Introduction to R for Data Analysis in the Health Sciences: Lecture 8

Amy Willis, Biostatistics, UW

05 December, 2019

Today: Advanced programming

- Writing functions
  - Basic structure
  - Conditionals: `if()`
- Automating repetitive or iterative procedures with loops
  - Looping: `for()`
  - Avoiding loops where possible

As always. . .

```r
library(tidyverse)
```

```
## -- Attaching packages ------------------------------------

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------- tidy
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

As always...

```r
airquality <- read_csv("datasets/airquality.csv")
```

```
## Parsed with column specification:
## cols(
##   Ozone = col_double(),
##   Solar.R = col_double(),
##   Wind = col_double(),
##   Temp = col_double(),
##   Month = col_double(),
##   Day = col_double()
## )
```

Writing functions

Functions are created using the function `function()`

```r
my_first_function <- function(input1, input2, ...) {
    # # commands go here, e.g.:
    # output_object <- input1 + input2

    # # output something, e.g.
    # return(output_object)
}
```

4 parts to a function

There are 4 parts to a function

1. The name of the function (e.g., `my_function`) and the creation of the function via `function(...)`

```
my_function <- function(x, y) {
    x + y
}
```

4 parts to a function

2. Inputs to the function (e.g. `input1, input2, ...`). These are passed to the function and used to perform calculations.
    - There can be as many inputs as you choose!
    - Programming speak for “input” is “argument”

```r
my_function <- function(x, y) {
    x + y
}

another_function <- function(amy) {
    paste(amy, "is glad you are here!")
}
another_function("Your BIOST 509 team")
```

```
## [1] "Your BIOST 509 team is glad you are here!"
```

4 parts to a function

3. The body of the function. This tells R what to do with the inputs:

```
my_function <- function(x, y) {
    x + y
}
```

4 parts to a function

4. The output of the function. This is what is returned to the user at the end of the function. If the last line of the function contains an expression, it will be returned. You can be explicit by wrapping what-you-want-returned in `return()`

```r
my_function2 <- function(x, y) {
    return(x + y)
}
# ...does the same thing as:
my_function <- function(x, y) {
    x + y
}
```

Functions can return only a single object, but it can be a list containing many objects. *We will discuss lists next week!*

Writing functions: examples

Here's a function for calculating the coefficient of variation (the ratio of the standard deviation to the mean). It takes a vector as input:

```
coef_of_var <- function(x){
   meanval <- mean(x, na.rm = TRUE)
   sdval   <- sd(x, na.rm = TRUE)
   return(sdval / meanval)
}
```

Translated, this function says "if you give me an object, that I will call `x`, I will store its `mean` as `meanval`, then its `sd` as `sdval`, and then return their ratio `sdval / meanval`."

Using a function you created

```r
coef_of_var(airquality$Ozone)
```

```
## [1] 0.7830151
```

Using a function you created

If you just type the function name, R gives you the definition of the function:

```
coef_of_var
```

```
## function(x){
##    meanval <- mean(x, na.rm = TRUE)
##    sdval   <- sd(x, na.rm = TRUE)
##    return(sdval / meanval)
## }
```

Writing a more complex function

Today we're going to analyze some microbiome data!

- Very generally speaking, having more a more diverse set of bacteria living in your gut is healthier
- *Important question*: How many bacteria live in your gut? How many do you lose after a course of antibiotics?
- *Big problem*: We cannot sequence *all* the bacteria in your gut, but we can sequence *some*.

How can we estimate how many bacteria were missing from our sample but present in the gut?

This is called the "species problem"

Implementing the Chao1 estimator

A ubiquitous estimator based on a very simple Poisson model is the "Chao1" estimator.

Let c be the number of species observed in the sample, f_1 be the number of species observed once, and f_2 be the number of species observed twice. Then the Chao1 estimate of total diversity is

$$\hat{C}_{Chao1} = c + \frac{f_1^2}{2f_2}$$

Let's write a function that calculates the Chao1 estimate of total diversity!

Implementing the Chao1 estimator

Start by asking yourself: What input does my function take?

- We could write a function that takes in c, f_1 and f_2. But then the user would have to calculate them!
- Let's write a function that takes in a vector of bacterial species counts (the number of times each species was observed in the sample), e.g.

```r
my_counts <- c(5, 1, 1, 7, 20, 2, 1, 1550, 1, 2)
```

In this dataset, the first taxon is observed 5 times; the second taxon is observed once, etc.

$f_1 = 4$, $f_2 = 2$ and $c = 10$

Implementing the Chao1 estimator

```
my_chao1 <- function(my_data) {
  c <- sum(my_data > 0)
  f1 <- sum(my_data == 1)
  f2 <- sum(my_data == 2)
  c + f1^2 / (2*f2)
}
my_chao1(my_counts)
```

```
## [1] 14
```

Implementing the Chao1 estimator

```r
my_counts <- c(5, 1, 1, 7, 20, 2, 1, 1550, 1, 2)
my_chao1(my_counts)
```

```
## [1] 14
```

Very important to do a "sanity check": a simple test that your function works *as you expect* in a case you understand.

Control structures: changing behaviour based on a condition

Problem! What happens if $f_2 = 0$?

Solution: tell the function to behave differently if $f_2 = 0$

The `if()` control

The basic structure of a `if()` conditional:

