Enhancing Functional Programming in R

William Jasmine, Anthony Arroyo

2022-10-27

Introduction

This document will attempt to show how to make more efficient certain functional programming processes in R using Tidyverse's `purrr` library. The data that will be used in this case comes from FiveThirtyEight's data set containing predictions for the 2022-2023 NBA season, which can be found on Github. To make the predictions FiveThirtyEight uses a ELO rating methodology, which can be read about in more detail on thier website.

Load and Clean Data

The cell below loads the data from Github and transforms it into a R dataframe.

```r
csv_data <- RCurl::getURL("https://projects.fivethirtyeight.com/nba-model/nba_elo_latest.csv")
dfRaw <- data.frame(read.csv(text=csv_data))
dplyr::glimpse(dfRaw)
```

```
## Rows: 1,230
## Columns: 27
## $ date           <chr> "2022-10-18", "2022-10-18", "2022-10-19", "2022-10-19",~
## $ season         <int> 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2~
## $ neutral        <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ playoff        <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ team1          <chr> "BOS", "GSW", "DET", "IND", "ATL", "MIA", "BRK", "TOR",~
## $ team2          <chr> "PHI", "LAL", "ORL", "WAS", "HOU", "CHI", "NOP", "CLE",~
## $ elo1_pre       <dbl> 1657.640, 1660.620, 1393.525, 1399.202, 1535.408, 1617.~
## $ elo2_pre       <dbl> 1582.247, 1442.352, 1366.089, 1440.077, 1351.165, 1447.~
## $ elo_prob1      <dbl> 0.7329497, 0.8620114, 0.6755904, 0.5842751, 0.8370220, ~
## $ elo_prob2      <dbl> 0.2670503, 0.1379886, 0.3244096, 0.4157249, 0.1629780, ~
## $ elo1_post      <dbl> 1662.199, 1663.449, 1397.249, 1388.883, 1538.164, 1598.~
## $ elo2_post      <dbl> 1577.688, 1439.523, 1362.366, 1450.396, 1348.409, 1466.~
## $ carm.elo1_pre  <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ carm.elo2_pre  <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ carm.elo_prob1 <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ carm.elo_prob2 <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ carm.elo1_post <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ carm.elo2_post <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ raptor1_pre    <dbl> 1693.243, 1615.718, 1308.970, 1462.353, 1618.257, 1649.~
## $ raptor2_pre    <dbl> 1641.877, 1472.174, 1349.865, 1472.018, 1283.328, 1494.~
## $ raptor_prob1   <dbl> 0.6706123, 0.7765022, 0.5632696, 0.5995097, 0.9176509, ~
## $ raptor_prob2   <dbl> 0.32938773, 0.22349780, 0.43673038, 0.40049033, 0.08234~
## $ score1         <int> 126, 123, 113, 107, 117, 108, 108, 108, 115, 115, 102, ~
## $ score2         <int> 117, 109, 109, 114, 107, 116, 130, 105, 112, 108, 129, ~
## $ quality        <int> 96, 67, 3, 37, 24, 76, 80, 86, 80, 34, 37, 79, 92, 42, ~
## $ importance     <int> 13, 20, 1, 28, 1, 19, 44, 40, 25, 4, 34, 32, 19, 49, 28~
## $ total_rating   <int> 55, 44, 2, 33, 13, 48, 62, 63, 53, 19, 36, 56, 56, 46, ~
```

To limit the scope of the data, the following cell converts `df` to only include predictions for games in which the New York Knicks are playing:

```
dfSubset <- dfRaw %>%
  dplyr::filter(team1 == 'NYK' | team2 == 'NYK')
nrow(dfSubset)
```

```
## [1] 82
```

We see now that the data has been limited to only the 82 regular season games that the Knicks will play. Next, the cell below takes the data cleaning one step further by transforming the dataframe to only include stats pertaining to Knicks (as opposed to their opponents):

```
df <- dfSubset %>%
  dplyr::transmute(
    opponent = ifelse(team1=='NYK', team2, team1),
    win_percentage = ifelse(team1=='NYK', elo_prob1, elo_prob2),
    rating_pre = ifelse(team1=='NYK', elo1_pre, elo2_pre),
    quality = quality,
    importantance = importance
  )
dplyr::glimpse(df)
```

```
## Rows: 82
## Columns: 5
## $ opponent       <chr> "MEM", "DET", "ORL", "CHO", "MIL", "CLE", "ATL", "PHI",~
## $ win_percentage <dbl> 0.2567636, 0.7807578, 0.8241579, 0.6008730, 0.2777317, ~
## $ rating_pre     <dbl> 1520.387, 1517.886, 1524.826, 1528.374, 1532.596, 1527.~
## $ quality        <int> 80, 21, 19, 36, 77, 63, 71, 72, 81, 58, 58, 21, 33, 63,~
## $ importantance  <int> 25, 8, 7, 44, 35, 57, 54, 43, 36, 55, 72, 10, 18, 36, 2~
```

