afp

Handbook for the Applied Functional Programming R Package

Robert Schnitman

January 1, 2020 (Last Updated: March 17, 2019)

Contents

1 Introduction

The `afp` package–*Applied Functional Programming*–provides functionals to simplify iterative processes in R. The Base R `*apply()` family, `purrr`[1] library, and Julia programming language[2] are the principal influences. The former two contain tools essential for functional programming that minimize the need to incorporate loops and increase code brevity; however, there is inelegance with respect to specific situations.

For example, to map a function and consequently bind rows or columns, `purrr` splits the decision into two functions rather than within one: `map_dfr()` and `map_dfc()`[3], both of which only output to data frames (the former behaves the same as `map_df()`[4])–while this feature is as intended, they nonetheless omit the possibility of a matrix when such a data type is preferred. Additionally, to reduce the results of a mapping, one must encase `Map()`/`lapply()` in `Reduce()`, while Julia blends the routine into `mapreduce()`[5].

As such, `afp` exists to supplement the functionals in Base R, `purrr`, and others to support efficient and concise programming.

This handbook shows how to install `afp`, followed by descriptions of each function in the package before concluding.

2 Installing *afp*

The library `afp` currently is only installable via GitHub and is contingent on R versions at or above 3.5. To install the package, first install `devtools` so that you may make use of the function `install_github`, referencing `afp` by the package creator's username ("robertschnitman") followed by "/afp" as shown in the code below:

```
## afp dependency: R >= 3.5

install.packages("devtools") # Install library necessary for installing afp.

devtools::install_github("robertschnitman/afp") # Install afp.
```

The following sections will assume that you have loaded this library, so please load it so that the codes in the mentioned sections will be executable for you.

```
library(afp)
```

[1]https://purrr.tidyverse.org/
[2]https://julialang.org/
[3]https://purrr.tidyverse.org/reference/map.html
[4]Type `purrr::map_df` in the console and compare with `purrr::map_dfr`.
[5]https://docs.julialang.org/en/v0.6.1/stdlib/collections/#Base.mapreduce-NTuple%7B4,Any%7D

3 *agg()*

The function `agg()` mimics Base R's `aggregate()` with the exception that an unnested data frame is maintained when calling multiple functions in a vector.

To demonstrate, let's compare the structure between the two functions.

3.0.1 aggregate()vs.agg()'

```r
### GOAL: Compare the output between aggregate vs. agg() when
###       calling multiple functions within a vector.

ms   <- function(x) c(m = mean(x), s = sd(x))
form <- formula(cbind(mpg, disp) ~ am + gear)

A <- aggregate(form, mtcars, ms)
B <- agg(form, mtcars, ms)

str(A) # Nested results
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ am  : num  0 0 1 1
##  $ gear: num  3 4 4 5
##  $ mpg : num [1:4, 1:2] 16.11 21.05 26.27 21.38 3.37 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr  "m" "s"
##  $ disp: num [1:4, 1:2] 326.3 155.7 106.7 202.5 94.9 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr  "m" "s"
```

```r
str(B) # Unnested results
```

```
## 'data.frame':    4 obs. of  6 variables:
##  $ am    : num  0 0 1 1
##  $ gear  : num  3 4 4 5
##  $ mpg.m : num  16.1 21.1 26.3 21.4
##  $ mpg.s : num  3.37 3.07 5.41 6.66
##  $ disp.m: num  326 156 107 202
##  $ disp.s: num  94.9 14 37.2 115.5
```

As a result, `aggregate` nests the output into the dependent variables, whereas `agg()` "flattens" the output. The benefit of flattening is that the user can refer to these specific columns more directly than having to call the nested information. In other words, to refer to the mean MPG vector in our example with `aggregate()`, you would have to execute `A$mpg[, 'm']`, whereas in `agg()` it is simply `B$mpg.m`. As such, `agg()` can be more efficient than its counterpart.

4 *bcast()*

Inspired by the dot syntax from Julia's `broadcast()` function[6], `bcast()`/`.()` allows the user to execute `mapply()` with a convenient shorthand.

The function has two required inputs and two optional ones. The required arguments are `f`, the function to call, and `x`, the first argument over which to vectorize. The optional arguments are `...`, which is a list of additional arguments to the function, and `simplify`, which defaults to `FALSE` to maintain a list type in the output–when to set to `TRUE`, the output is an array (typically a matrix a la‘ the `mapply()` function).

