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.

Today marks the release of v0.2.1 of `{poorman}` and with it 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; the internals will be saved for another post. In stead, we will simply be taking a look at what some of them can do.

Selecting Distinct Rows

The first function we will take a look at is `distinct()`. Let's say you want to select only the distinct, or unique, rows from your `data.frame`, `distinct()` will help you do that. Let's create some fake data; some are duplicated.

```
df <- data.frame(
  id = c(1, 2, 3, 4, 5, 6, 1, 2, 7, 1, 4, 6),
  age = c(26, 24, 26, 22, 23, 24, 26, 24, 22, 26, 22, 25),
  score = c(85, 63, 55, 74, 31, 77, 85, 63, 42, 85, 74, 78)
)
df
#    id age  score
# 1   1  26    85
# 2   2  24    63
# 3   3  26    55
# 4   4  22    74
# 5   5  23    31
# 6   6  24    77
# 7   1  26    85
# 8   2  24    63
# 9   7  22    42
# 10  1  26    85
# 11  4  22    74
# 12  6  25    78
```

Now we wish to see the distinct records from this data.

```
library(poorman, warn.conflicts = FALSE)
df %>% distinct()
#    id age  score
# 1   1  26    85
# 2   2  24    63
# 3   3  26    55
# 4   4  22    74
# 5   5  23    31
# 6   6  24    77
# 9   7  22    42
# 12  6  25    78
```

So we see that we now only have 8 records out of the original 12 because the duplicates have been removed. We can actually obtain the distinct rows for a particular column, returning just that column.

```
df %>% distinct(age)
#    age
# 1   26
# 2   24
# 4   22
# 5   23
# 12  25
```

But if you need the other variables still, you can choose to keep those too.

```
df %>% distinct(age, .keep_all = TRUE)
#    id age score
# 1   1  26    85
# 2   2  24    63
# 4   4  22    74
# 5   5  23    31
# 12  6  25    78
```

Slicing Data

`{dplyr}` provides a couple of ways to selecting a subset of rows. It has the functions `top_n()` and `top_frac()` as well as the `slice_*()` family of functions. The former functions have now been superseded by the latter and so `{poorman}` skipped the implementation of the former. So what exactly do they do? Let's take a look at some examples using the `mtcars` dataset.

`slice_head()` returns the first `n` rows (defaults to 1). `slice_tail()` returns the *last* `n` rows (not shown here).

```
slice_head(mtcars, n = 3)
#                mpg cyl disp  hp drat    wt  qsec vs am gear carb
# Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
# Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
# Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

`slice_sample()` randomly selects rows with or without replacement.

```
slice_sample(mtcars, n = 3, replace = TRUE)
#                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
# Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
# Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
# Merc 280C     17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
```

`slice_min()` and `slice_max()` select rows with highest or lowest values of a variable.

```
mtcars %>% slice_min(mpg, n = 3)
#                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
# Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
# Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
# Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

Selecting With Predicates

It is now possible to select columns in your `data.frame` which match a predicate such as `is.numeric()`. `where()` takes a function and returns all variables for which the function returns `TRUE`.

```
df <- data.frame(
  col1 = c(1, 2, 3),
  col2 = c("x", "y", "z"),
  col3 = c(TRUE, FALSE, TRUE)
)
```

```
df %>% select(where(is.numeric))
#   col1
# 1    1
# 2    2
# 3    3
```

Working With NA Values

Finding the First Non-Missing Element

Given a set of vectors, the `coalesce()` function finds the first non-missing value at each position.

```
# Use a single value to replace all missing values
x <- sample(c(1:5, NA, NA, NA))
coalesce(x, 0L)
# [1] 4 0 5 0 1 2 0 3

# Or match together a complete vector from missing pieces
y <- c(1, 2, NA, NA, 5)
z <- c(NA, NA, 3, 4, 5)
coalesce(y, z)
# [1] 1 2 3 4 5
```

Convert Values To NA

We can convert values in a vector `x` if they match values in a second vector `y`.

```
na_if(1:5, 5:1)
# [1]  1  2 NA  4  5
```

This is particularly useful in a `data.frame` if you need to replace a particular value.

```
df <- data.frame(a = c("a", "b", "c", "BAD_VALUE"))
df %>% mutate(a = na_if(a, "BAD_VALUE"))
#      a
# 1    a
# 2    b
# 3    c
# 4
```

Replacing NA Values

Within a `data.frame` we often have missing values in multiple columns. We sometimes wish to replace these values which is where `replace_na()` comes in. `replace_na()` is actually a function from the `{tidyr}` package but I decided to add it to `{poorman}` as it is extremely useful. Let's take a look.

```
df <- data.frame(x = c(1, 2, NA), y = c("a", NA, "b"))
df %>% replace_na(list(x = 0, y = "unknown"))
#   x       y
# 1 1       a
# 2 2 unknown
# 3 0       b
```

Recoding Values

If we wish to replace values within a vector or a column of a `data.frame`, we can use `recode()`. This is a vectorised version of `base::switch()`: you can replace numeric values based on their position or their name, and character or factor values only by their name.

```
char_vec <- sample(c("a", "b", "c"), 10, replace = TRUE)
recode(char_vec, a = "Apple")
#  [1] "b"     "c"     "b"     "c"     "Apple" "Apple" "Apple" "c"     "Apple"
"b"
recode(char_vec, a = "Apple", b = "Banana")
#  [1] "Banana" "c"      "Banana" "c"      "Apple"  "Apple"  "Apple"  "c"
"Apple"
# [10] "Banana"
```

Group Details

The final group (no pun intended) of features are focussed solely on grouped data. Given how many there are, I am not going to go into detail and instead I provide a brief overview here for the reader. The plan is to detail these functions in a separate blog post since a lot of work went on under the hood that may be interesting to discuss.

- Functions for splitting `data.frame`s: `group_split()`, `group_keys()`
- Extract grouping metadata: `group_data()`, `group_indices()`, `group_vars()`, `group_rows()`, `group_size()`, `n_groups()`, `groups()`
- Extract information about the *current* group: `cur_data()`, `cur_group()`, `cur_group_id()`, `cur_group_rows()`, `cur_column()`

Conclusion

You made it this far, great! I won't keep you much longer.…