The across function

The GRAPH Courses team

November 2022

This document serves as an accompaniment for a lesson found on https://thegraphcourses.org.

The GRAPH Courses is a project of the Global Research and Analyses for Public Health (GRAPH) Network, with the support of the World Health Organization (WHO) and other partners

Intro

In previous lessons, you learned how to perform a range of wrangling operations like filtering, mutating and summarizing. But so far, you only performed these operations *one column at a time*. Sometimes however, it will be useful (and efficient) to apply the same operation to *several columns at the same time*. For this, the `across()` function can be used.

Let's see how!

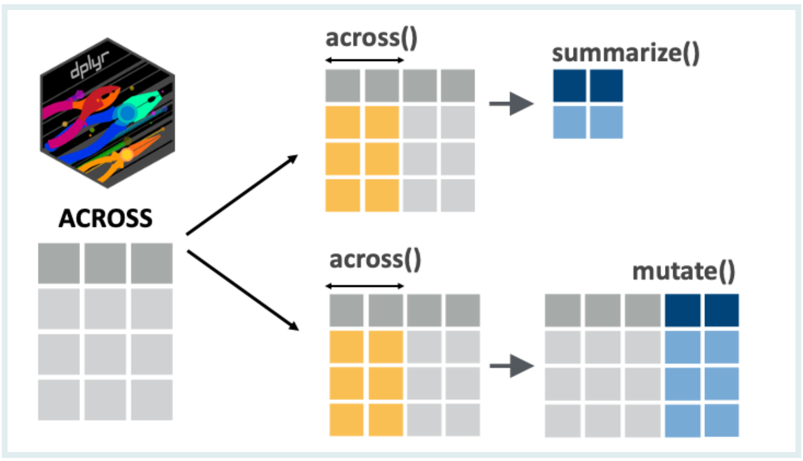

Fig: the `across()` verb.

Learning objectives

1. You can use `across()` with the `mutate()` and `summarize()` verbs to apply operations over multiple columns.

2. You can use the `.names` argument within `mutate(across())` to create new columns.

3. You can write anonymous (lambda) functions within `across()`

Packages

This lesson will require the packages loaded below:

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(here, tidyverse)
```

Datasets

In this lesson, we will again use data from the COVID-19 serological survey conducted in Yaounde, Cameroon.

```
yaounde <- read_csv(here("data/yaounde_data.csv"))

yaounde <- yaounde %>% rename(age_years = age)

yaounde
```

```
## # A tibble: 5 × 53
##   id                    date_surveyed age_years age_category
##   <chr>                 <date>            <dbl> <chr>
## 1 BRIQUETERIE_000_0001 2020-10-22           45 45 - 64
## 2 BRIQUETERIE_000_0002 2020-10-24           55 45 - 64
## 3 BRIQUETERIE_000_0003 2020-10-24           23 15 - 29
## 4 BRIQUETERIE_002_0001 2020-10-22           20 15 - 29
## 5 BRIQUETERIE_002_0002 2020-10-22           55 45 - 64
## # … with 49 more variables: age_category_3 <chr>,
## #   sex <chr>, highest_education <chr>, occupation <chr>, …
```

We will also use data from a hospital study conducted in Burkina Faso, in which a range of clinical data was collected from patients with febrile (fever-causing) diseases, with the aim of predicting the cause of the fever.

```
febrile_diseases <- read_csv(here("data/febrile_diseases_burkina_faso.csv"))
febrile_diseases
```

```
## # A tibble: 5 × 36
##   age_category     sexe   pretreatment_ma…¹ pretreatment_an…²
##   <chr>            <chr>  <chr>             <chr>
## 1 5 years or old… 2      do not know       currently taking
## 2 5 years or old… 2      no                no
## 3 5 years or old… female currently taking  no
## 4 5 years or old… female no                currently taking
## 5 5 years or old… female no                no
## # … with 32 more variables: onset_fever <dbl>,
## #   abd_pain <chr>, diarrhoea <chr>, runny_nose <chr>, …
```

Finally, we will use data from a dietary diversity survey conducted in Vietnam, in which women were asked to recall (one or several days) the foods and drinks they consumed the previous day.

```r
diet <- read_csv(here("data/vietnam_diet_diversity.csv"))
diet <- diet %>% rename(household_id = hhid)

diet
```

```
## # A tibble: 5 × 45
##   household_id date_of_visit       age_y age_group
##          <dbl> <dttm>              <dbl> <chr>
## 1          278 2017-05-23 00:00:00    47 40-49
## 2          348 2017-06-17 00:00:00    34 30-39
## 3          354 2017-06-17 00:00:00    37 30-39
## 4          324 2017-06-17 00:00:00    35 30-39
## 5          209 2017-06-07 00:00:00    35 30-39
## # … with 41 more variables: kilocalories_consumed <dbl>,
## #   water_consumed_grams <dbl>, …
```

Using `across()` with `mutate()`

The `mutate()` function gives you an easy way to create new variables or modify in place variables.

But sometimes you have a large number of columns to operate on, and typing out `mutate()` statements line-by-line can become onerous. In such cases `across()` can radically simplify and shorten your code.

Let's see an example.

Consider the symptoms columns (from `symp_fever` to `symp_stomach_ache`) in the `yaounde` data frame:

```r
yao_symptoms <-
  yaounde %>%
  select(age_years, sex, date_surveyed, symp_fever:symp_stomach_ache)

yao_symptoms
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>        
## 1        45 Female 2020-10-22    No         No           
## 2        55 Male   2020-10-24    No         No           
## 3        23 Male   2020-10-24    No         No           
## 4        20 Female 2020-10-22    No         No           
## 5        55 Female 2020-10-22    No         No           
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

The **13 columns** between `symp_fever` and `symp_stomach_ache` indicate whether or not each respondent had a specific COVID-compatible symptom.

