Lecture 9: Writing functions

EDUC 263: Managing and Manipulating Data Using R

Ozan Jaquette

1 Introduction

Libraries

```
library(tidyverse)
#> -- Attaching packages --------------------------------------------------- tidyv
#> v ggplot2 3.0.0     v purrr   0.2.5
#> v tibble  1.4.2     v dplyr   0.7.6
#> v tidyr   0.8.1     v stringr 1.3.1
#> v readr   1.1.1     v forcats 0.3.0
#> -- Conflicts ---------------------------------------------------- tidyverse_c
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(haven)
```

Data we will work with

```r
#load dataset with one obs per recruiting event
load("../../data/recruiting/recruit_event_somevars.Rdata")

#load dataset with one obs per high school
load("../../data/recruiting/recruit_school_somevars.Rdata")
```

2 Why and when to write a function

What are functions

Functions are pre-written bits of code that accomplish some task. Functions generally follow three sequential steps:

1. take in an **input** object(s)
2. **process** the input.
3. **return** (A) a new object or (B) a visualizatoin (e.g., plot)

We've been working with functions all quarter. For example, the `select()` function (type `?select` in console):

1. **input**. takes in a data frame object as the input
2. **processing**. keeps selected variables that you specify
3. **return**. Returns a new object, which may be a vector, a data-frame, a plot, etc.

```
#?select
select(df_event,event_type,event_state,zip) %>% str()
#df_event %>% select(event_type,event_state,zip) %>% str() # same result
```

The `sum()` function (`?sum` in console):

1. **input**. takes in a vector of elements (numeric or logical)
2. **processing**. Calculates the sum of elements
3. **return**. Returns a numeric vector of length=1 whose value is the sum of input vector

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

What are user-written functions

user-written functions [my term] are functions you write to perform some specific task, usually a data-manipulation or analysis task specific to your project

Like all functions, user-written functions follow three steps:

1. take in one or more inputs
2. process the inputs (this may include using pre-written functions like `select()` or `sum()`)
3. return a new object

Before showing you how to write your own functions, let's see an example of a function in action [FIX]

POTENTIAL EXAMPLES:

- SOME SPECIFIC TABULATION LIKE COUNT OF EVENT BY TYPE FOR EACH UNIVERSITY
- READ IN DATA
- FIX MISSING VALUES https://www.btskinner.me/rworkshop/modules/programming_one.html

When should you write a function

Wickham chapter 19 has some practical advice about when to write your own function.

Before stating this, let's introduce a task from the "off-campus recruiting project" that we might want to achieve by writing a function

Task:

- Dataset `df_event` has one observation for each university-recruiting_event for several public universities
  - Variable `event_type` identifies location type of recruiting event (e.g., public high school, private high school)
- We want to create the following descriptive statistics tables for each university
  - Table A: count of number of recruiting events by event type and the average of median income at each event type
  - Table B: same as Table A, but separately for in-state and out-of-state events

Here is some code to create these tables for Stonybrook University in New York

```r
df_event %>% filter(univ_id==196097) %>% group_by(event_type) %>%
  summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))

df_event %>% filter(univ_id==196097) %>% group_by(event_inst, event_type) %>%
  summarise(
    n_events=n(),
    mean_inc=mean(med_inc, na.rm = TRUE))
```

When should you write a function

A function is a self-contained bit of code that performs some specific task. Functions allow you to “automate” tasks that you perform more than once

The alternative to writing a function to perform some specific task is to copy and paste the code each time you want to perform a task

- e.g., for the off-campus recruiting descriptive stats, we would copy above code for each university and change the university ID

Grolemund and Wickham say:

> *“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).”*

Darin Christenson refers to the programming mantra **DRY**

> *Do not Repeat Yourself (DRY) - Functions enable you to perform multiple tasks (that are similar to one another) without copying the same code over and over*

Why write functions

Advantages of writing functions to complete a task compared to the copy-and-paste approach

