CSSS508, Lecture 7

Functions

Michael Pearce

(based on slides from Chuck Lanfear)

November 10, 2022

UW CS&SS

Reminders

- Homework 6 is due at *midnight* tonight!
- Lab/Office Hours for next week are *modified:*
  - Office Hours: Monday (11/14) 8:30-10:30am; Thursday (11/17) 10am-12pm
  - Lab (11/15): Replaced with additional office hours (above!)
- Homework 7 will be due on on 11/17 at *midnight*
  - You will have time in class today to work on this homework!
  - *No assignment due Thanksgiving!*

Topics

Last time, we learned about,

1. Why we use loops
2. `for()` loops
3. `while()` loops

Today, we will cover,

1. Aside: Visualizing the Goal
2. Building blocks of functions
3. Simple functions
4. Using functions with `apply()`

1. Visualizing the Goal

Visualizing the Goal

Before you can write effective code, you need to know *exactly* what you want:

- **Goal:** Do I want a single value? vector? one observation per person? per year?
- **Current State:** What do I currently have? matrix, vector? long or wide format?
- **Translate:** How can I take what I have and turn it into my goal?
    - Sketch out the steps!
    - Break it down into little operations

As we become more advanced coders, this concept is key!!

Remember: *When you're stuck, try searching your problem on Google!!*

2. Building blocks of functions

Why Functions?

R (as well as mathematics in general) is full of functions!

We use functions to:

- Compute summary statistics (`mean()`, `sd()`, `min()`)
- Fit models to data (`lm(Fertility~Agriculture,data=swiss)`)
- Load data (`read_csv()`)
- Create ggplots (`ggplot()`)
- And so much more!!

Examples of Existing Functions

- `mean()`:
  - Input: a vector
  - Output: a single number

- `dplyr::filter()`:
  - Input: a data frame, logical conditions
  - Output: a data frame with rows removed using those conditions

- `readr::read_csv()`:
  - Input: a file path, optionally variable names or types
  - Output: a data frame containing info read in from file

Each function requires **inputs**, and returns **outputs**

Why Write Your Own Functions?

Functions encapsulate actions you might perform often, such as:

- Given a vector, compute some special summary stats
- Given a vector and definition of "invalid" values, replace with `NA`
- Defining a new logical operator

Advanced function applications (not covered in this class):

- Parallel processing
- Generating *other* functions
- Making custom packages containing your functions

Anatomy of a Function

```r
NAME <- function(ARGUMENT1, ARGUMENT2=DEFAULT){
  BODY
  return(OUTPUT)
}
```

- **Name**: What you call the function so you can use it later
- **Arguments** (aka inputs, parameters): things the user passes to the function that affect how it works
    - e.g. `ARGUMENT1`, `ARGUMENT2`
    - `ARGUMENT2=DEFAULT` is example of setting a default value
    - In this example, `ARGUMENT1`, `ARGUMENT2` values won't exist outside of the function
- **Body**: The actual operations inside the function.
- **Output**: The object inside `return()`. Could be anything (or nothing!)
    - If unspecified, will be the last thing calculated

3. Simple functions

Example 1: Doubling A Number

```r
double_x <- function(x){
  double_x <- x * 2
  return(double_x)
}
```

Let's run it!

```r
double_x(5)
```

```
## [1] 10
```

```r
double_x(NA)
```

```
## [1] NA
```

```r
double_x(1:2)
```

```
## [1] 2 4
```

Example 2: Extract First/Last

```r
first_and_last <- function(x) {
    first <- x[1]
    last  <- x[length(x)]
    return(c("first" = first, "last" = last))
}
```

Test it out:

```r
first_and_last(c(4, 3, 1, 8))
```

```
## first  last
##     4     8
```

Example 2: Testing `first_and_last`

What if I give `first_and_last()` a vector of length 1?

```
first_and_last(7)
```

```
## first  last
##     7     7
```

Of length 0?

```
first_and_last(numeric(0))
```

```
## first
##    NA
```

Maybe we want it to be a little smarter.

