_a_variable <- function(___) {
  ___
}
```

A. Functions

A not very useful function!

```
standardize_variable <- function(variable) {
  variable
}
```

```
mpg %>%
  mutate(cty_std = standardize_variable(cty)) %>%
  summarize(mean_cty_std = mean(cty_std),
            sd_cty_std = sd(cty_std))
```

```
## # A tibble: 1 × 2
##   mean_cty_std sd_cty_std
##          <dbl>      <dbl>
## 1         16.9       4.26
```

A. Functions

A useful `standardize_variable`

```
standardize_variable <- function(variable) {
  variable <- variable - mean(variable) # transforms variable to have M = 0
  variable <- variable / sd(variable) # transforms variable to have SD = 1
  variable
}
```

```
mpg %>%
  mutate(cty_std = standardize_variable(cty)) %>%
  summarize(mean_cty_std = mean(cty_std),
            sd_cty_std = sd(cty_std))
```

```
## # A tibble: 1 × 2
##   mean_cty_std sd_cty_std
##          <dbl>      <dbl>
## 1     2.98e-16          1
```

A. Functions

Let's double-check that the mean is practically equal to 0 (by rounding to precision to three decimal points)

```
mpg %>%
  mutate(cty_std = standardize_variable(cty)) %>%
  summarize(mean_cty_std = mean(cty_std),
            sd_cty_std = sd(cty_std)) %>%
  mutate(mean_cty_std_rounded = round(mean_cty_std, 3))
```

```
## # A tibble: 1 × 3
##   mean_cty_std sd_cty_std mean_cty_std_rounded
##          <dbl>      <dbl>                <dbl>
## 1     2.98e-16          1                    0
```

A. Functions

We can add an argument to a function to modify how it works.

```
standardize_variable <- function(variable, remove_na_values = FALSE) {
  variable <- variable - mean(variable, na.rm = remove_na_values)
  variable <- variable / sd(variable, na.rm = remove_na_values)
  variable
}
```

A. Functions

What if we wanted to scale the variables cty, hwy, and cyl? Imagine that cyl has a missing (NA) value.

```
mpg %>%
  mutate(cty_std = standardize_variable(cty),
         hwy_std = standardize_variable(hwy),
         cyl_std = standardize_variable(cyl, remove_na_values = TRUE)) %>%
  select(cty_std, hwy_std, cyl_std, everything()) # bring the new variables to the beginning of the d
```

```
## # A tibble: 234 × 14
##    cty_std hwy_std cyl_std manufacturer model      displ  year   cyl trans drv
##      <dbl>   <dbl>   <dbl> <chr>        <chr>      <dbl> <int> <int> <chr> <chr>
##  1   0.268   0.934 -1.17   audi         a4           1.8  1999     4 auto… f
##  2   0.973   0.934 -1.17   audi         a4           1.8  1999     4 manu… f
##  3   0.738   1.27  -1.17   audi         a4           2    2008     4 manu… f
##  4   0.973   1.10  -1.17   audi         a4           2    2008     4 auto… f
##  5  -0.202   0.430  0.0689 audi         a4           2.8  1999     6 auto… f
##  6   0.268   0.430  0.0689 audi         a4           2.8  1999     6 manu… f
##  7   0.268   0.598  0.0689 audi         a4           3.1  2008     6 auto… f
##  8   0.268   0.430 -1.17   audi         a4 quattro   1.8  1999     4 manu… 4
##  9  -0.202   0.262 -1.17   audi         a4 quattro   1.8  1999     4 auto… 4
## 10   0.738   0.766 -1.17   audi         a4 quattro   2    2008     4 manu… 4
## # … with 224 more rows, and 4 more variables: cty <int>, hwy <int>, fl <chr>,
## #   class <chr>
```

A. Functions

In summary:

- You already use functions all of the time
- You can write your own functions that take one or more types of input and return output
- The functions that you are using within R were written by other people!
- https://github.com/tidyverse/dplyr/blob/master/R/select.R
- https://github.com/SurajGupta/r-source/blob/master/src/library/stats/R/median.R
- https://github.com/jrosen48/konfound
- https://github.com/data-edu/tidyLPA
- If you find yourself copying and pasting the same code, it may be worthwhile to encapsulate your code within a function (more on this in a moment!)

B. Iteration

Iteration is another name for your computer carrying out some step(s) multiple times.

Iteration is helpful when, even after writing a function, you find yourself copying and pasting the same code (with modifications)

B. Iteration

Another way of writing the above code in which we are scaling multiple variables:

```
mpg %>%
  mutate_at(vars(cty, hwy, cyl), standardize_variable) %>%
  select(cty, hwy, cyl, everything()) # bring the transformed variables to the beginning of the data
```