4.0.1 Broadcasting with `bcast()`

```
# GOAL: Operate across multiple matrices.

a <- matrix(1:9, 3, 3)
b <- 20:22
c <- matrix(rnorm(9), 3)

bcast(`/`, a, b, simplify = TRUE)   # matrix.
```

```
##            [,1]       [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.05000000 0.10000000 0.1500000 0.2000000 0.2500000 0.3000000 0.3500000
## [2,] 0.04761905 0.09523810 0.1428571 0.1904762 0.2380952 0.2857143 0.3333333
## [3,] 0.04545455 0.09090909 0.1363636 0.1818182 0.2272727 0.2727273 0.3181818
##           [,8]      [,9]
## [1,] 0.4000000 0.4500000
## [2,] 0.3809524 0.4285714
## [3,] 0.3636364 0.4090909
```

```
bcast(`/`, a, b, c, simplify = TRUE)[1:3] # list occurs when lengths do not match.
```

```
## Warning in mapply(f, x, y = dots, SIMPLIFY = simplify): longer argument not a
## multiple of length of shorter

## [[1]]
## [1] 0.05000000 0.04761905 0.04545455
##
## [[2]]
##            [,1]       [,2]     [,3]
## [1,]   1.596784 -0.8670251 3.052809
## [2,] 111.338263 -1.7205692 1.313825
## [3,]   2.735286 -8.5548812 2.527751
##
## [[3]]
## [1] 0.1500000 0.1428571 0.1363636
```

[6] https://docs.julialang.org/en/v0.6.1/manual/arrays/#Broadcasting-1

5 *do.bind()*

When executing `lapply()` to manipulate subsets of data, calling `rbind()` or `cbind()` within `do.call()` is common to fuse the transformed partitions. While the implementation is possible on one line by way of the `do.call(*bind, lapply(x, f))` form, the readability and intended concision decreases as the complexity of the anonymous function increases. Even if the `lapply()` portion is stored in an object before hand, the intention of `do.call()` is not clear until `*bind` is stated. To minimize these issues, `do.bind()`[7] wraps this process and clarifies its purpose: *bind the results of the given function.*

One may obtain similar results with `map_dfr()`/`map_dfc()` from `purrr`; but the output would always result in a data frame rather than the possibility of a matrix. Additionally, the binding rows or columns must be done in different functions, whereas it can be defined within `do.bind()`.

There are three required parameters in this function: `f`, `x`, and `m`–respectively the function, collection (e.g. data frame), and margin (rbind/cbind designation). If `m = 1` (the default), the results are combined row-wise; `2` for column-wise. A fourth parameter `...` passes to `do.call()`. The output is a matrix or dataframe, depending on the inputs being passed.

This function can be useful for storing coefficients from multiple models into a single matrix.

5.0.1 Coefficient Matrix

```
# GOAL: Create a matrix of coefficients stemming from 3 models.

## Split mtcars by gear
split1  <- split(mtcars, mtcars$gear)

## Create a function that excecutes a model for each subset and obtains the coefficients.
adhoc1  <- function(s) {

  coef(lm(mpg ~ disp + wt + am, s))

}

## Execute the ad-hoc function for each subset.
output1 <- do.bind(adhoc1, split1, 1) # == do.call(rbind, lapply(split1, adhoc1)).
output2 <- do.bind(adhoc1, split1, 2) # == do.call(cbind, lapply(split1, adhoc1)).

## Print the outputs.
output1
```

```
##   (Intercept)         disp        wt        am
## 3    27.99461 -0.007982643 -2.384834        NA
## 4    46.68250 -0.097327135 -3.171284 -2.817164
## 5    41.77904 -0.006730729 -7.230952        NA
```

```
output2
```

```
##                        3           4            5
## (Intercept) 27.994609509 46.68249578 41.779042017
## disp        -0.007982643 -0.09732714 -0.006730729
## wt          -2.384834379 -3.17128412 -7.230951906
## am                    NA -2.81716389           NA
```

The outputs above fit well into `kable()` from the `knitr` package:

[7] The naming of `do.bind()` stems from `do.call()`.

```r
knitr::kable(output1)
```

	(Intercept)	disp	wt	am
3	27.99461	-0.0079826	-2.384834	NA
4	46.68250	-0.0973271	-3.171284	-2.817164
5	41.77904	-0.0067307	-7.230952	NA

```r
knitr::kable(output2)
```

	3	4	5
(Intercept)	27.9946095	46.6824958	41.7790420
disp	-0.0079826	-0.0973271	-0.0067307
wt	-2.3848344	-3.1712841	-7.2309519
am	NA	-2.8171639	NA

6 *mapchr()*

`mapchr()` is a general functional for altering character vectors: it applies any function to each of its elements. Similar to `map_chr()` from the `purrr` library with the exception that `mapchr()` only accepts character vectors as the data input. This function is useful for when manipulating the arrangement of the characters is desired.

The required inputs are `f` and `x`, respectively the function and character vector. The output is typically a character vector (depending on the function being passed).