```
if (condition_holds) {
  # do something
}
```

If the condition holds, the code between the braces runs. If the condition doesn't hold, the code between the braces **does not** *run.*

The `if()` `else` control

The basic structure of a `if()` `else` conditional:

```
if (condition_holds) {
  # do something
} else {
  # do something different
}
```

If the condition holds, the code between the first braces runs. If the condition doesn't hold, the code between the second braces runs.

Returning to the Chao1 example…

Important first step *before coding*: **Decide what you want the function to do!**

$$\hat{C}_{Chao1} = c + \frac{f_1^2}{2f_2}$$

If both f_1 and f_2 are zero, $\hat{C} = c$. If $f_1 > 0$ and $f_2 = 0$, then the estimate is undefined and we return `NA`.

Returning to the Chao1 example. . .

```r
my_even_better_chao1 <- function(my_data) {
  c <- sum(my_data > 0)
  f1 <- sum(my_data == 1)
  f2 <- sum(my_data == 2)

  if (f2 == 0) {
    chat <- ifelse(f1 == 0, c, NA)
  } else {
    chat <- c + f1^2 / (2*f2)
  }
  chat
}
```

Notice how `ifelse()` is used differently from `if (...) { ... } else {...}`

Closing comments on functions

Writing functions is great! I encourage you to do it often!

- Almost any task that you would run more than once benefits from being a function!
- Break down complex tasks into multiple functions
  - Avoid megafunctions!
- Writing complex functions takes time and practice
  - Advanced tutorials on writing functions in `R`: `http://adv-r.had.co.nz/Functions.html`
- Save yourself time by checking if someone has implemented the same function in a `R` package

Questions?

The for() loop

For each value of a counter, do something

```
for (counter in set_of_values){
   # code that depends on 'counter'
}
```

The `for()` loop

A silly example:

```
for (i in 1:5) {
   print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

- `i` is the *counter*; takes incremental values from 1 to 5
- *code that depends on* `i`: print the value of `i`

Applying a function to every column using a loop

Let's analyse the data from *The Pervasive Effects of an Antibiotic on the Human Gut Microbiota, as Revealed by Deep 16S rRNA Sequencing* by Les Dethlefsen, Sue Huse, Mitchell L Sogin, and David A Relman (PLoS Biology, 2008).

This was an early paper looking at the long-term effects of antibiotics on the gut microbiome using a particular type of microbial sequencing (that was new at the time)

Applying a function to every column using a loop

```
antib_tib <- read_csv("datasets/microbial_abundances.csv")
```

```
## Parsed with column specification:
## cols(
##   `refOTU designation` = col_character(),
##   A1 = col_double(),
##   A2a = col_double(),
##   A2b = col_double(),
##   A2c = col_double(),
##   A3a = col_double(),
##   A3b = col_double(),
##   A4 = col_double(),
##   A5 = col_double(),
##   B1 = col_double(),
##   B2 = col_double(),
##   B3 = col_double(),
##   B4 = col_double(),
##   B5 = col_double(),
```

Applying a function to every column using a loop

The columns index the samples, and the rows index the bacterial taxa:

```
antib_tib
```

```
## # A tibble: 5,670 x 19
##    `refOTU designa~    A1   A2a   A2b   A2c   A3a   A3b
##    <chr>            <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 V3_Gp1_refOTU_1      0     0     3     0     0     6
##  2 V3_Gp3_refOTU_1      2     0     0     0     0     0
##  3 V3_Actinobaculu~     0     0     0     6     0     0
##  4 V3_Actinomyces_~     0     0     0     0     2     2
##  5 V3_Actinomyces_~     0     0     0     0     2     2
##  6 V3_Actinomyces_~     0     0     0     0     2     0
##  7 V3_Actinomyces_~     0     4     0     0     0     0
##  8 V3_Actinomyces_~     0     0     0     0     2     0
##  9 V3_Actinomyces_~     0     0     0     0     0     2
## 10 V3_Actinomyces_~     0     0     0     0     0     2
## # ... with 5,660 more rows, and 9 more variables: B2 <d
```

for() loops

Let's loop over the columns, storing the Chao1 index for each column.

First, let's create an empty vector to contain the estimates

```
estimate <- rep(NA, ncol(antib_tib) - 1)
estimate
```

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

for() loops

```
for (i in 2:ncol(antib_tib)) {
  df <- antib_tib[,i]
  estimate[i-1] <- my_even_better_chao1(df)
}
```

for() loops

```
tibble("sample" = names(antib_tib)[-1],
       "estimates" = estimate)
```

```
## # A tibble: 18 x 2
##    sample estimates
##    <chr>      <dbl>
##  1 A1         1116
##  2 A2a         917
##  3 A2b         983
##  4 A2c         770
##  5 A3a         511
##  6 A3b         492
##  7 A4          931
##  8 A5         1027
##  9 B1           NA
## 10 B2         2010.
## 11 B3          764
## 12 B4          908
```

for() loops: sanity check

```
sum(antib_tib[,2] > 0) +
sum(antib_tib[,2] == 1)^2 /
(2 * sum(antib_tib[,2] == 2))
```

```
## [1] 1116
```

Avoid loops where possible

Sometimes loops are necessary, but they can often be avoided. Try to avoid them where possible.