Now, imagine you wanted to convert all these columns to upper case. (That is, "Yes" to "YES" and "No" to "NO"). How might you do this? Without `across()`, you would have to mutate the columns one by one, with the `toupper()` function:

```r
yao_symptoms %>%
  mutate(symp_fever = toupper(symp_fever),
         symp_headache = toupper(symp_headache),
         symp_cough = toupper(symp_cough),
         symp_rhinitis = toupper(symp_rhinitis),
         symp_sneezing = toupper(symp_sneezing),
         symp_fatigue = toupper(symp_fatigue),
         symp_muscle_pain = toupper(symp_muscle_pain)
         #... And on and on and on and on
         )
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>        
## 1        45 Female 2020-10-22    NO         NO           
## 2        55 Male   2020-10-24    NO         NO           
## 3        23 Male   2020-10-24    NO         NO           
## 4        20 Female 2020-10-22    NO         NO           
## 5        55 Female 2020-10-22    NO         NO           
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

This is obviously not very time-efficient. An experienced data analyst who saw this code might scold you for not obeying the **DRY ("Don't Repeat Yourself") principle of programming**.

But with the `across()` function, you have the power to do this in all of two lines:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    NO         NO
## 2        55 Male   2020-10-24    NO         NO
## 3        23 Male   2020-10-24    NO         NO
## 4        20 Female 2020-10-22    NO         NO
## 5        55 Female 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Amazing!

Let's break down the code above. We used `across()` inside of the `mutate()` function, and provided it with two main arguments:

- `.cols` defined the *columns* to be modified. The `symp_fever:symp_stomach_ache` code means "all columns between `symp_fever` and `symp_stomach_ache`".
- `.fns` defined the *functions* to apply on the selected columns. In this case, the `toupper` function was applied.

And that's the basic gist of `across()`! But below we'll consider each of these arguments in a bit more detail.

Why follow the DRY (Don't Repeat Yourself) principle?

SIDE NOTE

There are many reasons to avoid repetitive code. Here are just a few:

1. You'll save time in writing the code (obviously).
2. You'll also save time in *maintaining* the code. This is because if you need to make a change (e.g. switch `toupper` to `tolower`), you won't need to make the same change in several places. You can fix it in a single place.
3. DRY code is usually easier to read and understand, both by yourself and by others.

The `.cols` argument

Now let's look at the arguments of `across()` in some more detail.

As mentioned above, the `.cols` argument of `across()` selects the columns to be modified.

Most the different methods you have learned for selecting columns can be used here.

One difference with the classical use of `select()` is that to list column names with `across()`, you must wrap them in `c()`:

```
yao_symptoms %>%
  mutate(across(.cols = c(symp_fever, symp_headache, symp_cough),
                .fns = toupper))
```

If, instead of `c(symp_fever, symp_headache, symp_cough)` you just put in `symp_fever, symp_headache, symp_cough`, you'll get an error:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever, symp_headache, symp_cough, # Don't do this
                .fns = toupper))
```

```
Error in `mutate()`:
! Problem while computing `..1 = across(.cols = symp_fever, symp_headache....
```

Other than that, the usual variable selection methods can be used here.

So you can use numeric ranges, like `4:16`:

```
yao_symptoms %>%
  mutate(across(.cols = 4:16,
                .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    NO         NO
## 2        55 Male   2020-10-24    NO         NO
## 3        23 Male   2020-10-24    NO         NO
## 4        20 Female 2020-10-22    NO         NO
## 5        55 Female 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Or helper verbs like `starts_with()`:

```r
yao_symptoms %>%
  mutate(across(.cols = starts_with("symp_"),
                .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    NO         NO
## 2        55 Male   2020-10-24    NO         NO
## 3        23 Male   2020-10-24    NO         NO
## 4        20 Female 2020-10-22    NO         NO
## 5        55 Female 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Or the function `where()` to select columns of a particular type:

```r
yao_symptoms %>%
  mutate(across(.cols = where(is.character),
                .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 FEMALE 2020-10-22    NO         NO
## 2        55 MALE   2020-10-24    NO         NO
## 3        23 MALE   2020-10-24    NO         NO
## 4        20 FEMALE 2020-10-22    NO         NO
## 5        55 FEMALE 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Or the catch-all `everything()`:

```r
yao_symptoms %>%
  mutate(across(.cols = everything(),
                .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##   <chr>     <chr>  <chr>         <chr>      <chr>
## 1 45        FEMALE 2020-10-22    NO         NO
## 2 55        MALE   2020-10-24    NO         NO
## 3 23        MALE   2020-10-24    NO         NO
## 4 20        FEMALE 2020-10-22    NO         NO
## 5 55        FEMALE 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Note that `everything()` is the default value for the `.cols`. So the above code, is equivalent to simply running:

```
yao_symptoms %>%
  mutate(across(.fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##   <chr>     <chr>  <chr>         <chr>      <chr>
## 1 45        FEMALE 2020-10-22    NO         NO
## 2 55        MALE   2020-10-24    NO         NO
## 3 23        MALE   2020-10-24    NO         NO
## 4 20        FEMALE 2020-10-22    NO         NO
## 5 55        FEMALE 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

PRACTICE

(in RMD)

In the `febrile_diseases` dataset, the columns from `abd_pain` to `splenomegaly` indicate whether a patient had a specified symptom, recorded as "yes" or "no".

```
febrile_diseases %>%
  select(abd_pain:splenomegaly)
```

```
## # A tibble: 5 × 18
##   abd_pain diarrhoea runny_nose earpain throat_ache cough
##   <chr>    <chr>     <chr>      <chr>   <chr>       <chr>
## 1 yes      yes       no         no      no          no
## 2 yes      yes       no         no      no          yes
## 3 yes      yes       no         no      no          yes
## 4 yes      no        yes        no      no          yes
## 5 yes      no        no         no      no          yes
## # … with 12 more variables: productive_cough <chr>,
## #   dyspnoa <chr>, dysuria <chr>, myalgia <chr>, …
```

Use `mutate()` and `across()` to convert the variable levels to uppercase. (That is, "yes" to "YES" and "no" to "NO")

```
Q_febrile_disease_symptoms <-
  febrile_diseases %>%

  _____________________________
```

The `.fns` argument

Now, on to the second argument in `across()`. As mentioned above, this argument takes in the function to be applied across columns.

You can provide any valid function here. We had previously used `toupper()`:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = toupper))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    NO         NO
## 2        55 Male   2020-10-24    NO         NO
## 3        23 Male   2020-10-24    NO         NO
## 4        20 Female 2020-10-22    NO         NO
## 5        55 Female 2020-10-22    NO         NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

In a similar style, we can also use `tolower()`:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = tolower))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    no         no
## 2        55 Male   2020-10-24    no         no
## 3        23 Male   2020-10-24    no         no
## 4        20 Female 2020-10-22    no         no
## 5        55 Female 2020-10-22    no         no
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Of course, the function we apply through `across()` needs to be **type-appropriate**: it should apply to the type (character, numeric, factor, etc) of the variables we are feeding in.

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = log))
```

```
Error in `mutate()`:
! non-numeric argument to mathematical function
```

Here we get an error message because we tried to apply a function made for numeric variables to character type variables.

SIDE NOTE

It is a bit confusing to write a function without parentheses, as in `.fns = toupper`. There is a difference between `toupper()` and `toupper`.

`toupper()` calls the function, while `toupper` without parenthesis makes a reference to the function. With a reference to the function, `across()` will take care of calling it from its back-end code (the code that defines `across()`. We call it "back-end" because it's "in the back" and you cannot see it unless you go looking into it explicitly.)

PRACTICE (in RMD)

In the `febrile_diseases` dataset, ensure that all the columns from `abd_pain` to `splenomegaly`, indicating symptoms of patients, are in lower case. Apply `tolower()` across all these variables using `mutate()` and `across()`.

```
Q_febrile_disease_symptoms_to_lower <-
  febrile_diseases %>%
  ________________________________
```

Custom ("anonymous") functions

Sometimes it is useful to use a custom function, called a "lambda function" or "anonymous function". You will see more about functions in later lessons. The idea here is that you write your own operation which will be applied across your selected variables. The writing of these lambda functions has certain strict rules so pay attention to this as we go through several examples.

The `toupper` example we saw above can be rewritten with this syntax:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ toupper(.x)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
## 1         45 Female 2020-10-22     NO          NO
## 2         55 Male   2020-10-24     NO          NO
## 3         23 Male   2020-10-24     NO          NO
## 4         20 Female 2020-10-22     NO          NO
## 5         55 Female 2020-10-22     NO          NO
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

In the code `.fns = ~ toupper(.x)`, the tilda, `~`, introduces the lambda function, and the `.x` references each of the columns across which you are applying the function. The `.x` takes the columns one by one and "calls" the function on each one.

So overall, this code can be read as "apply `toupper()` to each of the symptom variables."

Here is another example, but with `tolower()`:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ tolower(.x)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    no         no
## 2        55 Male   2020-10-24    no         no
## 3        23 Male   2020-10-24    no         no
## 4        20 Female 2020-10-22    no         no
## 5        55 Female 2020-10-22    no         no
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

The pattern is quite simple once you get used to it.

Now, with this anonymous function syntax, it becomes very intuitive to use functions that take in multiple arguments.