6.0.1 Collapsing

```
rn_mc <- rownames(mtcars)
mapchr(function(x) paste0(x, collapse = '|'), rn_mc)
```

```
##  [1] "M|a|z|d|a| |R|X|4"
##  [2] "M|a|z|d|a| |R|X|4| |W|a|g"
##  [3] "D|a|t|s|u|n| |7|1|0"
##  [4] "H|o|r|n|e|t| |4| |D|r|i|v|e"
##  [5] "H|o|r|n|e|t| |S|p|o|r|t|a|b|o|u|t"
##  [6] "V|a|l|i|a|n|t"
##  [7] "D|u|s|t|e|r| |3|6|0"
##  [8] "M|e|r|c| |2|4|0|D"
##  [9] "M|e|r|c| |2|3|0"
## [10] "M|e|r|c| |2|8|0"
## [11] "M|e|r|c| |2|8|0|C"
## [12] "M|e|r|c| |4|5|0|S|E"
## [13] "M|e|r|c| |4|5|0|S|L"
## [14] "M|e|r|c| |4|5|0|S|L|C"
## [15] "C|a|d|i|l|l|a|c| |F|l|e|e|t|w|o|o|d"
## [16] "L|i|n|c|o|l|n| |C|o|n|t|i|n|e|n|t|a|l"
## [17] "C|h|r|y|s|l|e|r| |I|m|p|e|r|i|a|l"
## [18] "F|i|a|t| |1|2|8"
## [19] "H|o|n|d|a| |C|i|v|i|c"
## [20] "T|o|y|o|t|a| |C|o|r|o|l|l|a"
## [21] "T|o|y|o|t|a| |C|o|r|o|n|a"
## [22] "D|o|d|g|e| |C|h|a|l|l|e|n|g|e|r"
## [23] "A|M|C| |J|a|v|e|l|i|n"
## [24] "C|a|m|a|r|o| |Z|2|8"
## [25] "P|o|n|t|i|a|c| |F|i|r|e|b|i|r|d"
## [26] "F|i|a|t| |X|1|-|9"
## [27] "P|o|r|s|c|h|e| |9|1|4|-|2"
## [28] "L|o|t|u|s| |E|u|r|o|p|a"
## [29] "F|o|r|d| |P|a|n|t|e|r|a| |L"
## [30] "F|e|r|r|a|r|i| |D|i|n|o"
## [31] "M|a|s|e|r|a|t|i| |B|o|r|a"
## [32] "V|o|l|v|o| |1|4|2|E"
```

6.1 *is.lower()* and *is.upper()*

The functions `is.upper()` and `is.lower()` test whether each element in a string vector is all uppercase or lowercase, respectively. The required input for both functions is a character vector. The output is a Boolean vector. These functions are useful for pattern matching acronyms, uppercase, and lowercase elements.

6.1.1 Testing whether an element is all uppercase or lowercase.

```
chr <- c('TEST', 'test', 'tEsT')
is.upper(chr)
```

```
## [1]  TRUE FALSE FALSE
```

```
is.lower(chr)
```

```
## [1] FALSE  TRUE FALSE
```

6.2 *jumble()*

The function `jumble(x)` randomly changes the order of the characters in every element. The input and output is a character vector. This function can be useful for generating variations of existing passwords or other types of string vectors.

6.2.1 Jumbling characters

```
set.seed(0)
rn_mc <- rownames(mtcars)
jumble(rn_mc)
```

```
##  [1] "4dRMa zXa"            "aWzMa4g Ra dX"        "u1Dn0 ast7"
##  [4] "iHnrtD e4 ovre"       "Stp ttobrouraHeon"    "nVlaati"
##  [7] "r3 sDt0eu6"           "42c 0DeMr"            "cM 2re30"
## [10] "80re 2cM"             "8cr e2MC0"            "E0e c54SrM"
## [13] "rLSM c450e"           "Mr4L5S0 Cce"          "doaalwi cdlteolCFe"
## [16] "CiitnLnnetoolnac nl" "irs leelyrphCraIm"    "1Ft 2ai8"
## [19] "aidHic vCno"          "raoCtlaool oyT"       "rCtono oaToya"
## [22] "alrg egdenhDeClo"     "Jv alnMiAeC"          "8a2C aromZ"
## [25] "F iePobrirnacitd"     " 9ait1-FX"            "9-ec1r Pho24s"
## [28] "oEuLtr saopu"         "rePaF ratL ndo"       "ain erFriorD"
## [31] " orisaBMetaar"        "4o 2ovlV1E"
```

6.3 *reverse()*

The function `reverse()` reverses the order of characters for each element. The input and output is a character vector. This function can be useful for decoding reversed strings.