Example 3: Checking Inputs

Let's make sure we get an error message when the vector is too small:

```r
smarter_first_and_last <- function(x) {
    if(length(x) < 2){
      stop("Input is not long enough!")
    } else{
      first <- x[1]
      last  <- x[length(x)]
      return(c("first" = first, "last" = last))
    }
}
```

`stop()` ceases running the function and prints the text inside as an error message.

Example 3: Testing Smarter Function

```r
smarter_first_and_last(NA)
```

```
## Error in smarter_first_and_last(NA): Input is not long enough!
```

```r
smarter_first_and_last(c(4, 3, 1, 8))
```

```
## first  last
##     4     8
```

Cracking Open Functions

If you type a function name without any parentheses or arguments, you can see its contents:

```
smarter_first_and_last
```

```
## function(x) {
##     if(length(x) < 2){
##       stop("Input is not long enough!")
##     } else{
##       first <- x[1]
##       last  <- x[length(x)]
##       return(c("first" = first, "last" = last))
##     }
## }
## <bytecode: 0x7f8dd4d33f48>
```

Activity: Writing A Function

In Olympic diving, a panel of 7 judges provide scores. After removing the worst and best scores, the mean of the remaining scores is given to the diver. We'll write code to calculate this score!

1. Suppose I give you an ordered vector (lowest to greatest) of length 7. How can you keep elements 2 through 6 (i.e., remove the first and last values).
2. Write a function to calculate a diver's score:
    - Input: Ordered vector of length 7
    - Output: Mean score after removing the first and last scores.
    - Checks: Check that the vector has length 7.
3. Calculate the diver's score given `x <- c(1,3:7,10)`

Activity: My Solution

1. Extract elements 2 through 6:
    - **Answer:** Given vector `x`, use `x[2:6]`
2. Function

```
divers_score <- function(x){
  if(length(x) != 7){
    stop("x is not of length 7!")
  } else{
    x_nofirst_nolast <- x[2:6]
    return(mean(x_nofirst_nolast))
  }
}
```

1. Calculate the diver's score given `x <- c(1,3:7,10)`

```
divers_score(x = c(1,3:7,10))
```

```
## [1] 5
```

4. Using functions with `apply()`

Applying Functions Multiple Times?

Last week, we saw an example where we wanted to take the mean of each column in the `swiss` data:

```r
for(col_index in seq_along(swiss)){
  mean_swiss_col <- mean(swiss[,col_index])
  names_swiss_col <- names(swiss)[col_index]
  print(c(names_swiss_col,round(mean_swiss_col,3)))
}
```

```
## [1] "Fertility" "70.143"
## [1] "Agriculture" "50.66"
## [1] "Examination" "16.489"
## [1] "Education" "10.979"
## [1] "Catholic" "41.144"
## [1] "Infant.Mortality" "19.943"
```

Isn't this kind of complex?!

`apply()`, don't loop!

Writing loops can be challenging and prone to bugs!!

The `apply()` can solve this issue:

- **apply** a function to values in each row or column of a matrix
- Doesn't require preallocation
- Can take built-in functions or user-created functions.

Structure of `apply()`

`apply()` takes 3 arguments:

1. Data (a matrix or data frame)
2. Margin (1 applies function to each *row*, 2 applies to each *column*)
3. Function

```
apply(DATA, MARGIN, FUNCTION)
```

For example,

```
apply(swiss, 2, mean)
```

```
##        Fertility      Agriculture      Examination        Education
##         70.14255         50.65957         16.48936         10.97872
##         Catholic Infant.Mortality
##         41.14383         19.94255
```

Example 1

```r
row_max <- apply(swiss,1,max) #maximum in each row
head(row_max,20)
```

```
##    Courtelary      Delemont Franches-Mnt       Moutier    Neuveville
##         80.20         84.84        93.40         85.80         76.90
##    Porrentruy         Broye         Glane       Gruyere        Sarine
##         90.57         92.85        97.16         97.67         91.38
##       Veveyse         Aigle       Aubonne      Avenches      Cossonay
##         98.61         64.10        67.50         68.90         69.30
##     Echallens      Grandson      Lausanne     La Vallee        Lavaux
##         72.60         71.70        55.70         54.30         73.00
```

