Column-wise operations

It's often useful to perform the same operation on multiple columns, but copying and pasting is both tedious and error prone:

```
df %>%
  group_by(g1, g2) %>%
  summarise(a = mean(a), b = mean(b), c = mean(c), d = mean(d))
```

(If you're trying to compute `mean(a, b, c, d)` for each row, instead see `vignette("rowwise")`)

This vignette will introduce you to the `across()` function, which lets you rewrite the previous code more succinctly:

```
df %>%
  group_by(g1, g2) %>%
  summarise(across(a:d, mean))
```

We'll start by discussing the basic usage of `across()`, particularly as it applies to `summarise()`, and show how to use it with multiple functions. We'll then show a few uses with other verbs. We'll finish off with a bit of history, showing why we prefer `across()` to our last approach (the `_if()`, `_at()` and `_all()` functions) and how to translate your old code to the new syntax.

```
library(dplyr, warn.conflicts = FALSE)
```

Basic usage

`across()` has two primary arguments:

- The first argument, `.cols`, selects the columns you want to operate on. It uses tidy selection (like `select()`) so you can pick variables by position, name, and type.
- The second argument, `.fns`, is a function or list of functions to apply to each column. This can also be a purrr style formula (or list of formulas) like `~ .x / 2`. (This argument is optional, and you can omit it if you just want to get the underlying data; you'll see that technique used in `vignette("rowwise")`.)

Here are a couple of examples of `across()` in conjunction with its favourite verb, `summarise()`. But you can use `across()` with any dplyr verb, as you'll see a little later.

```
starwars %>%
  summarise(across(where(is.character), n_distinct))
#> # A tibble: 1 × 8
#>    name hair_color skin_color eye_color   sex gender homeworld species
#>   <int>      <int>      <int>     <int> <int>  <int>     <int>   <int>
#> 1    87         12         31        15     5      3        49      38

starwars %>%
  group_by(species) %>%
```

```
  filter(n() > 1) %>%
  summarise(across(c(sex, gender, homeworld), n_distinct))
#> # A tibble: 9 × 4
#>   species    sex gender homeworld
#>   <chr>    <int>  <int>     <int>
#> 1 Droid        1      2         3
#> 2 Gungan       1      1         1
#> 3 Human        2      2        15
#> 4 Kaminoan     2      2         1
#> # ℹ 5 more rows

starwars %>%
  group_by(homeworld) %>%
  filter(n() > 1) %>%
  summarise(across(where(is.numeric), ~ mean(.x, na.rm = TRUE)))
#> # A tibble: 10 × 4
#>   homeworld height  mass birth_year
#>   <chr>      <dbl> <dbl>      <dbl>
#> 1 Alderaan    176.  64         43
#> 2 Corellia    175   78.5       25
#> 3 Coruscant   174.  50         91
#> 4 Kamino      208.  83.1       31.5
#> # ℹ 6 more rows
```

Because `across()` is usually used in combination with `summarise()` and `mutate()`, it doesn't select grouping variables in order to avoid accidentally modifying them:

```
df <- data.frame(g = c(1, 1, 2), x = c(-1, 1, 3), y = c(-1, -4, -9))
df %>%
  group_by(g) %>%
  summarise(across(where(is.numeric), sum))
#> # A tibble: 2 × 3
#>       g     x     y
#>   <dbl> <dbl> <dbl>
#> 1     1     0    -5
#> 2     2     3    -9
```

Multiple functions

You can transform each variable with more than one function by supplying a named list of functions or lambda functions in the second argument:

```
min_max <- list(
  min = ~min(.x, na.rm = TRUE),
  max = ~max(.x, na.rm = TRUE)
)
starwars %>% summarise(across(where(is.numeric), min_max))
#> # A tibble: 1 × 6
#>   height_min height_max mass_min mass_max birth_year_min birth_year_max
```

```
#>        <int>      <int>    <dbl>    <dbl>          <dbl>          <dbl>
#> 1         66        264       15     1358              8            896
starwars %>% summarise(across(c(height, mass, birth_year), min_max))
#> # A tibble: 1 × 6
#>   height_min height_max mass_min mass_max birth_year_min birth_year_max
#>        <int>      <int>    <dbl>    <dbl>          <dbl>          <dbl>
#> 1         66        264       15     1358              8            896
```