6.3.1 Reversing every element's order.

```
rn_mc <- rownames(mtcars)
reverse(rn_mc)
```

```
##  [1] "4XR adzaM"            "gaW 4XR adzaM"        "017 nustaD"
##  [4] "evirD 4 tenroH"       "tuobatropS tenroH"    "tnailaV"
##  [7] "063 retsuD"           "D042 creM"            "032 creM"
## [10] "082 creM"             "C082 creM"            "ES054 creM"
## [13] "LS054 creM"           "CLS054 creM"          "doowteelF callidaC"
## [16] "latnenitnoC nlocniL" "lairepmI relsyrhC"    "821 taiF"
## [19] "civiC adnoH"          "alloroC atoyoT"       "anoroC atoyoT"
## [22] "regnellahC egdoD"     "nilevaJ CMA"          "82Z oramaC"
## [25] "driberiF caitnoP"     "9-1X taiF"            "2-419 ehcsroP"
## [28] "aporuE sutoL"         "L aretnaP droF"       "oniD irarreF"
## [31] "aroB itaresaM"        "E241 ovloV"
```

7 *mapdims()*

The function `mapdims()` calls `apply()` to map over a dataset's dimensions, saving the column- and row-wise results separately in a list.

The required inputs for these functions are `f` and `x`, respectively the function to execute and the dataset over which to perform the function. The output is a list of arrays (typically a vector or matrix, depending on the function being passed).

7.0.1 Mapping dimensions

```r
mapdims(median, mtcars)
```

```
## $rowwise
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##               4.000               4.000               4.000               3.215
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##               3.440               3.460               4.000               4.000
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##               4.000               4.000               4.000               4.070
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##               3.730               3.780               5.250               5.424
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##               5.345               4.000               4.000               4.000
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##               3.700               3.520               3.435               4.000
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##               3.845               4.000               4.430               4.000
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##               5.000               6.000               8.000               4.000
##
## $colwise
##     mpg     cyl    disp      hp    drat      wt    qsec      vs      am    gear
##  19.200   6.000 196.300 123.000   3.695   3.325  17.710   0.000   0.000   4.000
##    carb
##   2.000
```

7.1 *mapc()* and *mapr()*

To apply a function column-wise in R, `apply(x, 2, f)` can be called–for row-wise results, the margin input (i.e., the second input) can be set to 1. For situational convenience, the functions `mapc()` and `mapr()` achieve the same results, respectively.

The required inputs for these functions are `f` and `x`, respectively the function to execute and the dataset over which to perform the function. The output is an array (typically a vector or matrix, depending on the function being passed).

7.1.1 `mapc/r()`

```r
mapc(median, mtcars) # Column-wise results
```

```
##     mpg     cyl    disp      hp    drat      wt    qsec      vs      am    gear
##  19.200   6.000 196.300 123.000   3.695   3.325  17.710   0.000   0.000   4.000
##    carb
##   2.000
```

```r
mapr(median, mtcars) # Row-wise results.
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##               4.000               4.000               4.000               3.215
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##               3.440               3.460               4.000               4.000
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##               4.000               4.000               4.000               4.070
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##               3.730               3.780               5.250               5.424
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##               5.345               4.000               4.000               4.000
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##               3.700               3.520               3.435               4.000
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##               3.845               4.000               4.430               4.000
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##               5.000               6.000               8.000               4.000
```

8 *mapreduce()*

Base R has functionals that output a list by way of `lapply()` and `Map()` (among others), while `Reduce()` diminishes given elements in a consecutive manner until a single result remains. While these functions are relatively straightforward to combine (depending on the functions being passed), R does not inherently possess a singular function to accomplish this operation unlike Julia with `mapreduce()`[8]. Therefore, `mapreduce()` in `afp` offers a simplified Julia-equivalent to streamline the intended procedure in R.

The three required parameters are `f`, `o`, and `x` ("fox," collectively)–function, (binary) operator, and collection (e.g. matrix). If `f` is multivariate, the fourth parameter `y` can take multiple arguments much like `MoreArgs` in `mapply()`. The final parameter `...` passes to `Reduce()`. The output is typically a matrix, depending on the inputs and functions being passed.

8.0.1 Map and Reduce

```
# GOAL: Map a function to each matrix and then reduce them by a binary operator.

matrixl <- list(A = matrix(c(1:9), 3, 3),
                B = matrix(10:18, 3, 3),
                C = matrix(19:27, 3, 3))

## Univariate case
output1 <- mapreduce(function(x) x^2 + 1, `/`, matrixl)

output1 ## == Reduce(`/`, Map(function(x) x^2 + 1, matrixl))
```

```
##              [,1]         [,2]         [,3]
## [1,] 0.0000547016 0.0002061856 0.0003107868
## [2,] 0.0001022035 0.0002490183 0.0003310752
## [3,] 0.0001560306 0.0002837380 0.0003456270
```

```
## Multivariate case
output2 <- with(matrixl,
                mapreduce(function(i, j, k) i*j - k,
                          `/`,
                          A,
                          list(B, C)))

output2
```

```
##               [,1]          [,2]          [,3]
## [1,] -1.387799e-10 -3.408547e-12 -3.442132e-13
## [2,] -2.925642e-11 -1.456427e-12 -1.839155e-13
## [3,] -9.059628e-12 -6.829299e-13 -1.031438e-13
```

8.1 *mrchop()*

The function `mrchop()` has similar properties to `mapreduce()`: it applies the latter row-wise or column-wise, which can be specified.

