Chapter 3: Style

Andrew Flowers

2/27/2017

Let's not kid ourselves: reading code is often terrible. Especially when you wrote it ages ago and were too rushed (or lazy) to properly clean and document your work. That's why having good style is so important.

There are several good R style guides. I can recommend [Google's R Style Guide](https://google.github.io/styleguide/Rguide.xml). Hadley Wickham has [great advice](http://adv-r.had.co.nz/Style.html), too. Reading those guides is a great start. But what I want to focus on here is the importance of *readability* in writing R code, specifically **for data science**.

### Embracing the power of piping with %>%

I'm of the strong opinion that the secret to readable R code comes down to one habit: liberal use of the %>% operator, or the "pipe" operator as its known. The %>% orginiated in the maggrittr package, but has become a de facto style of the tidyverse packages. The %>% allows you to write *chained* R commands. And it avoids ugly nested code.

Here's how to it works. We'll be using the data from the pro basketball (my favorite sport). [FiveThirtyEight](fivethirtyeight.com), my old employer, was fond of calculating Elo ratings for sports teams. Elo ratings originated in chess, and are a simple way of measuring quality. You can download the Elo ratings of *every* NBA team, before and after *every* NBA game, from the 1947 season through the 2015 season. The data is online [here](https://raw.githubusercontent.com/fivethirtyeight/data/master/nba-elo/nbaallelo.csv) (and can also be found on this book's [GitHub repository](https://github.com/andrewflowers/how-to-make-mistakes-in-R)).

First, let's make sure the tidyverse package is loaded. Next, we'll read the data into R using the read\_csv function from the readr package. This might take your computer some time -- it's over 126,000 games! Then we'll calculate the average post-game Elo rating for all teams in the dataset (the pre-game elo rating is elo\_i, the post-game rating is elo\_n). I'll show you two ways to do this: one straightforward way using mean() and anothering using the pipe (%>%) operator.

library(tidyverse)

## Loading tidyverse: ggplot2  
## Loading tidyverse: tibble  
## Loading tidyverse: tidyr  
## Loading tidyverse: readr  
## Loading tidyverse: purrr  
## Loading tidyverse: dplyr

## Conflicts with tidy packages ----------------------------------------------

## filter(): dplyr, stats  
## lag(): dplyr, stats

# Load data  
nba\_elo\_data <- read\_csv("https://raw.githubusercontent.com/fivethirtyeight/data/master/nba-elo/nbaallelo.csv")

## Parsed with column specification:  
## cols(  
## .default = col\_character(),  
## gameorder = col\_integer(),  
## `\_iscopy` = col\_integer(),  
## year\_id = col\_integer(),  
## seasongame = col\_integer(),  
## is\_playoffs = col\_integer(),  
## pts = col\_integer(),  
## elo\_i = col\_double(),  
## elo\_n = col\_double(),  
## win\_equiv = col\_double(),  
## opp\_pts = col\_integer(),  
## opp\_elo\_i = col\_double(),  
## opp\_elo\_n = col\_double(),  
## forecast = col\_double()  
## )

## See spec(...) for full column specifications.

# Calculate average Elo rating over time  
mean(nba\_elo\_data$elo\_n)

## [1] 1495.236

# Do the same, using pipes  
nba\_elo\_data %>% summarize(avg\_elo = mean(elo\_n))

## # A tibble: 1 × 1  
## avg\_elo  
## <dbl>  
## 1 1495.236

*Wait, what? Piping requires writing more code?!*

Hang on a second, this example is just meant to illustrate how the %>% operator works. Here the pipe is passing the nba\_elo\_data dataframe to the summarize() function (from the dplyr package). Not having to specify the dataframe you're operating on for each function will come in handy when you're calling multiple functions.

Let's continue to work with the nba\_elo\_data dataset, and continue using basic dplyr functions but writing code with and without pipes. First, let's calculate the highest Elo rating for each team -- or franchise, really -- by grouping each game according to the fran\_id column. Notice that with pipes, you can place each piece of code on a new line, as long as the line ends with the %>% operator.

library(tidyverse)  
  
# Code without pipes  
summarize(group\_by(nba\_elo\_data, fran\_id), best\_elo = max(elo\_n))

## # A tibble: 53 × 2  
## fran\_id best\_elo  
## <chr> <dbl>  
## 1 Baltimore 1513.935  
## 2 Bombers 1467.990  
## 3 Bucks 1757.149  
## 4 Bulls 1853.104  
## 5 Capitols 1563.444  
## 6 Cavaliers 1765.244  
## 7 Celtics 1815.692  
## 8 Clippers 1743.113  
## 9 Colonels 1672.633  
## 10 Condors 1499.175  
## # ... with 43 more rows

# Code with pipes  
nba\_elo\_data %>%   
 group\_by(fran\_id) %>%   
 summarize(best\_elo = max(elo\_n))

## # A tibble: 53 × 2  
## fran\_id best\_elo  
## <chr> <dbl>  
## 1 Baltimore 1513.935  
## 2 Bombers 1467.990  
## 3 Bucks 1757.149  
## 4 Bulls 1853.104  
## 5 Capitols 1563.444  
## 6 Cavaliers 1765.244  
## 7 Celtics 1815.692  
## 8 Clippers 1743.113  
## 9 Colonels 1672.633  
## 10 Condors 1499.175  
## # ... with 43 more rows