For example, we could explicit the "No" and "Yes" by pasting into the string what they are referring to, in other words, symptoms:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ paste0(.x, " symptoms")))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever  symp_headache
##       <dbl> <chr>  <date>        <chr>       <chr>
## 1        45 Female 2020-10-22    No symptoms No symptoms
## 2        55 Male   2020-10-24    No symptoms No symptoms
## 3        23 Male   2020-10-24    No symptoms No symptoms
## 4         20 Female 2020-10-22    No symptoms No symptoms
## 5         55 Female 2020-10-22    No symptoms No symptoms
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

Or we could use `str_sub()`, a function that allows to keep a subset of your string:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ str_sub(.x, start = 1, end = 1)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    N          N
## 2        55 Male   2020-10-24    N          N
## 3        23 Male   2020-10-24    N          N
## 4        20 Female 2020-10-22    N          N
## 5        55 Female 2020-10-22    N          N
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

In our case our string values are "No" and "Yes" so we will make a substring with just their first letters ("N" and "Y") to have a **one letter encoding**.

Or we can recode the "Yes" and "No" entries in a different manner:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ if_else(.x == "Yes", "Has symptom", "Does not have
        symptom")))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever
##       <dbl> <chr>  <date>        <chr>
## 1        45 Female 2020-10-22    Does not have symptom
## 2        55 Male   2020-10-24    Does not have symptom
## 3        23 Male   2020-10-24    Does not have symptom
## 4        20 Female 2020-10-22    Does not have symptom
## 5        55 Female 2020-10-22    Does not have symptom
## # … with 12 more variables: symp_headache <chr>,
## #   symp_cough <chr>, symp_rhinitis <chr>, …
```

Now we have the "Yes" encoded as "Has symptom" and the "No" encoded as "Does not have symptom". These strings are longer but they are clearer in their meaning than just "Yes" vs. "No".

We could also recode the "Yes" and "No" to numeric values:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ if_else(.x == "Yes", 1, 2)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>             <dbl>         <dbl>
## 1        45 Female 2020-10-22             2             2
## 2        55 Male   2020-10-24             2             2
## 3        23 Male   2020-10-24             2             2
## 4        20 Female 2020-10-22             2             2
## 5        55 Female 2020-10-22             2             2
## # … with 11 more variables: symp_cough <dbl>,
## #   symp_rhinitis <dbl>, symp_sneezing <dbl>, …
```

Now we have "Yes" encoded as choice 1 and "No" as 2.

Note that you can chain several `mutate()` calls together:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ if_else(.x == "Yes", 1, 2))) %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                .fns = ~ case_when(.x == 1 ~ 1,
                                   .x == 2 ~ 0)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>             <dbl>         <dbl>
## 1        45 Female 2020-10-22             0             0
## 2        55 Male   2020-10-24             0             0
## 3        23 Male   2020-10-24             0             0
## 4        20 Female 2020-10-22             0             0
## 5        55 Female 2020-10-22             0             0
## # … with 11 more variables: symp_cough <dbl>,
## #   symp_rhinitis <dbl>, symp_sneezing <dbl>, …
```

Above, we first convert from "Yes" & "No" to the numeric values 1 and 2, to follow R indexing (R counts from 1 onwards). However, other programming language start their index at 0, such as Python (Python counts from 0 onwards). For many machine learning algorithms, your encoding should be 0 or 1, so this could be a useful conversion of the encoding of your data. We use `case_when()` to define which numerical value should be switched to 1 (TRUE, has symptoms) and which numerical value should be switched to 0 (FALSE, does not have symptoms).

PRACTICE (in RMD)

The columns from `abd_pain` to `splenomegaly` in the `febrile_diseases` dataset contain information on whether a patient had a specified symptom, recorded as “yes” or “no”.

PRACTICE (in RMD)

Use `mutate()`, `across()` and an anonymous function to convert the variable levels to numbers, with “yes” as 1 and “no” as 0.

```
Q_febrile_disease_symptoms_to_numeric <-
  febrile_diseases %>%
  ______________________________
```

PRACTICE (in RMD)

In the `diet` dataset, the columns from `retinol` to `zinc` give the number of milligrams of each nutrient consumed by the surveyed women in a day.

```
diet %>%
  select(retinol:zinc)
```

```
## # A tibble: 5 × 15
##   retinol alpha_catorene beta_catorene beta_cryptoxanthin
##     <dbl>          <dbl>         <dbl>              <dbl>
## 1   0.998         0.0273          1.58             0.141
## 2   3.53          0.114           3.66             0.0804
## 3   1.32          0.388          13.9              0.0117
## 4   1.08          0.0305         10.9              0.509
## 5   1.25          0.102           9.66             0.164
## # … with 11 more variables: vitamin_c <dbl>,
## #   vitamin_b2 <dbl>, vitamin_b3 <dbl>, vitamin_b6 <dbl>, …
```

Use `mutate()`, `across()` and an anonymous function to convert these values to grams (divide by 1000).

