Introduction to R

Module 3: Conditionals, loops, functions, and joins

Andrew Proctor

andrew.proctor@phdstudent.hhs.se

February 5, 2018

Intro

Goals for Today

1. Basics of programming in R—learn how to write and use:
   - Iterations (loops and map functions)
   - Conditional statements
   - Basic functions
2. Learn how to perform different types of dataset joins.

Iteration

For loops

For tasks that you want to iterate over multiple data frames/variables/elements, you may want to think about creating a **loop**.

- A loop performs a function/functions multiple times, across either a list of objects or a set of index values.

Syntax:

```
for(indexname in range) {
  do stuff
}
```

For loop across numeric values

```
for (i in 1:4){
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

For loop across named elements

You can also loop over elements instead of values.

- In the last module exercises, you had to convert many columns to numeric. Here's how you could do that with a loop:

```
wdi_data <- import("wdi_data.rds")

vars <-colnames(wdi_data)
indicators <- vars[-(1:4)]

for (i in indicators){
  wdi_data[,i] <- as.numeric(wdi_data[,i])
}
```

The map() function

For iterations over vectors and dataframes, the map() function is a great alternative to the for loop.

Map functions take a user-supplied function and iterate it over:

- Elements for a vector
- Objects of a list
- Columns of a data frame

Map functions are much simpler to write than loops and are also generally a good bit faster.

- ***Sidenote***: Map is a part of the tidyverse collection of packages. In base R, the apply() family of functions does roughly the same thing, but map() simplifies and improves this task.

Using the map() function

Syntax:

```
map(data, fxn, option1, option2...)
```

Example:

```
wdi_data[,4:15] <- map(wdi_data[,4:15], as.numeric)
```

Applies the function as.numeric to the columns 4-15 of wdi_data.

Using class-specific map variants

There are multiple map variants that enforce a given data type on results. You should use these whenever you want output of a certain class.

- map_lgl for logical vector
- map_dbl for numeric vector
- map_chr for character vector
- map_df for a data frame

```
map_dbl(wdi_data, IQR, na.rm=TRUE)
```

```
##       GDP_pc          pop          pov
##     10548.89  40860583.00        65.40
```

Using map() with anonymous functions

map() works with not only predefined functions, but also "anonymous functions"— unnamed functions defined inside of map().

- In the example below, I z-standardize the values of year, GDP, population, and poverty.

```
ztransform <- map_df(wdi_data, function(x)
  (x - mean(x,  na.rm=TRUE)) / sd(x, na.rm=TRUE)
  )
```

Did it work?

```
map_dbl(ztransform, function(x)
  round(mean(x, na.rm=TRUE),10))
```

```
##    year GDP_pc     pop     pov
##       0      0       0       0
```

```
map_dbl(ztransform, function(x)
  round(sd(x, na.rm=TRUE),10))
```

```
##    year GDP_pc     pop     pov
##       1      1       1       1
```

Conditional Statements

If statement

"If statements" are also a useful part of programming, either in conjunction with iteration or seperately.

- An if statement performs operations only if a specified condition is met.
    - An important thing to know, however, is that if statements evaluate conditions of length one (ie non-vector arguments).
    - We will cover a vector equivalent to the if statement shortly.

Syntax

```
if(condition){
     do stuff
   }
```

Example of an if statement

- In the for loop example, the loop was indexed over only the columns of indicator codes.
- Equally, the loop could be done over all columns with an if-statement to change only the indicator codes.

```
for (j in colnames(wdi_data)){

  if(j %in% indicators){
      wdi_data[,j] <- as.numeric(wdi_data[,j])
  }
}
```

Multiple conditions

You can encompass several conditions using the **else if** and catch-all **else** control statements.

```
if (condition1) {
do stuff
} else if (condition2) {
do other stuff
} else {
do other other stuff
}
```

Vectorized if statements

- As alluded to earlier, if statements can't test-and-do for vectors, but only single-valued objects.
- Most of the time, you probably want to use conditional statements on vectors. The vector equivalent to the if statement is ifelse()

Syntax:

```
ifelse(condition, true_statement, false_statement)
```

The statements returned can be simple values, but they can also be functions or even further conditions. You can easily nest multiple ifelses if desired.

An ifelse example

```
numbers <- sample(1:30, 7); numbers
```

```
## [1] 10 30  7 24 14 21 12
```

```
ifelse(numbers %% 2 == 0,"even","odd")
```

```
## [1] "even" "even" "odd"  "even" "even" "odd"  "even"
```

Note: What if we tried a normal if statement instead?