```
## # A tibble: 234 × 11
##       cty   hwy     cyl manufacturer model  displ  year trans  drv   fl    class
##     <dbl> <dbl>   <dbl> <chr>        <chr>  <dbl> <int> <chr>  <chr> <chr> <chr>
##  1  0.268 0.934 -1.17   audi         a4       1.8  1999 auto(… f     p     comp…
##  2  0.973 0.934 -1.17   audi         a4       1.8  1999 manua… f     p     comp…
##  3  0.738 1.27  -1.17   audi         a4       2    2008 manua… f     p     comp…
##  4  0.973 1.10  -1.17   audi         a4       2    2008 auto(… f     p     comp…
##  5 -0.202 0.430  0.0689 audi         a4       2.8  1999 auto(… f     p     comp…
##  6  0.268 0.430  0.0689 audi         a4       2.8  1999 manua… f     p     comp…
##  7  0.268 0.598  0.0689 audi         a4       3.1  2008 auto(… f     p     comp…
##  8  0.268 0.430 -1.17   audi         a4 qu…   1.8  1999 manua… 4     p     comp…
##  9 -0.202 0.262 -1.17   audi         a4 qu…   1.8  1999 auto(… 4     p     comp…
## 10  0.738 0.766 -1.17   audi         a4 qu…   2    2008 manua… 4     p     comp…
## # … with 224 more rows
```

B. Iteration

Iteration can be helpful when you want to apply a function multiple times.

For example, I recently needed to download 14 surveys for a teacher professional development-focused research project, https://megabitess.org!

There is a great package, qualtRics, which can help with this, but it's tedious to have to download the surveys one-by-one.

The map functions can help with this.

B. Iteration

The `purrr` package is a package that is part of the tidyverse and lets you do various iteration operations

The package consists of a family of related functions based on `map`

```
library(qualtRics)
library(tidyverse)

# qualtRics::all_surveys()

description <- c("effectiveness - communication",
                 "effectiveness - summer",
                 "end-of-day survey",
                 "post-summer survey",
                 "pre-summer survey",
                 "end-of-day survey",
                 "end-of-day survey",
                 "end-of-day survey",
                 "end-of-day survey",
                 "post-summer survey",
                 "effectiveness - summer",
                 "effectiveness - GIS",
                 "post-GIS survey",
                 "pre-GIS survey"
)

survey_id <- c("SV_3IP9R9vFxZ7qGUe",
               "SV_bJEe2AWjZdH4dcp",
               "SV_81baWN1LtwNzV9r",
               "SV_5jue08g3DvyGlmd",
               "SV_6s9qu47PyM0js8t",
               "SV_5iNHeOZns6zzxat",
               "SV_9uhuHORliSFZq6x",
               "SV_d0EQx7RXOZxasT3",
               "SV_a46oo0Kklrlnh0F",
               "SV_e2PwPb0wmRAWCcl",
               "SV_9Zz8bjbBUPw1K4t",
               "SV_cFRH5pgKwnYJRLT",
               "SV_8HVSt4wfW3Hj2pn",
```

B. Iteration

```
my_surveys <- tibble(description = description,
                     survey_id = survey_id)

all_surveys <- my_surveys$survey_id %>% map(fetch_survey)

my_surveys_end_of_day <- my_surveys %>%
  filter(description == "end-of-day survey")

all_end_of_day_surveys <- my_surveys_end_of_day$survey_id %>% map_df(fetch_survey)
```

C. Shiny continued: Reactivity

Shiny apps work on the principle of Reactivity:

- Functions in your server code need to run again when there are changes to the input
- UI is the same for all users of your app, server changes for each user
- The input object can only be modified in the UI, and can only be accessed in a reactive context
- We don't want to tell the app when to re-run code
- Shiny figures out when to run the code, you provide recipes for what to do if that happens
- When code is run depends on the connections between reactive dependencies, not the order written

C. Shiny continued: Reactivity

Shiny apps work on the principle of Reactivity:

- Inputs from the UI connect to outputs in the server
- Inputs are modified based on user input, server code then runs reactively to change output accordingly
- Reactive expressions are a middle ground between inputs and outputs
- Reactive expressions change based on input, but they can be used in constructing outputs
- This can allow you to avoid repeating yourself
- This is necessary because using a reactive context lets your code respond to changing input values

C. Shiny continued: Controlling when code is run

Sometimes you don't want your code to run right away

A tool that shiny provides to allow you to control this behavior is an action button

An action button is a UI widget, but by using it in conjunction with the function `eventReactive` code running is dependent on the button, rather than reactive response to changing inputs

Observers allow you to make other reactive side effects dependent on actions e.g. `observeEvent`

Note: the most up to date version of Shiny recommends the use of a new function `bindEvent`

Logistics

This week

- Homework 12: Available Thursday; **Due by Tuesday, 11/2**
- Final Project: More detailed plan present to class: Thursday (~ 3-5 minutes, ~ 2-3 slides)

Schedule

- The product for your final project will be due are both due by the end of the day on Tuesday December 7
- We will do presentations of your final project in class on our last day - Tuesday November 30

Wrapping up

On Slack:

- What is one thing you learned today?
- What is something you want to learn more about?
- Share your feelings in GIF form!