```
Q_diet_to_grams <-
  diet %>%
  ______________________________
```

Creating new columns with the `.names` argument

The examples we have seen so far all involved replacing existing columns.

But what if you want to create new columns instead?

To illustrate this, let’s create a smaller subset of `yao_symptoms`:

```
yao_symptoms_mini <-
  yao_symptoms %>%
  select(symp_fever, symp_headache, symp_cough)

yao_symptoms_mini
```

```
## # A tibble: 5 × 3
##   symp_fever symp_headache symp_cough
##   <chr>      <chr>         <chr>
## 1 No         No            No
## 2 No         No            No
## 3 No         No            No
## 4 No         No            No
## 5 No         No            No
```

Now, to convert all columns to uppercase, we would usually run:

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper))
```

```
## # A tibble: 5 × 3
##   symp_fever symp_headache symp_cough
##   <chr>      <chr>         <chr>
## 1 NO         NO            NO
## 2 NO         NO            NO
## 3 NO         NO            NO
## 4 NO         NO            NO
## 5 NO         NO            NO
```

This code modifies existing columns *in place*

(Remember that the default argument for `.cols` is `everything()`, so the above code modifies all columns in the dataset)

Now, if we instead want to make *new* columns that are uppercase, we can use the `.names` argument of `across()`

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper,
                .names = "{.col}_uppercase"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough symp_fever_uppercase
##   <chr>      <chr>         <chr>      <chr>
## 1 No         No            No         NO
## 2 No         No            No         NO
## 3 No         No            No         NO
## 4 No         No            No         NO
## 5 No         No            No         NO
## # … with 2 more variables: symp_headache_uppercase <chr>,
## #   symp_cough_uppercase <chr>
```

`{.col}` represents each of the old column names. The rest of the string. "_uppercase" is pasted together with the old column names. So the code `"{.col}_uppercase"` code can be read as "for each column, convert to uppercase and name it by pasting the existing lowercase column name with _uppercase."

Of course, we can input any arbitrary string:

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper,
                .names = "{.col}_BIG_LETTERS"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough symp_fever_BIG_LETTERS
##   <chr>      <chr>         <chr>      <chr>
## 1 No         No            No         NO
## 2 No         No            No         NO
## 3 No         No            No         NO
## 4 No         No            No         NO
## 5 No         No            No         NO
## # … with 2 more variables: symp_headache_BIG_LETTERS <chr>,
## #   symp_cough_BIG_LETTERS <chr>
```

If we want the text to come before the old column name, we can also do this:

```
yao_symptoms_mini %>%
  mutate(across(.fns = toupper,
                .names = "uppercase_{.col}"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough uppercase_symp_fever
##   <chr>      <chr>         <chr>      <chr>
## 1 No         No            No         NO
## 2 No         No            No         NO
## 3 No         No            No         NO
## 4 No         No            No         NO
## 5 No         No            No         NO
## # … with 2 more variables: uppercase_symp_headache <chr>,
## #   uppercase_symp_cough <chr>
```

More usefully, we can create a numeric version of these symptoms variables:

```
yao_symptoms_mini %>%
  mutate(across(.fns = ~ if_else(.x == "Yes", 1, 0),
                .names = "numeric_{.col}"))
```

```
## # A tibble: 5 × 6
##   symp_fever symp_headache symp_cough numeric_symp_fever
##   <chr>      <chr>         <chr>                   <dbl>
## 1 No         No            No                          0
## 2 No         No            No                          0
## 3 No         No            No                          0
## 4 No         No            No                          0
## 5 No         No            No                          0
## # … with 2 more variables: numeric_symp_headache <dbl>,
## #   numeric_symp_cough <dbl>
```

PRACTICE (in RMD)

Now you will convert again the columns from `abd_pain` to `splenomegaly` in the `febrile_diseases` dataset, on patient symptoms, into numerical values. But, you will create new columns named `numeric_abd_pain` to `numeric_splenomegaly` using the `.names` argument within `across()`.

```
Q_febrile_disease_symptoms_to_numeric_new_variables <-
  febrile_diseases %>%
  ______________________________
```

Using `across()` with `summarize()`

To get summary statistics over multiple variables it is often helpful to use `across()`.

Consider again the columns from `retinol` to `zinc` in the `diet` dataset, which indicate the number of milligrams of each nutrient consumed by surveyed Vietnamese women in a day:

```
diet %>%
  select(retinol:zinc)
```

```
## # A tibble: 5 × 15
##   retinol alpha_catorene beta_catorene beta_cryptoxanthin
##     <dbl>          <dbl>         <dbl>              <dbl>
## 1   0.998         0.0273          1.58             0.141
## 2   3.53          0.114           3.66             0.0804
## 3   1.32          0.388          13.9              0.0117
## 4   1.08          0.0305         10.9              0.509
## 5   1.25          0.102           9.66             0.164
## # … with 11 more variables: vitamin_c <dbl>,
## #   vitamin_b2 <dbl>, vitamin_b3 <dbl>, vitamin_b6 <dbl>, …
```

Imagine you wanted to find the average amount of each nutrient consumed. To do this the usual way, you would need to type:

```
diet %>%
  summarize(mean_retinol = mean(retinol),
            mean_alpha_catorene = mean(alpha_catorene),
            mean_beta_catorene = mean(beta_catorene),
            mean_vitamin_c = mean(vitamin_c),
            mean_vitamin_b2 = mean(vitamin_b2)
            # And on and on and on for 15 columns
            )
```

```
## # A tibble: 1 × 5
##   mean_retinol mean_alpha_catorene mean_beta_catorene
##          <dbl>               <dbl>              <dbl>
## 1         2.06               0.152               6.15
## # … with 2 more variables: mean_vitamin_c <dbl>,
## #   mean_vitamin_b2 <dbl>
```

Of course this is not very efficient.