- As task requirements change (and they always do!), you only need to revise code in one place rather than many places
- Functions give you an opportunity to make continual improvements to the way you complete a task
  - Often, I have two tasks and I write a separate function for each task. Over time, I realize that these two tasks have many things and common and that I can write a single function that completes both tasks.
- Reduce errors that are common in copy-and-paste approach (e.g., forgetting to change variable name or variable value)

Why write functions

How I use functions in my research (acquiring, processing, and analyzing data)

1. **Acquiring data**. Since I often create longitudinal datasets from annual "input data," I usually write a function or loop to read-in the data and do initial processing
   - ▷ After writing a function for a specific data source, I generalize the function to read-in other data sources that share commonalities
2. **Processing data** (the big step between acquiring data and analyzing data). Write functions for data processing steps:
   - ▷ sometimes these are small/quick steps that I do over and over (e.g., cleaning a "string" ID variable)
   - ▷ sometimes these are big/multi-step processes (e.g., writing a general function that takes-in longitudinal data on number of degrees degrees awarded by field an award-level for each university, and creates measures of "degree adoption")
3. **Analyzing data** (after creating analysis datasets). I **ALWAYS** write functions to automate analyses and the creation of tables/graphs
   - ▷ As a young research assistant, bosses were always asking me to change the variables and then recreate the regression tables
   - ▷ Functions flexible in terms of which models to run, variables to include, etc.

Learning how to write functions is a requirement for anybody working on my research projects

- ◦ When the RAs move on, I need to be able to efficiently modify tasks they completed. This is only possible when they write functions.

3 Function basics

Strategy for learning to write functions

How I'll approach teaching you how to write functions

1. Introduce the basic components of a function
2. Non-practical example:
   - start by writing a function that simply prints "hello"
   - then, we'll make iterative improvements to this function
3. Practical example: create descriptive tables for off-campus recruiting project
   - start by writing simple version of this function
   - then, we'll make iterative improvements to this function
4. student tasks: practice writing functions with a partner
5. Then, we'll introduce more complicated elements of writing a function (e.g., conditional execution)

Central theme is the importance of continually revising your functions

3.1 How to write a function

Three components of a function

The `function()` function tells R that you are writing a function

```
#to get help type "?base" in console and scroll index for "function", but help f
function_name <- function(x,y,z) {
  #function body
}
```

Three components of a function:

1. **function name**
   - specify function name before the assignment operator `<-`
2. **function arguments** (sometimes called “inputs”)
   - Inputs that the function takes
     - can be vectors, data frames, logical statements, etc.
   - in above hypothetical code, the function took three inputs `x` , `y` , `z`
     - we could have written this instead: `function(Larry,Curly,Moe)`
   - In “function call,” you specify values to assign to these function arguments
3. **function body**
   - What the function does to the inputs
   - Above hypothetical function doesn’t do anything

Hello function

Our first example is to write a function that simply prints “Hello!”

First step in writing a function to perform a task is always to perform the task outside of a function

```
"Hello!"
#> [1] "Hello!"
```

Create the function

```
print_hello <- function() {
  "Hello!"
}
```

1. **function name**
   - ▷ function name is `print_hello`
2. **function arguments** (sometimes called “inputs”)
   - ▷ the `print_hello` function doesn’t take any arguments
3. **function body**
   - ▷ What the function does to the inputs
   - ▷ body of `print_hello` simply prints “Hello!”

Call the function

```
print_hello()
#> [1] "Hello!"
```

Hello function

Task:

- modify the `print_hello` function so that it also prints our name, which we specify as an input.
- first, perform task outside a function. A few approaches we could take

This seems wrong because my name is not an input

```
"Hello! My name is Ozan Jaquette"
#> [1] "Hello! My name is Ozan Jaquette"
```

Why doesn't this work?

```
x <- "Ozan Jaquette"
x
#> [1] "Ozan Jaquette"
"Hello! My name is x"
#> [1] "Hello! My name is x"
```