Control how the names are created with the `.names` argument which takes a glue spec:

```
starwars %>% summarise(across(where(is.numeric), min_max, .names = "{.fn}.{.col}"))
#> # A tibble: 1 × 6
#>   min.height max.height min.mass max.mass min.birth_year max.birth_year
#>        <int>      <int>    <dbl>    <dbl>          <dbl>          <dbl>
#> 1         66        264       15     1358              8            896
starwars %>% summarise(across(c(height, mass, birth_year), min_max, .names = "{.fn}.
{.col}"))
#> # A tibble: 1 × 6
#>   min.height max.height min.mass max.mass min.birth_year max.birth_year
#>        <int>      <int>    <dbl>    <dbl>          <dbl>          <dbl>
#> 1         66        264       15     1358              8            896
```

If you'd prefer all summaries with the same function to be grouped together, you'll have to expand the calls yourself:

```
starwars %>% summarise(
  across(c(height, mass, birth_year), ~min(.x, na.rm = TRUE), .names = "min_{.col}"),
  across(c(height, mass, birth_year), ~max(.x, na.rm = TRUE), .names = "max_{.col}")
)
#> # A tibble: 1 × 6
#>   min_height min_mass min_birth_year max_height max_mass max_birth_year
#>        <int>    <dbl>          <dbl>      <int>    <dbl>          <dbl>
#> 1         66       15              8        264     1358            896
```

(One day this might become an argument to `across()` but we're not yet sure how it would work.)

We cannot however use `where(is.numeric)` in that last case because the second `across()` would pick up the variables that were newly created ("min_height", "min_mass" and "min_birth_year").

We can work around this by combining both calls to `across()` into a single expression that returns a tibble:

```
starwars %>% summarise(
  tibble(
    across(where(is.numeric), ~min(.x, na.rm = TRUE), .names = "min_{.col}"),
    across(where(is.numeric), ~max(.x, na.rm = TRUE), .names = "max_{.col}")
  )
)
#> # A tibble: 1 × 6
#>   min_height min_mass min_birth_year max_height max_mass max_birth_year
#>        <int>    <dbl>          <dbl>      <int>    <dbl>          <dbl>
#> 1         66       15              8        264     1358            896
```

Alternatively we could reorganize results with `relocate()`:

```
starwars %>%
  summarise(across(where(is.numeric), min_max, .names = "{.fn}.{.col}")) %>%
  relocate(starts_with("min"))
#> # A tibble: 1 × 6
#>   min.height min.mass min.birth_year max.height max.mass max.birth_year
#>        <int>    <dbl>          <dbl>      <int>    <dbl>          <dbl>
#> 1         66       15              8        264     1358            896
```

Current column

If you need to, you can access the name of the "current" column inside by calling `cur_column()`. This can be useful if you want to perform some sort of context dependent transformation that's already encoded in a vector:

```
df <- tibble(x = 1:3, y = 3:5, z = 5:7)
mult <- list(x = 1, y = 10, z = 100)

df %>% mutate(across(all_of(names(mult)), ~ .x * mult[[cur_column()]]))
#> # A tibble: 3 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1    30   500
#> 2     2    40   600
#> 3     3    50   700
```

Gotchas

Be careful when combining numeric summaries with `where(is.numeric)`:

```
df <- data.frame(x = c(1, 2, 3), y = c(1, 4, 9))

df %>%
  summarise(n = n(), across(where(is.numeric), sd))
#>    n x        y
#> 1 NA 1 4.041452
```