But with `across()`, this can be done in just two lines:

```
diet %>%
  summarize(across(.cols = retinol:zinc,
                   .fns = mean))
```

```
## # A tibble: 1 × 15
##   retinol alpha_catorene beta_catorene beta_cryptoxanthin
##     <dbl>          <dbl>         <dbl>              <dbl>
## 1    2.06          0.152          6.15              0.210
## # … with 11 more variables: vitamin_c <dbl>,
## #   vitamin_b2 <dbl>, vitamin_b3 <dbl>, vitamin_b6 <dbl>, …
```

And recall that one of the primary benefits of `summarize()` is that it facilitates grouped summaries. Well, we can still use those here!

```
diet %>%
  group_by(age_group) %>%
  summarize(across(.cols = retinol:zinc,
                   .fns = mean))
```

```
## # A tibble: 4 × 16
##   age_group retinol alpha_catorene beta_catorene
##   <chr>       <dbl>          <dbl>         <dbl>
## 1 20-29        2.27          0.130          5.37
## 2 30-39        2.92          0.164          6.18
## # … with 12 more variables: beta_cryptoxanthin <dbl>,
## #   vitamin_c <dbl>, vitamin_b2 <dbl>, vitamin_b3 <dbl>, …
```

Beautiful! So much information extracted so easily.

Here we grouped the data by age group, then across all the nutrient variables, we calculated their mean by age group. It can be read as: "for the 40-49 years old age group, the mean consumption of retinol is roughly of 1.343 micrograms, which seems lower than for other age groups."

Let's see another example.

The columns from `is_drug_parac` to `is_drug_other` in the `yaounde` dataset indicate, as 1 or 0, whether or not a survey respondent was treated with the named drug:

```
yao_drugs <-
  yaounde %>%
  select(age_years, sex, date_surveyed, is_drug_parac:is_drug_other)

yao_drugs
```

```
## # A tibble: 5 × 12
##   age_years sex    date_surveyed is_drug_parac
##       <dbl> <chr>  <date>                <dbl>
## 1        45 Female 2020-10-22                1
## 2        55 Male   2020-10-24               NA
## 3        23 Male   2020-10-24               NA
## 4        20 Female 2020-10-22                0
## 5        55 Female 2020-10-22               NA
## # … with 8 more variables: is_drug_antibio <dbl>,
## #   is_drug_hydrocortisone <dbl>, …
```

How could we count the number of respondents who took each drug?

We can simply take the sum of each column selecting the columns intelligently and using the `sum()` function:

```
yao_drugs %>%
  summarize(across(.cols = starts_with("is_drug"),
                   .fns = sum))
```

```
## # A tibble: 1 × 9
##   is_drug_parac is_drug_antibio is_drug_hydrocortisone
##           <dbl>           <dbl>                  <dbl>
## 1            NA              NA                     NA
## # … with 6 more variables: is_drug_other_anti_inflam <dbl>,
## #   is_drug_antiviral <dbl>, is_drug_chloro <dbl>, …
```

Oh no! we get all NAs!

We were smart and selected all our columns using `starts_with()` but we forgot to consider that `sum()` has `na.rm` set to `FALSE` by default. We need to ensure the `na.rm` argument is set to TRUE.

The best way to do this is with lambda/anonymous function syntax:

```
yao_drugs %>%
  summarize(across(.cols = starts_with("is_drug"),
                   .fns = ~ sum(.x, na.rm = TRUE)))
```

```
## # A tibble: 1 × 9
##   is_drug_parac is_drug_antibio is_drug_hydrocortisone
##           <dbl>           <dbl>                  <dbl>
## 1           162              79                     14
## # … with 6 more variables: is_drug_other_anti_inflam <dbl>,
## #   is_drug_antiviral <dbl>, is_drug_chloro <dbl>, …
```

Again, we could also create a grouped summary:

```
yao_drugs %>%
  group_by(sex) %>%
  summarize(across(.cols = starts_with("is_drug"),
                   .fns = ~ sum(.x, na.rm = TRUE)))
```

```
## # A tibble: 2 × 10
##   sex    is_drug_parac is_drug_antibio is_drug_hydrocortis…¹
##   <chr>          <dbl>           <dbl>                 <dbl>
## 1 Female            93              42                     7
## 2 Male              69              37                     7
## # … with 6 more variables: is_drug_other_anti_inflam <dbl>,
## #   is_drug_antiviral <dbl>, is_drug_chloro <dbl>, …
```

This last code chunk counts the number of individuals, per sex (group by sex), who have received each drug (summing the number of people across each drug variable).

A final example.