The four required parameters are `f`, `o`, `x`, and `m`–function, (binary) operator, collection (e.g. matrix), and margin (either 1 for row-wise or 2 for column-wise). The fifth, optional parameter `...` passes to `mapreduce()`, which passes to `Reduce()`. The output is typically a matrix, depending on the inputs and functions being passed.

[8] https://docs.julialang.org/en/v0.6.1/stdlib/collections/#Base.mapreduce-NTuple%7B4,Any%7D

8.1.1 Map and Reduce Column/Row-wise

```r
mrchop(function(x) x/2, `+`, mtcars, 1) # Row-wise
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##            164.4900            164.8975            129.7900            213.0675
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##            295.1550            192.7700            328.4600            135.4900
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##            149.7850            175.2300            174.8300            255.3700
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##            255.7500            254.9250            364.2800            363.3220
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##            362.8475            106.9250             97.5825            103.4775
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##            136.8875            259.8250            253.0425            323.1400
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##            315.5875            104.1075            136.2850            136.8415
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##            335.3450            189.7950            347.3550            144.4450
```

```r
mrchop(function(x) x/2, `+`, mtcars, 2) # Column-wise
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  321.450   99.000 3691.550 2347.000   57.545   51.476  285.580    7.000
##       am     gear     carb
##    6.500   59.000   45.000
```

8.2 *reducechop()*

The function `reducechop()` has similar properties to `mrchop()`: it applies `Reduce()` row-wise or column-wise, which can be specified.

8.2.1 Reduce Column/Row-wise

```r
reducechop(`+`, mtcars, 1) # Row-wise. Equivalent to Reduce(`+`, mtcars).
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##             328.980             329.795             259.580             426.135
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##             590.310             385.540             656.920             270.980
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##             299.570             350.460             349.660             510.740
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##             511.500             509.850             728.560             726.644
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##             725.695             213.850             195.165             206.955
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##             273.775             519.650             506.085             646.280
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##             631.175             208.215             272.570             273.683
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##             670.690             379.590             694.710             288.890
```

```r
reducechop(`+`, mtcars, 2) # Column-wise (default).
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  642.900  198.000 7383.100 4694.000  115.090  102.952  571.160   14.000
##       am     gear     carb
##   13.000  118.000   90.000
```

9 *mop()*

Essentially, mop is a wrapper for sweep(x, MARGIN, apply(...), FUN). This function is useful for indexing variables by their means, for example, so that the magnitude of a value relative to its average is known.

The four required arguments are `x`, `m`, `s`, and `f`–the collection (e.g. matrix), margin (1 for row-wise or 2 for column-wise), summary statistic function, and binary opertaor function, respectively. A fifth, optional argument `...` passes to `sweep()`. The output is typically a matrix or dataframe, depending on the inputs and functions being passed.

9.0.1 Mopping a Dataframe

```
head(mop(mtcars, 2, mean, `/`)) # == head(sweep(mtcars, 2, apply(mtcars, 2, mean), `/`))
```

```
##                         mpg       cyl      disp        hp      drat        wt
## Mazda RX4         1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8143601
## Mazda RX4 Wag     1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8936203
## Datsun 710        1.1348577 0.6464646 0.4680961 0.6340009 1.0704666 0.7211128
## Hornet 4 Drive    1.0651734 0.9696970 1.1182295 0.7498935 0.8563733 0.9993006
## Hornet Sportabout 0.9307824 1.2929293 1.5603202 1.1930124 0.8758363 1.0692361
## Valiant           0.9009177 0.9696970 0.9752001 0.7158074 0.7673994 1.0754526
##                        qsec       vs       am      gear      carb
## Mazda RX4         0.9221934 0.000000 2.461538 1.0847458 1.4222222
## Mazda RX4 Wag     0.9535682 0.000000 2.461538 1.0847458 1.4222222
## Datsun 710        1.0426500 2.285714 2.461538 1.0847458 0.3555556
## Hornet 4 Drive    1.0891519 2.285714 0.000000 0.8135593 0.3555556
## Hornet Sportabout 0.9535682 0.000000 0.000000 0.8135593 0.7111111
## Valiant           1.1328524 2.285714 0.000000 0.8135593 0.3555556
```

9.1 *mop_div()*

The function mop_div simplifies mop by operating only on columns and assuming f to be the division operator.

Only two parameters are reuqired: the collection `x` and summary statistic function `s`. The output is similar to that of `mop()`.

