Introduction

Welcome to my series of blog posts about my data manipulation package, {poorman}. For those of you that don't know, {poorman} is aiming to be a replication of {dplyr} but using only {base} R, and therefore be completely dependency free. What's nice about this series is that if you would rather just use {dplyr}, then that's absolutely OK! By highlighting {poorman} functionality, this series of blog posts simultaneously highlights {dplyr} functionality too! However I sometimes also describe how I developed the internals of {poorman}, often highlighting useful {base} R tips and tricks.

Since my last blog post about {poorman}, versions 0.2.2 and 0.2.3 have been released, bringing with them a whole host of new functions and features. In today's blog post we will be taking a look at some of these new features. Given the sheer amount of features this release brings, we won't be focusing on the internals of any of these functions. Instead, we will simply be taking a look at what some of them can do.

`across()`

One of the newer features in {dplyr}, `across()` is intended to eventually replace the scoped variants (`_if`, `_at`, `_all`) of the "single table" verb functions which have now been superseded. These functions will supposedly remain within {dplyr} for "several years" still, giving developers plenty of time to update their code.

`across()` makes it easy to apply the same transformation to multiple columns, allowing you to use `poor-select` (or `tidy-select`) semantics inside of `summarise()` and `mutate()`. Let's take a look at the function in action.

```
library(poorman, warn.conflicts = FALSE)
iris %>%
  group_by(Species) %>%
  summarise(across(.cols = starts_with("Sepal"), .fn = mean))
#      Species Sepal.Length Sepal.Width
# 1     setosa        5.006       3.428
# 2 versicolor        5.936       2.770
# 3  virginica        6.588       2.974
```

In the above code chunk, we take the iris dataset and group it by the `Species` column; then we look to summarise across all columns which start with the string `"Sepal"` (`Sepal.Length` and `Sepal.Width`) by taking the mean of those columns within each `Species` group. Let's take a look at a more complex example.

```
iris %>%
  group_by(Species) %>%
  summarise(across(.cols = contains("Width"), .fn = list(mean, sd)))
#      Species Sepal.Width_1 Sepal.Width_2 Petal.Width_1 Petal.Width_2
# 1     setosa         3.428     0.3790644         0.246     0.1053856
# 2 versicolor         2.770     0.3137983         1.326     0.1977527
# 3  virginica         2.974     0.3224966         2.026     0.2746501
```

So here, we are saying give me the mean and standard devitaion across all columns containing the string `"Width"` for each `Species` of iris flower. Notice how the output is named, the

function will give the columns numbers to represent the functional output, i.e. here `_1` represents the mean and `_2` represents the standard deviation. You can control the names yourself but providing them to the `.names` argument.

```
iris %>%
  group_by(Species) %>%
  summarise(across(
    .cols = contains("Width"),
    .fn = list(mean, sd),
    .names = c(
      "sepal_width_mean", "sepal_width_sd", "petal_width_mean",
"petal_width_sd"
    )
  ))
#      Species sepal_width_mean sepal_width_sd petal_width_mean
petal_width_sd
# 1     setosa            3.428      0.3790644            0.246
0.1053856
# 2 versicolor            2.770      0.3137983            1.326
0.1977527
# 3  virginica            2.974      0.3224966            2.026
0.2746501
```

This is slightly different to how {dplyr} works since it imports {glue}, but remember, {poorman} aims to be dependency free. This functionality will be expanded upon in future releases of {poorman}.

case_when()

This function allows you to vectorise multiple `if_else()` statements. It is an R equivalent of the SQL `CASE WHEN` statement. If no cases match, `NA` is returned. The syntax for the function is a sequence of two-sided formulas. The left hand side determines which values match the particular case whereas the right hand side provides the replacement value.

```
x <- 1:50
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)
#  [1] "1"         "2"         "3"         "4"         "fizz"      "6"
"buzz"
#  [8] "8"         "9"         "fizz"      "11"        "12"        "13"
"buzz"
# [15] "fizz"      "16"        "17"        "18"        "19"
"fizz"      "buzz"
# [22] "22"        "23"        "24"        "fizz"      "26"        "27"
"buzz"
# [29] "29"        "fizz"      "31"        "32"        "33"        "34"
"fizz buzz"
# [36] "36"        "37"        "38"        "39"        "fizz"      "41"
```

```
"buzz"
# [43] "43"        "44"        "fizz"      "46"        "47"        "48"
"buzz"
# [50] "fizz"
```

Like an if statement, the arguments are evaluated in order, so you must proceed from the most specific to the most general. `case_when()` is particularly useful inside `mutate()` when you want to create a new variable that relies on a complex combination of existing variables.

```
mtcars %>%
  mutate(efficient = case_when(mpg > 25 ~ TRUE, TRUE ~ FALSE))
#                     mpg cyl  disp  hp drat    wt  qsec vs am gear
carb efficient
# Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4
4     FALSE
# Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4
4     FALSE
# Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4
1     FALSE
# Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3
1     FALSE
# Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3
2     FALSE
# Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3
1     FALSE
# Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3
4     FALSE
# Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4
2     FALSE
# Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4
2     FALSE
# Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4
4     FALSE
# Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4
4     FALSE
# Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3
3     FALSE
# Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3
3     FALSE
# Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3
3     FALSE
# Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3
4     FALSE
# Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3
4     FALSE
# Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3
4     FALSE
# Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4
1      TRUE
# Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4
2      TRUE
# Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4
```

```
1      TRUE
# Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3
1     FALSE
# Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3
2     FALSE
# AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3
2     FALSE
# Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3
4     FALSE
# Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3
2     FALSE
# Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4
1      TRUE
# Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5
2      TRUE
# Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5
2      TRUE
# Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5
4     FALSE
# Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5
6     FALSE
# Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5
8     FALSE
# Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4
2     FALSE
```

rename_with()

`rename_with()` acts like `rename()`, only it allows you to rename columns with a function. In the below example, we rename the columns of `iris` to be upper case.

```
rename_with(iris, toupper) %>% head()
#   SEPAL.LENGTH SEPAL.WIDTH PETAL.LENGTH PETAL.WIDTH SPECIES
# 1          5.1         3.5          1.4         0.2  setosa
# 2          4.9         3.0          1.4         0.2  setosa
# 3          4.7         3.2          1.3         0.2  setosa
# 4          4.6         3.1          1.5         0.2  setosa
# 5          5.0         3.6          1.4         0.2  setosa
# 6          5.4         3.9          1.7         0.4  setosa
```

However we can have more control over which columns we rename by making use of the `.cols` parameter and `poor-select` selection semantics.

```
rename_with(iris, toupper, contains("Petal")) %>% head()
#   Sepal.Length Sepal.Width PETAL.LENGTH PETAL.WIDTH Species
# 1          5.1         3.5          1.4         0.2  setosa
# 2          4.9         3.0          1.4         0.2  setosa
# 3          4.7         3.2          1.3         0.2  setosa
# 4          4.6         3.1          1.5         0.2  setosa
# 5          5.0         3.6          1.4         0.2  setosa
# 6          5.4         3.9          1.7         0.4  setosa
```

Conclusion

This post has demonstrated some of the capabilities of the {poorman} (and therefore {dplyr}) package. The v0.2.2 and v0.2.3 releases actually includes plenty more features and functions so be sure to check out …