10 How to apply a function across many columns

> What will this tutorial cover?
>
> In this tutorial you will learn how to apply one or more functions to many columns. You will learn to explain the structure of the function `across` and you will be able to use the function for six use cases.

> Who do I have to thank?
>
> For this tutorial, I have to thank Rebecca Barter, who wrote a wonderful blog post about the `across` function. I have adapted a few of her examples for this tutorial.

One of the credos of programming is "Don't repeat yourself". We have seen in previous tutorials that many of us fall victim to this principle quite often. Fortunately, the tidyverse team has developed a set of functions that make it easier not to repeat ourselves. We have seen it with the `rename_with` function, which allows us to rename many columns at once. In this tutorial, we'll take this idea further and figure out how to use the `across` function to apply a function to many columns.

We will talk about six use cases of the across function. These cases are tied to two dplyr functions: `summarise` and `mutate`:

- `summarise`: How to calculate summary statistics across many columns
- `summarise`: How to calculate the number of distinct values across many columns
- `mutate`: How to change the variable type across many columns
- `mutate`: How to normalize many columns
- `mutate`: How to impute values across many columns
- `mutate`: How to replace characters across many columns

Each use case will work with this general structure:

```
<DFRAME> %>%
  <DPLYR VERB>(
    across(
      .cols  = <SELECTION OF COLUMNS>,
      .fns   = <FUNCTION TO BE APPLIED TO EACH COLUMN>,
      .names = <NAME OF THE GENERATED COLUMNS>
    )
  )
```

A couple of things are important here:

- The function across only works inside dplyr verbs (e.g. `mutate`)
- The function has three important arguments: `.cols` stands for the column to apply a function to. You can use the tidyselect functions here; `.fns` stands for the function(s) that will be applied to these columns; `.names` is used whenever you want to change the names of the selected columns.
- The `.fns` argument takes three different values: (1) A simple function (e.g. `mean`). (2) A purrr-style lambda function (e.g. `~ mean(.x, na.rm = TRUE)`). You use these lamba functions if you need to change the arguments of a function. (3) A list of functions (e.g. `list(mean = mean, sd = sd)`). The list can also be combined with lambda functions (e.g. `list(mean = mean(.x, na.rm = TRUE), sd = sd(.x, na.rm = TRUE))`).

For the following examples, I will stick to this structure so that you can easily see the differences between the code snippets. Let' start with the combination of `across` and `summarise`.

10.1 across and summarise

How to calculate summary statistics across many columns

Here is a typical example of how many people calculate summary statistics with dplyr:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    mean_displ = mean(displ, na.rm = TRUE),
    mean_cty   = mean(cty  , na.rm = TRUE),
  )
```

```
# A tibble: 15 x 3
   manufacturer mean_displ mean_cty
   <chr>             <dbl>    <dbl>
 1 audi               2.54     17.6
 2 chevrolet          5.06     15
 3 dodge              4.38     13.1
 4 ford               4.54     14
 5 honda              1.71     24.4
 6 hyundai            2.43     18.6
 7 jeep               4.58     13.5
 8 land rover         4.3      11.5
 9 lincoln            5.4      11.3
10 mercury            4.4      13.2
11 nissan             3.27     18.1
12 pontiac            3.96     17
13 subaru             2.46     19.3
14 toyota             2.95     18.5
15 volkswagen         2.26     20.9
```

This approach works, but it is not scalable. Imagine you had to calculate the mean and standard deviation of dozens of columns.

Instead you can use the `across` function to get the same result:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols  = c("displ", "cty"),
      .fns   = ~ mean(.x, na.rm = TRUE),
      .names = "mean_{.col}"
    )
  )
```

```
# A tibble: 15 x 3
   manufacturer mean_displ mean_cty
   <chr>             <dbl>    <dbl>
 1 audi               2.54     17.6
 2 chevrolet          5.06     15
 3 dodge              4.38     13.1
 4 ford               4.54     14
 5 honda              1.71     24.4
```

```
 6 hyundai            2.43     18.6
 7 jeep               4.58     13.5
 8 land rover         4.3      11.5
 9 lincoln            5.4      11.3
10 mercury            4.4      13.2
11 nissan             3.27     18.1
12 pontiac            3.96     17
13 subaru             2.46     19.3
14 toyota             2.95     18.5
15 volkswagen         2.26     20.9
```

With the argument `.cols` (`.cols = c("displ", "cty")`) we told across that we want to apply a function to these two columns. With the argument `.fns` (`.fns = ~ mean(.x, na.rm = TRUE)`) we told across that we want to calculate the mean of the two columns. We also want to remove NAs from each column. Using the `.names` argument (`.names = "mean_{.col}"`), we told across that we want the summarised columns to have the following structure: Start with the string `mean_` and glue the name of the column to this string (`{.col}`).