9.1.1 Indexing a Dataframe

```
head(mop_div(mtcars, mean))
```

```
##                         mpg       cyl      disp        hp      drat        wt
## Mazda RX4         1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8143601
## Mazda RX4 Wag     1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8936203
## Datsun 710        1.1348577 0.6464646 0.4680961 0.6340009 1.0704666 0.7211128
## Hornet 4 Drive    1.0651734 0.9696970 1.1182295 0.7498935 0.8563733 0.9993006
## Hornet Sportabout 0.9307824 1.2929293 1.5603202 1.1930124 0.8758363 1.0692361
## Valiant           0.9009177 0.9696970 0.9752001 0.7158074 0.7673994 1.0754526
##                        qsec       vs       am      gear      carb
## Mazda RX4         0.9221934 0.000000 2.461538 1.0847458 1.4222222
## Mazda RX4 Wag     0.9535682 0.000000 2.461538 1.0847458 1.4222222
## Datsun 710        1.0426500 2.285714 2.461538 1.0847458 0.3555556
## Hornet 4 Drive    1.0891519 2.285714 0.000000 0.8135593 0.3555556
## Hornet Sportabout 0.9535682 0.000000 0.000000 0.8135593 0.7111111
## Valiant           1.1328524 2.285714 0.000000 0.8135593 0.3555556
```

9.2 *smop()*

The function `smop()` simplifies `mop()` by operating only on columns–this is more general than `mop_div()` in which it operates on columns and only uses the division binary operator.

The three required inputs are the collection `x`, summary statistic function `s`, and binary operator function `f`. A fourth, optional input `...` passes to `mop()`, which passes to `sweep()`.

9.2.1 Indexing a Dataframe, Part 2

```
head(smop(mtcars, mean, `/`))
```

```
##                         mpg       cyl      disp        hp      drat        wt
## Mazda RX4         1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8143601
## Mazda RX4 Wag     1.0452636 0.9696970 0.6934756 0.7498935 1.0843688 0.8936203
## Datsun 710        1.1348577 0.6464646 0.4680961 0.6340009 1.0704666 0.7211128
## Hornet 4 Drive    1.0651734 0.9696970 1.1182295 0.7498935 0.8563733 0.9993006
## Hornet Sportabout 0.9307824 1.2929293 1.5603202 1.1930124 0.8758363 1.0692361
## Valiant           0.9009177 0.9696970 0.9752001 0.7158074 0.7673994 1.0754526
##                        qsec       vs       am      gear      carb
## Mazda RX4         0.9221934 0.000000 2.461538 1.0847458 1.4222222
## Mazda RX4 Wag     0.9535682 0.000000 2.461538 1.0847458 1.4222222
## Datsun 710        1.0426500 2.285714 2.461538 1.0847458 0.3555556
## Hornet 4 Drive    1.0891519 2.285714 0.000000 0.8135593 0.3555556
## Hornet Sportabout 0.9535682 0.000000 0.000000 0.8135593 0.7111111
## Valiant           1.1328524 2.285714 0.000000 0.8135593 0.3555556
```

10 *mtapply()*

Being a multivariate version of `tapply()`, `mtapply` applies a function over an array by a list of indices.

The two required inputs are the object `X` and list of indices `INDEX`. The optional inputs are the function to apply `FUN` and `...`, which passses to `mapply`. For the first three inputs, see the documentation for `tapply()`[9] for more information; for the fourth, see `mapply()`[10]. The output is typically a list of vectors or matrices, depending on the inputs and funciton being passed.

10.0.1 Multivariate `tapply()`

```
A <- mtcars[, c('mpg', 'wt', 'disp')]  # Targets.
B <- mtcars[, c('gear', 'am', 'carb')] # Indices.

mtapply(A, B, mean) # Output
```

```
## $mpg_by_gear
##        3        4        5
## 16.10667 24.53333 21.38000
##
## $wt_by_am
##        0        1
## 3.768895 2.411000
##
## $disp_by_carb
##        1        2        3        4        6        8
## 134.2714 208.1600 275.8000 308.8200 145.0000 301.0000
```

[9]https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/tapply
[10]https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/mapply

11 *pairbind()* and *pairbind_df()*

The function `pairbind()` appends two datasets' rows in a pairwise fashion. In other words, it appends the 1st row of the first dataset with the 1st row of the second dataset, the 2nd row of the first with the 2nd row of the second, and so on. This function can be useful for kable-friendly frequency distribution tables.

The two required inputs are `x` and `y`, each being a 2D object (e.g. dataframe, matrix, etc). The output is a matrix.

The function `pairbind_df()` accomplishes the same as `pairbind()` except that the output is a data frame.

