## Introduction

Hello all and welcome to another edition of the `poorman` series of blog posts. In this series I am discussing my progress in writing a `base` R equivalent of `dplyr`. What's nice about this series is that if you're not into `poorman` and would prefer just to use `dplyr`, then that's absolutely OK! By highlighting `poorman` functionality, this series of blog posts simultaneously highlights `dplyr` functionality too!

Today I want to showcase some column selection helper features from the `tidyselect` package - often used in conjunction with `dplyr` - which I have finished now replicated within `poorman`, of course using `base` only. I'll also discuss a little bit about what is happening in the background of `poorman`'s development with regards to testing.

## Select Helpers

The first official release version of `poorman` (v 0.1.9) was the first version that I considered to contain all of the "core" functionality of `dplyr`; everything from `select()` to `group_by()` and `summarise()`. Now that that functionality is nailed down, it gives me time to focus of some of the smaller features of `dplyr` and the wider `tidyverse` and so over the last couple of weeks I have been working on adding the `tidyselect::select_helpers` to `poorman`. For those that are unaware, `select_helpers` are a collection of functions that help the user to select variables based on their names. For example you may wish to select all columns which start with a certain prefix or maybe select columns matching a particular regular expression. Let's take a look at some examples.

## Selecting Columns Based On Partial Column Names

If your data contain lots of columns whose names share a similar structure, you can use partial matching by adding `starts_with()`, `ends_with()` or `contains()` in your `select()`/`relocate()` statement.

```
library(poorman, warn.conflicts = FALSE)
iris %>% select(starts_with("Petal"), ends_with("Width")) %>% head()
#   Petal.Length Petal.Width Sepal.Width
# 1          1.4         0.2         3.5
# 2          1.4         0.2         3.0
# 3          1.3         0.2         3.2
# 4          1.5         0.2         3.1
# 5          1.4         0.2         3.6
# 6          1.7         0.4         3.9
```

## Reordering Columns

The columns of the `iris` dataset come in the following order.

```
colnames(iris)
```

```
# [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

But what if we wanted all of the “Width” columns at the start of this `data.frame`? There are a couple of ways in which we can achieve this. Firstly, we can use `select()` in combination with the select helper `everything()`.

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

`poorman` here first selects the columns “Petal.Width” and “Sepal.Width” before selecting everything else. This is great, but if your data contain a lot of columns containing “Width” then you will have to write out a lot of column names. Well this is where we can use `relocate()` and the select helper `contains()` to move those columns to the start of `iris`.

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

By default, `relocate()` will move all selected columns to the start of the `data.frame`. You can adjust this behaviour with the `.before` and `.after` parameters. Let’s move the “Petal” columns to appear after the “Species” column.

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

## Select Columns Using a Regular Expression

The previous helper functions work with exact pattern matches. Let’s say you have similar patterns within your column names that are not quite the same, you can use regular expressions with the `matches()` helper function to identify them. Here I will use the `mtcars` dataset and look to extract all columns which start with a “w” or a “d” and end with a “t”.

```
mtcars %>% select(matches("^[wd].*[t]$")) %>% head()
#                   drat    wt
# Mazda RX4         3.90 2.620
# Mazda RX4 Wag     3.90 2.875
# Datsun 710        3.85 2.320
# Hornet 4 Drive    3.08 3.215
# Hornet Sportabout 3.15 3.440
# Valiant           2.76 3.460
```

## The Select Helper List

We have seen a few examples of select helpers now available in `poorman`. There are more, however, and the following list details each of them. Remember that these functions can be used to help users `select()` and `relocate()` columns within `data.frame`s.

- `starts_with()`: Starts with a prefix.
- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like `x01`, `x02`, `x03`.
- `all_of()`: Matches variable names in a character vector. All names must be present, otherwise an out-of-bounds error is thrown.
- `any_of()`: Same as `all_of()`, except that no error is thrown for names that don't exist.
- `everything()`: Matches all variables.
- `last_col()`: Select last variable, possibly with an offset.

## Docker

There was a request on Twitter to put together a Docker image for `poorman`. This has now been done and can be seen on Dockerhub. This means if you have Docker installed, you can run a containerised version of `poorman` easily with the following line of code.

```
docker run --rm -it nathaneastwood/poorman
```

## Test, Test, Test!

Since the last release of `poorman` (v 0.1.9) to CRAN, I have been working on a few bugs that I and other users of the package had identified. I'm happy to say that these have now been squashed and the issues list is looking very empty. As a brief overview, the following problems are now fixed:

- `mutate()` column creations are immediately available, e.g. `mtcars %>% mutate(mpg2 = mpg * 2, mpg4 = mpg2 * 2)` will create columns named `mpg2` and `mpg4`
- `group_by()` groups now persist in selections, e.g. `mtcars %>% group_by(am) %>% select(mpg)` will

return `am` and `mpg` columns

- `slice()` now duplicates rows, e.g. `mtcars %>% slice(2, 2, 2)` will return row 2 three times
- `summarize()` is now exported

`dplyr` is a very well known and extremely well developed package. In order for `poorman` to have any credibility, it needs to work correctly. Therefore a large amount of effort and energy has gone into testing `poorman` to ensure it produces the results one would expect. Since adding all of the new features and bug fixes described in this blog, `poorman` has surpassed 100 tests!

```
tinytest::test_all()
# Running test_arrange.R..............    5 tests OK
# Running test_filter.R...............    6 tests OK
# Running test_groups.R...............    5 tests OK
# Running test_joins_filter.R.........    4 tests OK
# Running test_joins.R................    7 tests OK
# Running test_mutate.R...............    6 tests OK
# Running test_pull.R.................    6 tests OK
# Running test_relocate.R.............    6 tests OK
# Running test_rename.R...............    4 tests OK
# Running test_rownames.R.............    2 tests OK
# Running test_select_helpers.R.......   25 tests OK
# Running test_select.R...............   13 tests OK
# Running test_slice.R................    5 tests OK
# Running test_summarise.R............    6 tests OK
# Running test_transmute.R............    3 tests OK
# Running test_utils.R................    1 tests OK
# [1] "All ok, 104 results"
```

This also means that the package coverage is extremely high.

```
covr::package_coverage()
# poorman Coverage: 97.87%
# R/utils.R: 80.00%
# R/group.R: 90.91%
# R/joins.R: 92.86%
# R/arrange.R: 100.00%
# R/filter.R: 100.00%
# R/init.R: 100.00%
# R/joins_filtering.R: 100.00%
# R/mutate.R: 100.00%
# R/pipe.R: 100.00%
# R/pull.R: 100.00%
# R/relocate.R: 100.00%
# R/rename.R: 100.00%
```

```
# R/rownames.R: 100.00%
# R/select_helpers.R: 100.00%
# R/select.R: 100.00%
# R/slice.R: 100.00%
# R/summarise.R: 100.00%
# R/transmute.R: 100.00%
```

I hope this gives users of `poorman` that extra confidence when using the package. I have now submitted this updated version of `poorman` to CRAN and I am just waiting on their feedback so hopefully in the coming days it will be available. Watch this space!