Suppose, we would like to also calculate the standard deviation of each column. In this case, we need to provide a list to the `.fns` argument instead of a purr-style lambda function:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols  = c("displ", "cty"),
      .fns   = list(mean = mean, sd = sd),
      .names = "{.fn}_{.col}"
    )
  )
```

```
# A tibble: 15 x 5
   manufacturer mean_displ sd_displ mean_cty sd_cty
   <chr>             <dbl>    <dbl>    <dbl>  <dbl>
 1 audi               2.54    0.673     17.6  1.97
 2 chevrolet          5.06    1.37      15    2.92
 3 dodge              4.38    0.868     13.1  2.49
 4 ford               4.54    0.541     14    1.91
 5 honda              1.71    0.145     24.4  1.94
 6 hyundai            2.43    0.365     18.6  1.50
 7 jeep               4.58    1.02      13.5  2.51
 8 land rover         4.3     0.258     11.5  0.577
```

```
 9 lincoln            5.4     0         11.3  0.577
10 mercury            4.4     0.490     13.2  0.5
11 nissan             3.27    0.864     18.1  3.43
12 pontiac            3.96    0.808     17    1
13 subaru             2.46    0.109     19.3  0.914
14 toyota             2.95    0.931     18.5  4.05
15 volkswagen         2.26    0.443     20.9  4.56
```

Two things have changed. First, we created a list with two functions (`list(mean = mean, sd = sd)`). As I told you at the beginning of this tutorial, you can also combine the list with purrr-style lambda functions: `list(mean = ~ mean(.x, na.rm = TRUE), sd = ~ sd(.x, na.rm = TRUE))`. Second, we changed the string of the `.names` argument. Instead of `mean_{.col}` we use the string `{.fn}_{.col}`. `{.fn}` and `{.col}` are special symbols and stand for the function and the column. Since we have more than one function, we need to have the flexibility to create column names that combine the name of the column with the name of the function applied to the column.

Once we have this structure, we can add as many columns and functions as we need:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols  = where(is.numeric),
      .fns   = list(mean   = mean,   sd = sd,
                    median = median),
      .names = "{.fn}_{.col}"
    )
  ) %>%
  glimpse()
```

```
Rows: 15
Columns: 16
$ manufacturer <chr> "audi", "chevrolet", "dodge", "ford", "honda", "hyundai",~
$ mean_displ   <dbl> 2.544444, 5.063158, 4.378378, 4.536000, 1.711111, 2.42857~
$ sd_displ     <dbl> 0.6732032, 1.3704057, 0.8679910, 0.5407402, 0.1452966, 0.~
$ median_displ <dbl> 2.8, 5.3, 4.7, 4.6, 1.6, 2.4, 4.7, 4.3, 5.4, 4.3, 3.3, 3.~
$ mean_year    <dbl> 2003.500, 2004.684, 2004.108, 2002.600, 2003.000, 2004.14~
$ sd_year      <dbl> 4.630462, 4.460352, 4.520225, 4.500000, 4.743416, 4.62197~
$ median_year  <dbl> 2003.5, 2008.0, 2008.0, 1999.0, 1999.0, 2008.0, 2008.0, 2~
$ mean_cyl     <dbl> 5.222222, 7.263158, 7.081081, 7.200000, 4.000000, 4.85714~
$ sd_cyl       <dbl> 1.2153700, 1.3679711, 1.1150082, 1.0000000, 0.0000000, 1.~
```

```
$ median_cyl   <dbl> 6, 8, 8, 8, 4, 4, 8, 8, 8, 7, 6, 6, 4, 4, 4
$ mean_cty     <dbl> 17.61111, 15.00000, 13.13514, 14.00000, 24.44444, 18.6428~
$ sd_cty       <dbl> 1.9745108, 2.9249881, 2.4850907, 1.9148542, 1.9436506, 1.~
$ median_cty   <dbl> 17.5, 15.0, 13.0, 14.0, 24.0, 18.5, 14.0, 11.5, 11.0, 13.~
$ mean_hwy     <dbl> 26.44444, 21.89474, 17.94595, 19.36000, 32.55556, 26.8571~
$ sd_hwy       <dbl> 2.175322, 5.108759, 3.574182, 3.327662, 2.554952, 2.17881~
$ median_hwy   <dbl> 26.0, 23.0, 17.0, 18.0, 32.0, 26.5, 18.5, 16.5, 17.0, 18.~
```