11.0.1 Creating a kable-friendly Frequency Distribution Table

```r
library(tidyverse) # For Data management
```

```
## Warning: package 'tidyverse' was built under R version 3.6.2
```

```
## Warning: package 'ggplot2' was built under R version 3.6.2
```

```r
library(magrittr)  # For the update assignment pipe operator %<>%
library(knitr)     # For kable().
library(kableExtra)# For other kable functions.

# Frequencies
freq_df <- with(diamonds, table(color, clarity)) %>%
  as.data.frame() %>%
  spread(NCOL(.) - 1, NCOL(.)) %>%
  mutate(Total = apply(.[, 2:NCOL(.)], 1, sum))

# Percentages
 prop_df <- freq_df
 prop_df[, 2:NCOL(prop_df)] <- prop_df[, 2:NCOL(prop_df)]/prop_df$Total
 prop_df[, 2:NCOL(prop_df)] %<>%
   map_df(~ paste0(round(.x*100), '%'))
  freq_df %<>%
  format(., big.mark = ',', scientific = FALSE) %>%
  map_df(as.character)

# Output
prop_df %<>% map_df(as.character)

pb <- pairbind_df(freq_df, prop_df)

pb %>%
  kable(booktabs = TRUE) %>%
  kable_styling(full_width = TRUE) %>%
  collapse_rows(1, valign = 'top', latex_hline = 'major')
```

color	I1	SI2	SI1	VS2	VS1	VVS2	VVS1	IF	Total
D	42 1%	1,370 20%	2,083 31%	1,697 25%	705 10%	553 8%	252 4%	73 1%	6,775 100%
E	102 1%	1,713 17%	2,426 25%	2,470 25%	1,281 13%	991 10%	656 7%	158 2%	9,797 100%
F	143 1%	1,609 17%	2,131 22%	2,201 23%	1,364 14%	975 10%	734 8%	385 4%	9,542 100%
G	150 1%	1,548 14%	1,976 17%	2,347 21%	2,148 19%	1,443 13%	999 9%	681 6%	11,292 100%
H	162 2%	1,563 19%	2,275 27%	1,643 20%	1,169 14%	608 7%	585 7%	299 4%	8,304 100%
I	92 2%	912 17%	1,424 26%	1,169 22%	962 18%	365 7%	355 7%	143 3%	5,422 100%
J	50 2%	479 17%	750 27%	731 26%	542 19%	131 5%	74 3%	51 2%	2,808 100%

12 *telecast()*

`Map()`/`mapply()` from Base R executes functions pairwise when given multiple data objects, as do `map2()`/`pmap()` from `purrr`. While beneficial in their own right, said functions cannot concisely map over datasets *independently* of each other, which would be useful for storing disparate information into a single list.

Inspired by `broadcast()` from Julia[11], `telecast()` essentially wraps `mapply()` within `lapply()` to achieve this outcome.

The two required functions are `f` and `l`, respectively a function and list. The third parameter `as.vector`, which is optional, converts the output to a vector if set to `TRUE` (and thus will resemble the output from `rapply()`); by default, it is `FALSE` for a list format.

12.0.1 Iterative Means

```
# GOAL: Obtain the means for each column in 3 datasets.

### Create a list of 3 datasets independent of each other.
l <- list(mc = mtcars, aq = airquality, lcs = LifeCycleSavings)

### Create a function that removes missing values from calculating the average.
mean.nr <- function(x) mean(x, na.rm = TRUE) # airquality has NA values.

### Get the means for every column column in each dataset.
output1 <- telecast(mean.nr, l)

output1 # Compare: lapply(l, function(x) mapply(mean.nr, x))
```

```
## $mc
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
##
## $aq
##      Ozone    Solar.R       Wind       Temp      Month        Day
##  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
##
## $lcs
##        sr     pop15     pop75       dpi      ddpi
##    9.6710   35.0896    2.2930 1106.7584    3.7576
```

12.1 *chain()*

The function `chain()` is a simplification of `telecast()`: the output is a matrix via `sapply()`.

12.1.1 Chaining Means into a Bar Plot

```
# GOAL 1: Get means for every column in mtcars by cylinder.

l      <- split(mtcars, mtcars$cyl)
output <- chain(mean, l)
output
```

[11] https://docs.julialang.org/en/v0.6.1/manual/arrays/#Broadcasting-1

```
##        mpg cyl     disp        hp     drat       wt     qsec        vs
## 4 26.66364   4 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909
## 6 19.74286   6 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286
## 8 15.10000   8 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000
##          am     gear     carb
## 4 0.7272727 4.090909 1.545455
## 6 0.4285714 3.857143 3.428571
## 8 0.1428571 3.285714 3.500000
```

```r
all(chain(mean, l) == t(sapply(l, function(z) sapply(z, mean))))
```