Here `n` becomes `NA` because `n` is numeric, so the `across()` computes its standard deviation, and the standard deviation of 3 (a constant) is `NA`. You probably want to compute `n()` last to avoid this problem:

```
df %>%
  summarise(across(where(is.numeric), sd), n = n())
#>   x        y n
#> 1 1 4.041452 3
```

Alternatively, you could explicitly exclude `n` from the columns to operate on:

```
df %>%
  summarise(n = n(), across(where(is.numeric) & !n, sd))
#>   n x        y
#> 1 3 1 4.041452
```

Another approach is to combine both the call to `n()` and `across()` in a single expression that returns a tibble:

```
df %>%
  summarise(
    tibble(n = n(), across(where(is.numeric), sd))
  )
#>   n x        y
#> 1 3 1 4.041452
```

Other verbs

So far we've focused on the use of `across()` with `summarise()`, but it works with any other dplyr verb that uses data masking:

- Rescale all numeric variables to range 0-1:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
df <- tibble(x = 1:4, y = rnorm(4))
df %>% mutate(across(where(is.numeric), rescale01))
#> # A tibble: 4 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1 0     0.385
#> 2 0.333 1
#> 3 0.667 0
#> 4 1     0.903
```

For some verbs, like `group_by()`, `count()` and `distinct()`, you don't need to supply a summary function, but it can be useful to use tidy-selection to dynamically select a set of columns. In those cases, we recommend using the complement to `across()`, `pick()`, which works like `across()` but doesn't apply any functions and instead returns a data frame containing the selected columns.

- Find all distinct

```
starwars %>% distinct(pick(contains("color")))
#> # A tibble: 67 × 3
#>   hair_color skin_color  eye_color
#>   <chr>      <chr>       <chr>
#> 1 blond      fair        blue
#> 2 <NA>       gold        yellow
#> 3 <NA>       white, blue red
```

```
#> 4 none       white       yellow
#> # ℹ 63 more rows
```

- Count all combinations of variables with a given pattern:

```
starwars %>% count(pick(contains("color")), sort = TRUE)
#> # A tibble: 67 × 4
#>   hair_color skin_color eye_color     n
#>   <chr>      <chr>      <chr>     <int>
#> 1 brown      light      brown         6
#> 2 brown      fair       blue          4
#> 3 none       grey       black         4
#> 4 black      dark       brown         3
#> # ℹ 63 more rows
```

`across()` doesn't work with `select()` or `rename()` because they already use tidy select syntax; if you want to transform column names with a function, you can use `rename_with()`.

filter()

We cannot directly use `across()` in `filter()` because we need an extra step to combine the results. To that end, `filter()` has two special purpose companion functions:

- `if_any()` keeps the rows where the predicate is true for *at least one* selected column:

```
starwars %>%
  filter(if_any(everything(), ~ !is.na(.x)))
#> # A tibble: 87 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Luke Sky…    172    77 blond      fair       blue            19   male  mascu…
#> 2 C-3PO        167    75 <NA>       gold       yellow         112   none  mascu…
#> 3 R2-D2         96    32 <NA>       white, bl… red             33   none  mascu…
#> 4 Darth Va…    202   136 none       white      yellow          41.9 male  mascu…
#> # ℹ 83 more rows
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

- `if_all()` keeps the rows where the predicate is true for *all* selected columns:

```
starwars %>%
  filter(if_all(everything(), ~ !is.na(.x)))
#> # A tibble: 29 × 14
#>   name      height  mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
#> 1 Luke Sky…    172    77 blond      fair       blue            19   male  mascu…
#> 2 Darth Va…    202   136 none       white      yellow          41.9 male  mascu…
#> 3 Leia Org…    150    49 brown      light      brown           19   fema… femin…
#> 4 Owen Lars    178   120 brown, gr… light      blue            52   male  mascu…
#> # ℹ 25 more rows
```

```
#> # ℹ 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

_if, _at, _all

Prior versions of dplyr allowed you to apply a function to multiple columns in a different way: using functions with `_if`, `_at`, and `_all()` suffixes. These functions solved a pressing need and are used by many people, but are now superseded. That means that they'll stay around, but won't receive any new features and will only get critical bug fixes.

Why do we like `across()`?

Why did we decide to move away from these functions in favour of `across()`?

1. `across()` makes it possible to express useful summaries that were previously impossible:

```
df %>%
  group_by(g1, g2) %>%
  summarise(
    across(where(is.numeric), mean),
    across(where(is.factor), nlevels),
    n = n(),
  )
```

2. `across()` reduces the number of functions that dplyr needs to provide. This makes dplyr easier for you to use (because there are fewer functions to remember) and easier for us to implement new verbs (since we only need to implement one function, not four).
3. `across()` unifies `_if` and `_at` semantics so that you can select by position, name, and type, and you can now create compound selections that were previously impossible. For example, you can now transform all numeric columns whose name begins with "x": `across(where(is.numeric) & starts_with("x"))`.
4. `across()` doesn't need to use `vars()`. The `_at()` functions are the only place in dplyr where you have to manually quote variable names, which makes them a little weird and hence harder to remember.

Why did it take so long to discover `across()`?

It's disappointing that we didn't discover `across()` earlier, and instead worked through several false starts (first not realising that it was a common problem, then with the `_each()` functions, and most recently with the `_if()`/`_at()`/`_all()` functions). But `across()` couldn't work without three recent discoveries:

- You can have a column of a data frame that is itself a data frame. This is something provided by base R, but it's not very well documented, and it took a while to see that it was useful, not just a theoretical curiosity.
- We can use data frames to allow summary functions to return multiple columns.
- We can use the absence of an outer name as a convention that you want to unpack a data frame column into individual columns.

How do you convert existing code?

Fortunately, it's generally straightforward to translate your existing code to use `across()`:

- Strip the `_if()`, `_at()` and `_all()` suffix off the function.
- Call `across()`. The first argument will be:
  - For `_if()`, the old second argument wrapped in `where()`.
  - For `_at()`, the old second argument, with the call to `vars()` removed.
  - For `_all()`, `everything()`.

  The subsequent arguments can be copied as is.

For example:

```
df %>% mutate_if(is.numeric, ~mean(.x, na.rm = TRUE))
# ->
df %>% mutate(across(where(is.numeric), ~mean(.x, na.rm = TRUE)))

df %>% mutate_at(vars(c(x, starts_with("y"))), mean)
# ->
df %>% mutate(across(c(x, starts_with("y")), mean))

df %>% mutate_all(mean)
# ->
df %>% mutate(across(everything(), mean))
```

There are a few exceptions to this rule:

- `rename_*()` and `select_*()` follow a different pattern. They already have select semantics, so are generally used in a different way that doesn't have a direct equivalent with `across()`; use the new `rename_with()` instead.
- Previously, `filter_*()` were paired with the `all_vars()` and `any_vars()` helpers. The new helpers `if_any()` and `if_all()` can be used inside `filter()` to keep rows for which the predicate is true for at least one, or all selected columns:

```
df <- tibble(x = c("a", "b"), y = c(1, 1), z = c(-1, 1))

# Find all rows where EVERY numeric variable is greater than zero
df %>% filter(if_all(where(is.numeric), ~ .x > 0))
#> # A tibble: 1 × 3
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1 b         1     1

# Find all rows where ANY numeric variable is greater than zero
df %>% filter(if_any(where(is.numeric), ~ .x > 0))
#> # A tibble: 2 × 3
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1 a         1    -1
#> 2 b         1     1
```

- When used in a `mutate()`, all transformations performed by an `across()` are applied at once. This is different to the behaviour of `mutate_if()`, `mutate_at()`, and `mutate_all()`, which apply the transformations one at a time. We expect that you'll generally find the new behaviour less surprising:

```
df <- tibble(x = 2, y = 4, z = 8)
df %>% mutate_all(~ .x / y)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1   0.5     1     8

df %>% mutate(across(everything(), ~ .x / y))
#> # A tibble: 1 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1   0.5     1     2
```