See how we used a tidyselect function instead of a vector of column names? This gives us tremendous flexibility. Also note that we can easily compute four different summary statistics once we have a list.

Before we continue with the next use case, I would like to show you an example that the dplyr developers advise against. What I have not told you so far is that the function `across` can also take additional arguments. For example the `na.rm` argument of the `mean` function:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(
    across(
      .cols  = c("displ", "cty"),
      .fns   = mean,
      .names = "mean_{.col}",
      na.rm = TRUE
    )
  )
```

```
# A tibble: 15 x 3
   manufacturer mean_displ mean_cty
   <chr>             <dbl>    <dbl>
 1 audi               2.54     17.6
 2 chevrolet          5.06     15
 3 dodge              4.38     13.1
 4 ford               4.54     14
 5 honda              1.71     24.4
 6 hyundai            2.43     18.6
 7 jeep               4.58     13.5
 8 land rover         4.3      11.5
 9 lincoln            5.4      11.3
10 mercury            4.4      13.2
11 nissan             3.27     18.1
12 pontiac            3.96     17
```

```
13 subaru             2.46     19.3
14 toyota             2.95     18.5
15 volkswagen         2.26     20.9
```

There are two reasons not to do this: First, it causes problems in timing the evaluation. In other words, it can lead to errors. Second, it decouples the arguments from the functions to which they are applied. The best thing you can do is not to do this.

How to calculate the number of distinct values across many columns

This tip comes from Rebecca Barter. First of all, you can be quite creative with `across` and `summmarise`, because you can calculate any summary statistic for many columns. The mean, standard deviation or median are just obvious examples. Making the problem smaller and easier to digest, you might simply ask yourself, "What summary statistics can I compute from a vector?" One such statistic is the number of distinct values in a vector: How many people do I have? From how many states do these people come from? How many manufacturers are in the data?

```
mpg %>%
  summarise(
    across(
      .cols = everything(),
      .fns  = n_distinct
    )
  )
```

```
# A tibble: 1 x 11
  manufacturer model displ  year   cyl trans   drv   cty   hwy    fl class
         <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
1           15    38    35     2     4    10     3    21    27     5     7
```

A look at the results shows that the data includes 15 car manufacturers and 38 different car models.

10.2 across and mutate

How to change the variable type across many columns

Suppose some of your character columns need to be transferred to a factor. Let's first see how this works with `across` and `mutate` and then go through the example in more detail:

```
mpg %>%
  mutate(
    across(
      .cols = where(is.character),
      .fns  = as_factor
    )
  ) %>%
  select(where(is.factor) | where(is.character)) %>%
  glimpse()
```

```
Rows: 234
Columns: 6
$ manufacturer <fct> audi, audi, audi, audi, audi, audi, audi, audi, audi, aud~
$ model        <fct> a4, a4, a4, a4, a4, a4, a4, a4 quattro, a4 quattro, a4 qu~
$ trans        <fct> auto(l5), manual(m5), manual(m6), auto(av), auto(l5), man~
$ drv          <fct> f, f, f, f, f, f, f, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, r, ~
$ fl           <fct> p, p, p, p, p, p, p, p, p, p, p, p, p, p, p, p, p, p, r, ~
$ class        <fct> compact, compact, compact, compact, compact, compact, com~
```

With `mutate` you normally specify the new column to be created or overwritten. With `across` you don't have to do that. Without specifying the `.names` argument, the names of your columns remain the same, you just apply the function(s) to those columns. In this example, we applied the function `as_factor` to each character column. We could have also changed the names of the columns and created new columns instead:

```
mpg %>%
  mutate(
    across(
      .cols = where(is.character),
      .fns  = as_factor,
      .names = "{.col}_as_factor"
    )
  ) %>%
  select(where(is.factor) | where(is.character)) %>%
  glimpse()
```