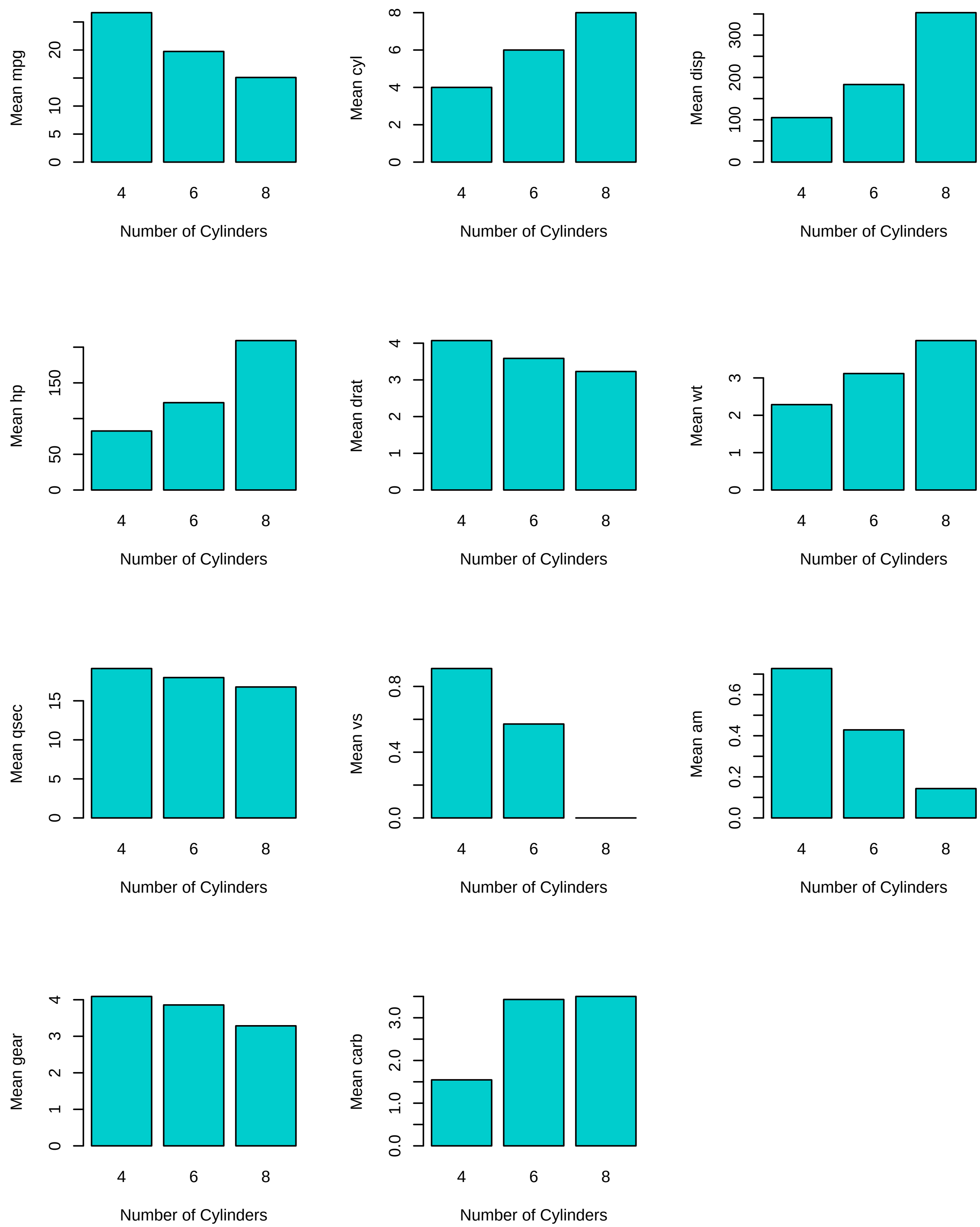
```
## [1] TRUE
```

```r
# GOAL 2: Plot the means
par (mfrow = c(4, 3))

for (i in 1:NCOL(output)) {

  barplot(output[, i],
          col  = 'cyan3',
          xlab = 'Number of Cylinders',
          ylab = paste('Mean', colnames(output)[i]))

}
```


Mean mpg
Mean cyl
Mean disp
Mean hp
Mean drat
Mean wt
Mean qsec
Mean vs
Mean am
Mean gear
Mean carb
Number of Cylinders

13 Conclusion

The functions discussed and demonstrated will be improved on a continuous basis to (1) minimize repetitive iterative processing and (2) emphasize code efficiency and brevity.

References

Julia programming language. https://julialang.org/

Julia - `broadcast()`. https://docs.julialang.org/en/v0.6.1/manual/arrays/#Broadcasting-1

Julia - `mapreduce()`. https://docs.julialang.org/en/v0.6.1/stdlib/collections/#Base.mapreduce-NTuple%7B4,Any%7D

R Documentation. `tapply()`. https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/tapply

R Documentation. `mapply()`. https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/mapply

Tidyverse - `purrr`. https://purrr.tidyverse.org/

See also

`afp` GitHub Page. https://github.com/robertschnitman/afp

`afpj`, a Julia-equivalent library of `afp`. https://github.com/robertschnitman/afpj

Wickham, Hadley. *Advanced R*, *Functionals* chapter. http://adv-r.had.co.nz/Functionals.html