*Ahhh*, see? The code with pipes is far easier on the eye. We can take this point a step further: let's filter out all the games before 1980 and then sort the resulting dataframe from highest to lowest peak franchine Elo rating. Notice how ugly the nested code becomes when the pipes aren't used.

library(tidyverse)  
  
# Code without pipes  
arrange(summarize(group\_by(filter(nba\_elo\_data, year\_id >= 1980), fran\_id), best\_elo = max(elo\_n)), desc(best\_elo))

## # A tibble: 30 × 2  
## fran\_id best\_elo  
## <chr> <dbl>  
## 1 Bulls 1853.104  
## 2 Warriors 1822.288  
## 3 Celtics 1815.692  
## 4 Lakers 1789.993  
## 5 Pistons 1788.091  
## 6 Magic 1781.763  
## 7 Sixers 1777.069  
## 8 Heat 1774.346  
## 9 Mavericks 1773.131  
## 10 Spurs 1771.188  
## # ... with 20 more rows

# Code with pipes  
nba\_elo\_data %>%   
 filter(year\_id >= 1980) %>%   
 group\_by(fran\_id) %>%   
 summarize(best\_elo = max(elo\_n)) %>%   
 arrange(desc(best\_elo))

## # A tibble: 30 × 2  
## fran\_id best\_elo  
## <chr> <dbl>  
## 1 Bulls 1853.104  
## 2 Warriors 1822.288  
## 3 Celtics 1815.692  
## 4 Lakers 1789.993  
## 5 Pistons 1788.091  
## 6 Magic 1781.763  
## 7 Sixers 1777.069  
## 8 Heat 1774.346  
## 9 Mavericks 1773.131  
## 10 Spurs 1771.188  
## # ... with 20 more rows

The fist bit of code is so messy and hard to understand that I made a mistake coding this example!

Piping is the key to clear, readable R code for data science.

### The right match

Another common R stylistic pitfall is misuing the match() function, either by not using the more readable %in% operator or by using it to join data between different dataframes.

The match() function is very useful. It takes as its input two vectors and returns the indices where matches of the elements in the first vector (if any) are found in the elements of the second vector.

large\_cities <- c("New York", "Los Angeles", "Chicago", "Houston", "Philadelphia", "Phoenix")  
  
warm\_cities <- c("Miami", "New Orleans", "Houston", "Phoenix", "San Diego", "Los Angeles")  
  
match(large\_cities, warm\_cities)

## [1] NA 6 NA 3 NA 4

match(warm\_cities, large\_cities)

## [1] NA NA 4 6 NA 2

In many cases, though, you'll want to use the %in% operator for matching, rather than the match() function itself, because it's aesthetically nicer. But the %in% operator works a little differently, returning a *logical* vector, with TRUE/FALSE as to whether there is a match at all.

large\_cities <- c("New York", "Los Angeles", "Chicago", "Houston", "Philadelphia", "Phoenix")  
  
warm\_cities <- c("Miami", "New Orleans", "Houston", "Phoenix", "San Diego", "Los Angeles")  
  
large\_cities %in% warm\_cities

## [1] FALSE TRUE FALSE TRUE FALSE TRUE

Using the match() function to join data to a dataframe is a mistake I often made in my early days of R coding. Taking two dataframes -- birthdays and cities -- and joining the later data by the name column. Here is the ugly way to do: using match() to subset out the city column in the correct order for the birthdays dataframe:

birthdays <- data.frame(  
 name = c('Steve', 'Laura', 'Kim'),  
 birthday = c('2/17/71', '10/4/83', '6/28/66')  
)  
  
cities <- data.frame(  
 name = c('Laura', 'Kim', 'Steve'),  
 city = c('Chicago', 'Houston', 'Seattle')  
)  
  
# Find match between two dataframes according to name column  
match(birthdays$name, cities$name)

## [1] 3 1 2

# Use that match to join city data to birthdays dataframe  
birthdays$city <- cities[match(birthdays$name, cities$name),]$city  
  
birthdays

## name birthday city  
## 1 Steve 2/17/71 Seattle  
## 2 Laura 10/4/83 Chicago  
## 3 Kim 6/28/66 Houston

Ugh. That was terrible. Don't do this. Why? You can easily flip the order of the two vectors in your match function and it will it return the data sorted incrrectly. But, more importantly, there is a stylically superior way to join data -- using the join-functions from the dplyr package.

Here I'll use the left\_join() function to extract join the city column from cities to the birthdays dataframe. All you must do is specify the by = parameter (which is the column or columns to join by).

library(dplyr)  
  
birthdays <- data.frame(  
 name = c('Steve', 'Laura', 'Kim'),  
 birthday = c('2/17/71', '10/4/83', '6/28/66')  
)  
  
cities <- data.frame(  
 name = c('Laura', 'Kim', 'Steve'),  
 city = c('Chicago', 'Houston', 'Seattle')  
)  
  
birthdays %>%   
 left\_join(cities, by = 'name')

## name birthday city  
## 1 Steve 2/17/71 Seattle  
## 2 Laura 10/4/83 Chicago  
## 3 Kim 6/28/66 Houston

There are other join functions in dplyr -- right\_join(), inner\_join(), outer\_join() and so on. Use those to join data across separate dataframes. Don't use match(). The benefits of using the join-functions are several. There is less risk of a "silent" error, for one. But it's also far easier to interpret.