```
Rows: 234
Columns: 12
$ manufacturer_as_factor <fct> audi, audi, audi, audi, audi, audi, audi, audi,~
$ model_as_factor        <fct> a4, a4, a4, a4, a4, a4, a4, a4 quattro, a4 quat~
```

```
$ trans_as_factor       <fct> auto(l5), manual(m5), manual(m6), auto(av), aut~
$ drv_as_factor         <fct> f, f, f, f, f, f, f, 4, 4, 4, 4, 4, 4, 4, 4, 4,~
$ fl_as_factor          <fct> p, p, p, p, p, p, p, p, p, p, p, p, p, p, p, p,~
$ class_as_factor       <fct> compact, compact, compact, compact, compact, co~
$ manufacturer          <chr> "audi", "audi", "audi", "audi", "audi", "audi",~
$ model                 <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 q~
$ trans                 <chr> "auto(l5)", "manual(m5)", "manual(m6)", "auto(a~
$ drv                   <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4~
$ fl                    <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p~
$ class                 <chr> "compact", "compact", "compact", "compact", "co~
```

How to normalize many columns

Statisticians and scientists often need to normalize their data. Suppose you want to normalize your data so that each column has a mean of 0 and a standard deviation of 1:

```
scaled_columns <- mpg %>%
  transmute(
    across(
      .cols = where(is.numeric),
      .fns  = scale,
    )
  )

scaled_columns %>%
  glimpse()
```

```
Rows: 234
Columns: 5
$ displ <dbl[,1]> <matrix[26 x 1]>
$ year  <dbl[,1]> <matrix[26 x 1]>
$ cyl   <dbl[,1]> <matrix[26 x 1]>
$ cty   <dbl[,1]> <matrix[26 x 1]>
$ hwy   <dbl[,1]> <matrix[26 x 1]>
```

Two things: First, I used the function `transmute` to keep only the columns to which the function `scale` was applied. Second, the result is not a set of vectors, but matrices. Let's have a look at one of these columns:

```
scaled_columns$displ %>% head
```

```
           [,1]
[1,] -1.2939999
[2,] -1.2939999
[3,] -1.1391962
[4,] -1.1391962
[5,] -0.5199816
[6,] -0.5199816
```

This is a matrix with one column and many rows. To return only the vector of this column, we can index it:

```
scaled_columns$displ[,1] %>% head
```

```
[1] -1.2939999 -1.2939999 -1.1391962 -1.1391962 -0.5199816 -0.5199816
```

The value before the comma indicates the rows we want to select, the value after the comma indicates the columns. To fix our problem, we need to add this syntax to our `scale` function:

```
mpg %>%
  transmute(
    across(
      .cols = where(is.numeric),
      .fns  = ~ scale(.)[,1],
    )
  ) %>%
  glimpse()
```

```
Rows: 234
Columns: 5
$ displ <dbl> -1.2939999, -1.2939999, -1.1391962, -1.1391962, -0.5199816, -0.5~
$ year  <dbl> -0.997861, -0.997861, 0.997861, 0.997861, -0.997861, -0.997861, ~
$ cyl   <dbl> -1.1721058, -1.1721058, -1.1721058, -1.1721058, 0.0689474, 0.068~
$ cty   <dbl> 0.26810155, 0.97299777, 0.73803236, 0.97299777, -0.20182926, 0.2~
$ hwy   <dbl> 0.93369639, 0.93369639, 1.26956872, 1.10163255, 0.42988788, 0.42~
```

```
scaled_columns <- mpg %>%
  transmute(
    across(
      .cols = where(is.numeric),
```

```
      .fns   = ~ scale(.)[,1],
    )
  )

scaled_columns %>%
  glimpse()
```

```
Rows: 234
Columns: 5
$ displ <dbl> -1.2939999, -1.2939999, -1.1391962, -1.1391962, -0.5199816, -0.5~
$ year  <dbl> -0.997861, -0.997861, 0.997861, 0.997861, -0.997861, -0.997861, ~
$ cyl   <dbl> -1.1721058, -1.1721058, -1.1721058, -1.1721058, 0.0689474, 0.068~
$ cty   <dbl> 0.26810155, 0.97299777, 0.73803236, 0.97299777, -0.20182926, 0.2~
$ hwy   <dbl> 0.93369639, 0.93369639, 1.26956872, 1.10163255, 0.42988788, 0.42~
```

We can prove that the columns were scaled (mean = 1, sd = 0) with the `across summarise`:

```
scaled_columns %>%
  summarise(
    across(
      .cols = everything(),
      .fns  = list(mean = mean, sd = sd)
    )
  ) %>%
  glimpse()
```

```
Rows: 1
Columns: 10
$ displ_mean <dbl> 1.56392e-15
$ displ_sd   <dbl> 1
$ year_mean  <dbl> 5.693451e-18
$ year_sd    <dbl> 1
$ cyl_mean   <dbl> -1.684313e-17
$ cyl_sd     <dbl> 1
$ cty_mean   <dbl> 2.095605e-16
$ cty_sd     <dbl> 1
$ hwy_mean   <dbl> -2.097088e-16
$ hwy_sd     <dbl> 1
```

How to impute values across many columns

Again, I have to thank Rebecca Barter and her good blog post on the across function for this trick. When we impute missing values, we replace them with substituted values. I am not an expert in this field, but I can show you how the method of imputation with `across` and `mutate` might work.

Suppose this is your data frame:

```
(dframe <- tibble(
  group = c("a", "a", "a", "b", "b", "b"),
  x     = c(3,  5, 4, NA, 4, 8),
  y     = c(2, NA, 3,  1, 9, 7)
))
```

```
# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
1 a         3     2
2 a         5    NA
3 a         4     3
4 b        NA     1
5 b         4     9
6 b         8     7
```

You have two columns and in both columns you have a missing value. You want to replace each missing value with the mean value of the respective column:

```
dframe %>%
  mutate(
    across(
      .cols = c(x, y), # or everything()
      .fns  = ~ ifelse(test = is.na(.),
                       yes  = mean(., na.rm = TRUE),
                       no   = .)
    )
  )
```

```
# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
```

```
1 a       3     2
2 a       5     4.4
3 a       4     3
4 b       4.8   1
5 b       4     9
6 b       8     7
```

For each value in each column, we test if the value is an `NA`. If it is, we replace this value with the value of the column, if it is a real number, we keep it. We could just as well have used the vectorized if function `case_when`:

```
dframe %>%
  mutate(
    across(
      .cols = c(x, y), # or everything()
      .fns  = ~ case_when(
        is.na(.) ~ mean(., na.rm = TRUE),
        TRUE ~ .
      )
    )
  )
```

```
# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
1 a       3     2
2 a       5     4.4
3 a       4     3
4 b       4.8   1
5 b       4     9
6 b       8     7
```

You can also impute the values within groups:

```
dframe %>%
  group_by(group) %>%
  mutate(
    across(
      .cols = c(x, y), # or everything()
      .fns  = ~ case_when(
        is.na(.) ~ mean(., na.rm = TRUE),
```

```
        TRUE ~ .
      )
    )
  ) %>%
  ungroup()
```

```
# A tibble: 6 x 3
  group     x     y
  <chr> <dbl> <dbl>
1 a         3   2
2 a         5   2.5
3 a         4   3
4 b         6   1
5 b         4   9
6 b         8   7
```

For example, the new value 6 from the column `x` is the mean of the values within the group "b" $((8 + 4)/2)$.

How to replace characters across many columns

Suppose you have the same typo in many columns:

```
typo_dframe <- tribble(
  ~pre_test,   ~post_test,
  "goud"       , "good",
  "medium"     , "good",
  "metium"     , "metium",
  "bad"        , "goud"
)
```

"goud" should be "good" and "metium" should be "medium". Again, we can combine `mutate` with `across` to correct these typos across both columns:

```
(typo_corrected <- typo_dframe %>%
  mutate(
    across(
      .cols = everything(),
      .fns  = ~ case_when(
        str_detect(., "goud")   ~ str_replace(., "goud", "good"),
```

```
        str_detect(., "metium") ~ str_replace(., "metium", "medium"),
        TRUE ~ .
      )
    )
  ))
```

```
# A tibble: 4 x 2
  pre_test post_test
  <chr>    <chr>
1 good     good
2 medium   good
3 medium   medium
4 bad      good
```

Summary

Here's what you can take away from this tutorial.

- The function `across` has three important arguments: `.cols`, `.fns`, and `.names`
- `across` must be used inside dplyr functions.
- The most common dplyr functions that use `across` are `mutate` and `summarise`.
- `across` keeps the column names by default, you can change them with `.names`.