Why?

- Faster
- Easier to read & debug

Avoid loops where possible

```
salary <- read_csv("../BIOST_509_Intro_R_Autumn_2018/Datase
```

```
## Warning: Missing column names filled in: 'X1' [1]

## Parsed with column specification:
## cols(
##   X1 = col_double(),
##   case = col_double(),
##   id = col_double(),
##   gender = col_character(),
##   deg = col_character(),
##   yrdeg = col_double(),
##   field = col_character(),
##   startyr = col_double(),
##   year = col_double(),
##   rank = col_character(),
##   admin = col_double(),
##   salary = col_double()
```

Avoid loops where possible

Suppose we want to get only the last observation for each person. We could loop through all of the individuals… or we could use `group_by`:

```
salary %>%
  group_by(id) %>%
  filter(year == max(year)) %>%
  ungroup
```

```
## # A tibble: 1,597 x 12
##       X1  case    id gender deg   yrdeg field startyr  year rank  ad
##    <dbl> <dbl> <dbl> <chr>  <chr> <dbl> <chr>   <dbl> <dbl> <chr> <d
##  1     1     1     1 F      Other    92 Other      95    95 Assi~
##  2     3     3     2 M      Other    91 Other      94    95 Assi~
##  3     4     4     4 M      PhD      96 Other      95    95 Assi~
##  4     9     9     6 M      PhD      66 Other      91    95 Full
##  5    29    29     7 M      PhD      70 Other      71    95 Assoc
##  6    30    30     8 M      PhD      75 Other      95    95 Assi~
##  7    39    39     9 M      PhD      82 Other      87    95 Assoc
##  8    55    55    10 M      PhD      68 Arts       80    95 Full
##  9    75    75    12 M      PhD      64 Other      64    95 Full
## 10    95    95    13 M      PhD      68 Prof       69    95 Full
```

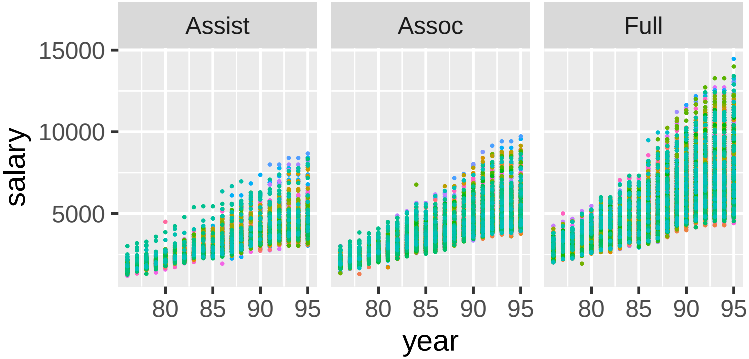
Avoid loops where possible

Suppose we wanted to look at the trajectories over time of salaries for each rank. We could loop through all ranks… or we could use `facet_wrap` or `facet_grid`.

Avoid loops where possible

```r
salary %>% mutate(id = as.character(id)) %>%
  filter(!is.na(rank)) %>%
  ggplot(aes(x = year, y = salary, col = id)) +
  geom_point(cex = 0.1) + theme(legend.position = "none") +
  facet_grid(~rank)
```

```
## Warning: Removed 4 rows containing missing values (geom_point).
```



Summary: Advanced programming

- Writing functions
  - A great way to avoid copying and pasting code
- Writing loops
  - `for()` loops: Essential for serious computing jobs; helpful for data management too (see HW)
  - `if()` and `if() else` structures allow blocks of code to be executed under different specified conditions
  - Other types of loops exist (`repeat`, `while`); they are less common

Questions?

Housekeeping

- **No lecture next week!**
  - Office hours: Tuesday & Wednesday
- Exercise 8 is the last (graded) exercise
- Homework 8 is the last (graded) exercise
  - Homework 8 is a “capstone” homework; it is longer and more challenging than previous homeworks, so please start it early.
  - Due 5 p.m. Wednesday 4 December
- You need 28 points (out of a possible 32) to get credit for the class.
  - *Please contact me ASAP if you do not think you are on track to get 28 points. I want you all to get credit for your hard work and learning!*

Questions?

In two weeks…

…we will discuss cool miscellanea!

- lists
- the `apply` family
- bootstrapping
- Making nice tables for publication

In the meantime… **Happy Holidays!**

The plan

- 5 minute break
- In-class exercise available via Canvas
  - Designed to be completed by 3:20 p.m.
  - Due today 6:30 p.m.
  - *Yellow sticky note* = urgent; *blue sticky note* = non-urgent
- Homework due 5 p.m. Wednesday 4 December
- Office hours: Tuesday & Wednesday *only*