Why doesn't this work?

```
"Hello! My name is " x
```

This sort of works

```
"Hello! My name is "
#> [1] "Hello! My name is "
x
```

Hello function

Task:

- modify the `print_hello` function so that it also prints our name, which we specify as an input.
- first, perform task outside a function. A few approaches we could take

Let's take another approach. Experiment with the `print()` function

```
#?print
print("Hello! My name is")
#> [1] "Hello! My name is"
print(x)
#> [1] "Ozan Jaquette"
```

Want our `print_hello` function to print everything on one line. Why doesn't this work?

```
print("Hello! My name is") print(x)
print("Hello! My name is"), print(x)
```

What went wrong? seems like `print()` function:

- Can only print one object at a time
- Can't put two instances of `print()` on same line of code
- Each instance of `print()` will be printed on separate line

Hello function

Task:

- modify the `print_hello` function so that it also prints our name, which we specify as an input.
- first, perform task outside a function. A few approaches we could take

We need to find an alternative to `print()` that can print multiple objects on the same line

Let's experiment with `cat()` function [I had to Google this]

```
#?cat
cat("Hello! My name is ")
#> Hello! My name is
cat(x)
#> Ozan Jaquette

cat("Hello! My name is ",x)
#> Hello! My name is  Ozan Jaquette
cat("Hello! My name is",x)
#> Hello! My name is Ozan Jaquette
```

Success! Now we can write a function for this task

Hello function

Task: modify `print_hello` function so that it also prints our name

Task outside of function

```r
x <- "Ozan Jaquette"
cat("Hello! My name is",x)
#> Hello! My name is Ozan Jaquette
```

Create function

```r
print_hello <- function(name) {
  cat("Hello! My name is",name)
}
```

1. **function name** is `print_hello`
2. **function arguments**. “inputs” to the function
   - `print_hello` function takes one argument, `name`
   - Instead of `name`, we could have named this argument `x` or `Ralph`
3. **function body**.What function does to the inputs
   - `cat("Hello! My name is",name)`

Call function

```r
print_hello("Patricia Martin")
#> Hello! My name is Patricia Martin
```

Hello function

Task: modify `print_hello` function so that it also takes our year of birth as an input and states our age

Perform task outside of function

```r
x <- "Ozan Jaquette"
y <- 1979
z <- 2018 - 1979
z
#> [1] 39
cat("Hello! My name is",x,". In 2018 I will turn",z,"years old")
#> Hello! My name is Ozan Jaquette . In 2018 I will turn 39 years old
```

Improvements we could make:

- use **date functions** to:
    1. specify current date (rather than manually typing “2018”)
    2. calculate age exactly (rather than as current year minus birth year)
    - But we haven’t learned date functions, so hold off
- use **string functions** to:
    - remove extra space between name and the period
    - but we haven’t learned string functions, so hold off

Hello function

Task: modify `print_hello` function so that it also takes our year of birth as an input and states our age

Perform task outside of function

```
cat("Hello! My name is",x,"and in 2018 I will turn",z,"years old")
#> Hello! My name is Ozan Jaquette and in 2018 I will turn 39 years old
```

Create function

```
print_hello <- function(name,birth_year) {
  age <- 2018 - birth_year
  cat("Hello! My name is",name,"and in 2018 I will turn",age,"years old")
}
```

1. **function name** is `print_hello`
2. **function arguments**. “inputs” to the function
   - ▷ `print_hello` function takes two arguments, `name` and `birth_year`
3. **function body**.What function does to the inputs
   - ▷ `age <- 2018 - birth_year`
   - ▷ `cat("Hello! My name is",name,"and in 2018 I will turn",age,"years old")`

Call function

```
print_hello("Ozan Jaquette",1979)
#> Hello! My name is Ozan Jaquette and in 2018 I will turn 39 years old
```

Recipe for writing a function

1. Experiment with performing the task outside of a function
   - experiment with performing task with different sets of inputs
   - sometimes you will have to revise this code, when an approach that worked outside a function does not work within a function
2. Write the function
3. Test the function; try to “break” it