The data in its final format only includes the following columns:

1. `opponent` - Who the Knicks are playing against.
2. `win_percentage` - The predicted probability that the Knicks will win.
3. `rating_pre` - The predicted pre-grame ELO rating of the Knicks.
4. `quality` - A relative score of how "fun the game will be to watch", scored from 0-100.
5. `importance` - A relative score of "how important will this game be in getting either team to the post-season", scored from 0-100.

Vectorized Programming

This section will go through an example of how one might use R functions to answer questions about the data, and how those functions can be enhanced using the Tidyverse `purrr` library. The goal in this case: *how does one quickly calculate the mean of each column*?

The old way

The below code chunk shows how we might carry out this task, if we came from a traditional C background:

```
calc_means <- function(df_input) {
  means <- vector("double", length(df_input))
  for (i in seq_along(df_input)) {
    means[i] <- mean(df_input[[i]])
  }
  results <- data.frame(matrix(ncol = ncol(df_input), nrow = 1))
  colnames(results) <- colnames(df_input)
  results[1,] <- means
  results
}

df %>%
  dplyr::select(-opponent) %>%
    calc_means(.)
```

```
##   win_percentage rating_pre  quality importantance
## 1      0.5033847   1511.403 61.15854      58.37805
```

The code above creates a function `calc_means` that calculates and reports the means of each column using a for loop. The results of this are shown above.

The `purrr` way

Using the `purrr` package, running a function over every column in a dataframe becomes incredibly easy via use of the package's `map` functions. There exists a different `map` function for the different R datatypes (i.e. for integers it becomes `map_int`), and each one returns a vector of the datatype specified. In this case, we will use `map_dbl` since calculating means of each column will return a vector of type `double` elements. This is done in the cell below:

```
dplyr::select(df, -opponent) %>%
  purrr::map_dbl(., mean)
```

```
## win_percentage     rating_pre        quality  importantance
##      0.5033847   1511.4031878     61.1585366     58.3780488
```

The Benefit

As can be seen in the code above, finding and reporting the means of each numerical column only takes one line of code using the `purrr` package. The function we are passing over the columns is the `mean` function, and similar descriptive stats functions (such as `sd` and `median`, can be used in the same way):

```
dplyr::select(df, -opponent) %>%
  purrr::map_dbl(., stats::median)
```

```
## win_percentage     rating_pre        quality  importantance
##      0.5368232   1511.0339677     69.0000000     62.0000000
```

```
dplyr::select(df, -opponent) %>%
  purrr::map_dbl(., stats::sd)
```

```
## win_percentage     rating_pre        quality  importantance
##      0.1694184      5.4950101     18.6007231     23.7152034
```

While calling the `calc_means` function we defined earlier now only requires one line since its code has been previously written, instantiating the `map_dbl` function turns out to have a faster runtime:

```
testDf = dplyr::select(df, -opponent)

microbenchmark::microbenchmark(
  c_style = calc_means(testDf),
  purrr = map_dbl(testDf, mean)
)
```

```
## Unit: microseconds
##     expr     min       lq      mean   median      uq     max neval
##  c_style 194.051 197.7440 227.63081 203.6925 243.077 466.872   100
##    purrr  41.026  44.7185  55.00333  54.9750  57.641 123.898   100
```

As can be seen in the output above, using the `purrr` map function saved about four times as much time when compared to using the user-defined `calc_means` function. In the R backend, all functions are vectorized which makes our code more concise, easier to read, less error prone, and alot less redundant.

Conclusion

The `purr` library as part of the larger Tidyverse set of packages provides a couple of great tools to help a user with their functional programming capabilities. Though this document focused solely on the `map` suite of functions, be sure to check others in the documentation that may suit your use case.