Example 2

```
apply(swiss,2,summary) # summary of each column
```

```
##         Fertility Agriculture Examination Education  Catholic
## Min.     35.00000     1.20000     3.00000   1.00000   2.15000
## 1st Qu.  64.70000    35.90000    12.00000   6.00000   5.19500
## Median   70.40000    54.10000    16.00000   8.00000  15.14000
## Mean     70.14255    50.65957    16.48936  10.97872  41.14383
## 3rd Qu.  78.45000    67.65000    22.00000  12.00000  93.12500
## Max.     92.50000    89.70000    37.00000  53.00000 100.00000
##         Infant.Mortality
## Min.            10.80000
## 1st Qu.         18.15000
## Median          20.00000
## Mean            19.94255
## 3rd Qu.         21.70000
## Max.            26.60000
```

**Note:* Matrix output!

Example 3: User-Created Function

```
scores <- matrix(1:21,nrow=3)
print(scores)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    4    7   10   13   16   19
## [2,]    2    5    8   11   14   17   20
## [3,]    3    6    9   12   15   18   21
```

```
apply(scores,1,divers_score)
```

```
## [1] 10 11 12
```

Activity

These are homework questions!!

1. What's similar/different between a `for` loop and `apply`?

2. Write an `apply()` function to take the median value of each column in the `cars` dataset

3. Using `ggplot`, make a scatterplot of the `speed` and `dist` variables in `cars`. Then, add an appropriate horizontal and vertical line symbolizing the median value of each variable.

Hint: Using the layers `geom_vline(xintercept = )` and `geom_hline(yintercept = )`

My Answers

1. What's similar/different between a `for` loop and `apply`?

   - **Possible Answer:** `for` loops and `apply` can usually accomplish the same tasks: Each uses code iteratively.
   - `for` loops are more flexible, in that you can provide different data using indices to the loop; they are harder to write.
   - `apply` loops are simpler and don't require preallocation; they require your data be a matrix or dataframe.

2. Write an `apply()` function to take the median value of each column in the `cars` dataset

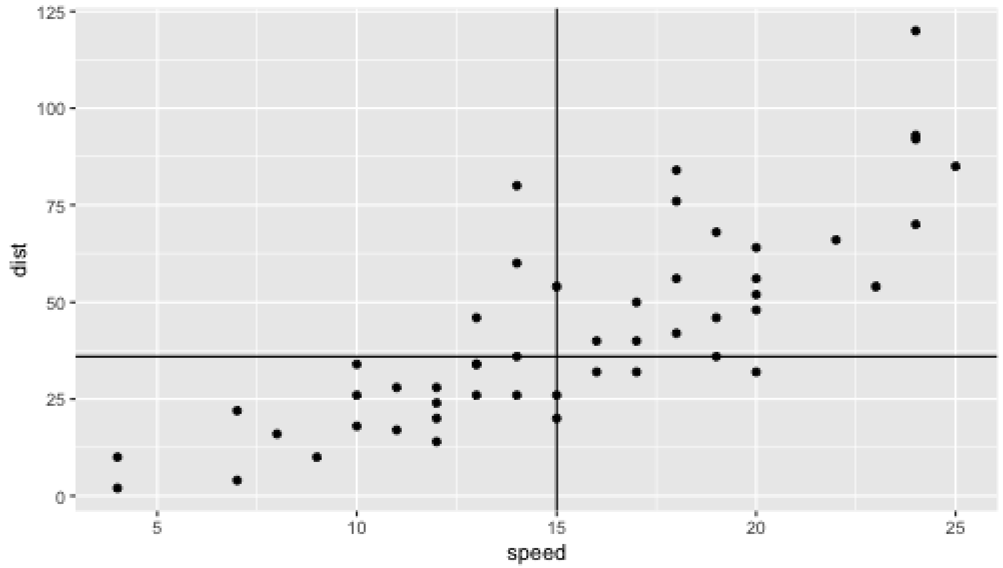
```
median_cars <- apply(cars,2,median)
median_cars
```

```
## speed  dist
##    15    36
```

My Answers

1. Make a ggplot

```
library(ggplot2)
ggplot(cars,aes(speed,dist))+geom_point()+
  geom_vline(xintercept = median_cars[1])+
  geom_hline(yintercept = median_cars[2])
```



Homework

Time to work on Homework 7!