```
if(numbers %% 2 == 0){
  print("even")} else{
    print("odd")}
```

```
## [1] "even"
```

Multiple vectorized if statements

A better alternative to multiple nested `ifelse` statements is the tidyverse case_when function.

Syntax:

```
case_when(
  condition1 ~ statement1,
  condition2 ~ statement2,
  condition3 ~ statement3,
)
```

A case_when example

```
nums_df <- numbers %>% as.tibble() %>%
  mutate(interval = case_when(
   (numbers > 0 & numbers <= 10) ~ "1-10",
   (numbers > 10 & numbers <= 20) ~ "10-20",
   (numbers > 20 & numbers <= 30) ~ "20-30"))
nums_df[1:4,]
```

```
## # A tibble: 4 x 2
##   value interval
##   <int> <chr>
## 1    10 1-10
## 2    30 20-30
## 3     7 1-10
## 4    24 20-30
```

Functions

When you should write a function

If you find yourself performing the same specific steps more than a couple of times (perhaps with slight variations), then you should consider writing a function.

A function can serve essentially as a wrapper for a series of steps, where you define generalized inputs/arguments.

Writing a function

Ingredients:

- Function name
- Arguments
- Function body

Syntax:

```
function_name <- function(arg1, arg2, ...){
  do stuff
}
```

Function example

Let's turn the calculation of even or odd that was completed earlier into a function:

```
# Make odd function
odd <- function(obj){
   ifelse(obj %% 2 == 0,"even","odd")
}
```

Notice that *obj* here is a descriptive placeholder name for the data object to be supplied as an argument for the function.

```
odd(numbers)
```

```
## [1] "even" "even" "odd"  "even" "even" "odd"  "even"
```

RStudio's "Extract Function"

A useful way of writing simple functions when you've already written the code for a specific instance is to use RStudio's *Extract Function* option, which is available from the code menu.

- *Extract function* will take the code chunk and treat any data objects referenced but not created within the chunk as function arguments.

Joins

Merging data

Shifting gears from programming...

Another staple task in applied work is combining data from multiple data sets. The tidyverse set of packages includes several useful types of merges (or "joins"):

- **left_join()** Appends columns from dataset B to dataset A, keeping all observations in dataset A.
- **inner_join()** Appends columns together, keeping only observations that appear in both dataset A and B.
- **semi_join()** Keeps only columns of dataset A for observations that appear in both dataset A and B.
- **anti_join()** Keeps only columns of dataset A for observations that *do not* appear in both dataset A and B.

Joining using keys

The starting point for any merge is to enumerate the column or columns that uniquely identify observations in the dataset.

- For cross-sectional data, this might be a personal identifier or (for aggregate data) something like municipality, state, country, etc.
- For panel data, this will typically be both the personal/group identifier and a timing variable, for example Sweden in 2015 in a cross-country analysis.

Mismatched key names across datasets

Sometimes the names of the key variables are different across datasets.

- You could of course rename the key variables to be consistent.
- But mismatched key names are easily handled by the tidyverse join functions.

Syntax:

```
join_function(x, y, by = c("x_name" = "y_name"))
```

left_join

The **left_join()** is the most frequent type of join, corresponding to a standard **merge** in Stata.

- left_join simply appends additional variables from a second dataset to a main dataset, keeping all the observations (rows) of the first dataset.

Syntax:

```
left_join(x, y, by = "key")
```

If the key is muliple columns, use **c()** to list them.

left_join example

```r
# Look at the datasets
earnings
```

```
##   person_id wage
## 1       001  150
## 2       002   90
## 3       003  270
```

```r
educ
```

```
##   person_id schooling
## 1       001        12
## 2       003         8
## 3       004        16
```

left_join example ctd

```
combined_data <- left_join(earnings, educ,
                           by="person_id")

combined_data
```

```
##   person_id wage schooling
## 1       001  150        12
## 2       002   90        NA
## 3       003  270         8
```

Notice that schooling is equal to NA for person '002' because that person does not appear in the *educ* dataset.

inner_join

If you want to combine the variables of two data sets, but only keep the observations present in both datasets, use the inner_join() function.

```
combined_data <- inner_join(earnings, educ,
                            by="person_id")
combined_data
```

```
##   person_id wage schooling
## 1       001  150        12
## 2       003  270         8
```

semi_join

To keep using only the variables in the first dataset, but where observations in the first dataset are matched in the second dataset, use semi_join().

- semi_join is an example of a *filtering join*. Filtering joins don't add new columns, but instead just filter observations for matches in a second dataset.
- left_join and inner_join are instead known as *mutating joins*, because new variables are added to the dataset.

```
filtered_data <- semi_join(earnings, educ, by="person_id")
filtered_data
```

```
##   person_id wage
## 1       001  150
## 2       003  270
```

anti_join

Another *filtering join* is anti_join(), which filters for observations that are *not matched* in a second dataset.

```
filtered_data <- anti_join(earnings, educ,
                           by="person_id")
filtered_data
```

```
##   person_id wage
## 1       002   90
```

There are still other join types, which you can read about here.