This journey started almost exactly a year ago, but it’s finally been sufficiently worked through and merged! Yay, to the tidyverse (minor as it may be).

It began with a tweet, recalling a surprise I encountered that day during some routine data processing

Source of today's mild heart-attack: I have categories W, X\_Y, and Z in some data. Intending to keep only the second two:

data %>% filter(g %in% c("X Y", "Z")

For those of you not so comfortable with pipes and dplyr, I was trying to subset a data.frame ‘data‘ (with a column g having values "W", "X\_Y" and "Z") to only those rows for which the column g had the value "X\_Y" or "Z" (not the actual values, of course, but that’s the idea). Without dplyr this might simply be

data[data$g %in% c("X Y", "Z"), ]

To make that more concrete, let’s actually show it in action

data <- data.frame(a = 1:5, g = c("X\_Y", "W", "Z", "Z", "W"))

data

#> a g

#> 1 1 X\_Y

#> 2 2 W

#> 3 3 Z

#> 4 4 Z

#> 5 5 W

data %>%

filter(g %in% c("X Y", "Z"))

#> a g

#> 1 3 Z

#> 2 4 Z

filter isn’t at fault here — the same issue would arise with [ — I have mis-specified the values I wish to match, so I am returned only the matching values. %in% is also performing its job – it returns a logical vector; the result of comparing the values in the column g to the vector c("X Y", "Z"). Both of these functions are behaving as they should, but the logic of what I was trying to achieve (subset to only these values) was lost.

Now, in some instances, that is exactly the behaviour you want — subset this vector to *any* of these values… where those values may not be present in the vector to begin with

data %>%

filter(values %in% all\_known\_values)

The problem, for me, is that there isn’t a way to say “all of these should be there”. The lack of matching happens silently. If you make a typo, you don’t get that level, and you aren’t told that it’s been skipped

simpsons\_characters %>%

filter(first\_name %in% c("Homer", "Marge", "Bert", "Lisa", "Maggie")

Technically this is a double-post because I also want to sidenote this with something I am amazed I have not known about yet (I was approximately today years old when I learned about this)… I’ve used regexmatching for a while occasionally. I’m familiar with counting patterns ((A){2} to match two occurrences of A) and ranges of counts ((A){2,4} to match between two and four occurrences of A) but I was not aware that you can specify a number of ***mistakes*** that can be included to still make a match…

grep("Bart", c("Bart", "Bort", "Brat"), value = TRUE)

#> [1] "Bart"

grep("(Bart){~1}", c("Bart", "Bort", "Brat"), value = TRUE)

#> [1] "Bart" "Bort"

(“Are you matching to me?”… “No, my regex *also matches to ‘Bort’”)*

Use (pattern){~n}to allow up to nsubstitutions in the pattern matching.

Back to the original problem — filterand %in%are doing their jobs, but we aren’t getting the result we want because we made a typo, and we aren’t told that we’ve done so.

Enter forcats(originally to dplyr, but forcatsdoes make more sense) which implements fct\_match(f, lvls). This checks that all of the values in lvlsare actually present in fbefore returning the logical vector of which entries they correspond to. With this, the pattern becomes :

data %>%

filter(fct\_match(g, c("X Y", "Z")))

#> Error in filter\_impl(.data, quo): Evaluation error: Levels not present in factor: "X Y".

Yay! We’re notified that we’ve made an error. "X Y"isn’t actually in our column g. If we don’t make the error, we get the result we actually wanted in the first place. We can now use this successfully

data %>%

filter(fct\_match(g, c("X\_Y", "Z")))

#> a g

#> 1 1 X\_Y

#> 2 3 Z

#> 3 4 Z

It took a while for the PR to be addressed (the tidyverse crew have plenty of backlog, no doubt) but after some minor requested changes and a very neat cleanup by Hadley himself, it’s been merged.

My original version had a few bells and whistles that the current implementation has put aside. The first was inverting the matching with fct\_excludeto make it easier to negate the matching without having to create a new anonymous function, i.e. ~!fct\_match(.x). I find this particularly useful since a pipe expects a call/named function, not a lambda/anonymous function, which is actually quite painful to construct

data %>%

pull(g) %>%

(function(x) !fct\_match(x, c("X\_Y", "Z")))

#> [1] FALSE TRUE FALSE FALSE TRUE

whereas if we defined

fct\_exclude <- function(f, lvls, ...) !fct\_match(f, lvls, ...)

we can use

data %>%

pull(g) %>%

fct\_exclude(c("X\_Y", "Z"))

#> [1] FALSE TRUE FALSE FALSE TRUE

The other was specifying whether or not to include missing levels when considering if lvls is a valid value in f since unique(f) and levels(f) can return different answers.

The cleanup really made me think about how much ‘fluff’ some of my code can have. Sure, it’s nice to encapsulate some logic in a small additional function, but sometimes you can actually replace all of that with a one-liner and not need all that. If you’re ever in the mood to see how compact internal code can really be, check out the source of forcats.

Hopefully this pattern of filter(fct\_match(f, lvls)) is useful to others. It’s certainly going to save me overlooking some typos.