Recall that the 13 columns from `symp_fever` to `symp_stomach_ache` in the `yao_symptoms` dataset indicate whether or not each respondent had a specific COVID-compatible symptom:

```
yao_symptoms
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>        <chr>      <chr>
## 1        45 Female 2020-10-22    No         No
## 2        55 Male   2020-10-24    No         No
## 3        23 Male   2020-10-24    No         No
## 4        20 Female 2020-10-22    No         No
## 5        55 Female 2020-10-22    No         No
## # … with 11 more variables: symp_cough <chr>,
## #   symp_rhinitis <chr>, symp_sneezing <chr>, …
```

How would we count the number of people with each symptom using `across()`.

We have two options.

Option 1: We could first `mutate()` the "Yes" and "No" to numeric values:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
                 .fns = ~ if_else(.x == "Yes", 1, 0)))
```

```
## # A tibble: 5 × 16
##   age_years sex    date_surveyed symp_fever symp_headache
##       <dbl> <chr>  <date>             <dbl>         <dbl>
## 1        45 Female 2020-10-22             0             0
## 2        55 Male   2020-10-24             0             0
## 3        23 Male   2020-10-24             0             0
## 4        20 Female 2020-10-22             0             0
## 5        55 Female 2020-10-22             0             0
## # … with 11 more variables: symp_cough <dbl>,
## #   symp_rhinitis <dbl>, symp_sneezing <dbl>, …
```

And then use `sum()` within `summarize()`:

```
yao_symptoms %>%
  mutate(across(.cols = symp_fever:symp_stomach_ache,
              .fns = ~ if_else(.x == "Yes", 1, 0))) %>%
  summarize(across(.cols = symp_fever:symp_stomach_ache,
                   .fns = sum))
```

```
## # A tibble: 1 × 13
##   symp_fever symp_headache symp_cough symp_rhinitis
##        <dbl>         <dbl>      <dbl>         <dbl>
## 1        143           135        130            89
## # … with 9 more variables: symp_sneezing <dbl>,
## #   symp_fatigue <dbl>, symp_muscle_pain <dbl>, …
```

Option 2: we could jump directly to `summarize()`, by summing with a condition:

```
yao_symptoms %>%
  summarize(across(.cols = symp_fever:symp_stomach_ache,
                   .fns = ~ sum(.x == "Yes")))
```

```
## # A tibble: 1 × 13
##   symp_fever symp_headache symp_cough symp_rhinitis
##        <int>         <int>      <int>         <int>
## 1        143           135        130            89
## # … with 9 more variables: symp_sneezing <int>,
## #   symp_fatigue <int>, symp_muscle_pain <int>, …
```

This code can be read as: across each symptom column, sum all individuals who have been recorded as receiving that drug (who have a "Yes" data entry for that drug).

PRACTICE

(in RMD)

In the `diet` data set, the variables `fao_fgw1` to `fao_fgw21` record the number of calories consumed from different FAO food groups. (FAO stands for "Food and Agricultural Organization"; the food groups are shown in Appendix 1.).

Use `summarize()` and `across()` to calculate the mean amount of calories obtained from each good group.

```
Q_diet_FAO_mean <-
  diet %>%
  ________________________________
```

PRACTICE

(in RMD)

In the `febrile_diseases` data set, the columns from `abd_pain` to `splenomegaly` in the `febrile_diseases` dataset contain information on whether a patient had a specified symptom, recorded as "yes" or "no". Use `summarize()`, `across()` to count the number of people with each symptom.

```
Q_febrile_disease_symptoms_count <-
  febrile_diseases %>%
  ________________________________
```

PRACTICE

(in RMD)

In the `yaounde` data set, calculate the median for the age, height, weight, number of bedridden days and numer of days off from work (i.e. from the variable `age_years` to the variable `n_bedridden_days`)

Use `summarize()` and `across()`, giving the `.fns` argument a lambda function to calculate the median. Careful ! A lambda function with the

PRACTICE (in RMD)

right arguments is indispensable, else you will have an `NA` median for some of the variables.

```
Q_yaounde_median <-
  yaounde %>%
  ________________________________
```

Multiple summary statistics

When we explored `summarize()` we rejoiced with the fact that we could calculate multiple summary statistics at the same time. This is also possible within `across()`.

Coming back to the diet data survey from Vietnam, we could calculate both the mean and the median across all the nutrient variables:

```
diet %>%
  summarise(across(.cols = retinol:zinc,
                   .fns = list(mean = mean,
                               median = median)))
```

```
## # A tibble: 1 × 30
##   retinol_mean retinol_median alpha_catorene_mean
##          <dbl>          <dbl>               <dbl>
## 1         2.06          0.724               0.152
## # … with 27 more variables: alpha_catorene_median <dbl>,
## #   beta_catorene_mean <dbl>, beta_catorene_median <dbl>, …
```

Here it is clear that on all numeric type variables of the data set, we want to calculate the mean and the median. We can do so by providing the `.fns` argument of `across()` with a list.

SIDE NOTE

Small joke: `.fns` isn't "functions" abbreviated plural for nothing ! If we could only apply one function within `across()`, it would have been named `.fn` (function singular abbreviated).

This time, for the naming, `across()` takes care of naming the resulting summary statistic columns. The syntax is : `list(desired_name_1 = function_1, desired_name_2 = function_2)`.

Let's see how you could control the naming even more, when operating on a list of functions:

```
diet %>%
  summarise(across(.cols = retinol:zinc,
                   .fns = list(average = mean, median = median),
                   .names = "{.fn}_{.col}"))
```

```
## # A tibble: 1 × 30
##   average_retinol median_retinol average_alpha_catorene
##             <dbl>          <dbl>                  <dbl>
## 1            2.06          0.724                  0.152
## # … with 27 more variables: median_alpha_catorene <dbl>,
## #   average_beta_catorene <dbl>, …
```

Here we reference the name of the function using `{.fn}` and the name of the column with `{.col}`. It is important to note that **both abbreviations are singular**! They are singular because they reference the function and the column one by one. Within the `across()` procedure, `across()` takes the functions and the columns one by one and for each one, takes the function name, such as `average`, and the column name, such as `retinol`, and makes the summary statistic `average_retinol` (i.e. `{.fn}=average` and `{.col}=retinol`).

As we are discussing mean, median, standard deviation calculations, we have to anticipate for `NA` values. Consider the code below:

```
diet %>%
  summarise(across(.cols = retinol:zinc,
                   .fns = list(average = ~ mean(.x, na.rm = TRUE),
                               median = ~ median(.x, na.rm = TRUE)),
                   .names = "{.fn}_{.col}"))
```

```
## # A tibble: 1 × 30
##   average_retinol median_retinol average_alpha_catorene
##             <dbl>          <dbl>                  <dbl>
## 1            2.06          0.724                  0.152
## # … with 27 more variables: median_alpha_catorene <dbl>,
## #   average_beta_catorene <dbl>, …
```

Here we have the same code as above, except we ensure that none of the means or medians will be `NA` by adding the `na.rm=TRUE` argument to the functions. For this, as we have seen above, we need to use the lambda/anonymous function style. Here we are giving the `.fns` argument a list of lambda functions.

PRACTICE

(in RMD)

In the `diet` data set, calculate the mean and the standard deviation for kilocalories, water, carbohydrates, fat, and protein consumed (i.e. from the variable `kilocalories_consumed` to the variable `carbs_consumed_grams`)

PRACTICE

(in RMD)

Use `summarize()` and `across()`, giving the `.fns` argument a list of the desired summary statistics. Make sure your means are named `COLUMN_NAME_mean` and your standard deviations are named `COLUMN_NAME_sd`.

```
Q_diet_food_composition_mean_sd <-
  diet %>%
  ______________________________
```

PRACTICE

(in RMD)

In the `febrile_diseases` data set, calculate the mean and the standard deviation for white blood cells,and all other blood analysis measurements (i.e. from the variable `wbc` to the variable `relymp_a`, seeing Appendix 2 for the detailed names of the variable name abbreviations)

Use `summarize()` and `across()`, giving the `.fns` argument a list of the desired summary statistics. Careful ! You need to give a list of lambda functions to calculate the mean and standard deviation, paying attention to the `na.rm` arguments, else your summary statistics will be set to `NA`.

Make sure your means are named `COLUMN_NAME_mean` and your standard deviations are named `COLUMN_NAME_sd`.

```
Q_febrile_diseases_mean_blood_composition <-
  febrile_diseases %>%
  ______________________________
```

Recap !

`across()` can be used inside many different {dplyr} verbs:

- `mutate(across(multiple_columns, function(s) to apply))`
- `summarize(across(multiple_columns, function(s) to apply))`

The statement defining multiple columns can be:

- a list of names e.g. `c(symp_fever, symp_headache, symp_cough)`
- a range of names e.g. `retinol:zinc`

The function(s) to apply across all columns can be:

- an existing function of R (such as `as.factor`, `mean` etc.)
- a custom (lambda/anonymous) function
- a list of existing functions (such as `list(mean = mean, sd = sd)`)
- a list of custom (lambda/anonymous) functions

Wrap up !

This was your first approach to `across()`: congrats for making it through ! Remember the power of combination of `across()` and other verbs. If you feel a summarizing or mutation operation is identical for more than one variable, then usually you should think of using `across()`.

In the upcoming lessons we will see some more data wrangling verbs: see you soon !

Contributors

The following team members contributed to this lesson:

LAURE VANCAUWENBERGHE
Data analyst, the GRAPH Network
A firm believer in science for good, striving to ally programming, health and education

KENE DAVID NWOSU
Data analyst, the GRAPH Network
Passionate about world improvement

References

Some material in this lesson was adapted from the following sources:

- *Summarise each group to fewer rows*. (n.d.). Retrieved 21 February 2022, from https://dplyr.tidyverse.org/reference/summarize.html
- *Create, modify, and delete columns — Mutate*. (n.d.). Retrieved 21 February 2022, from https://dplyr.tidyverse.org/reference/mutate.html
- *Apply a function (or functions) across multiple columns — Across*. (n.d.). Retrieved 21 February 2022, from https://dplyr.tidyverse.org/reference/across.html

Artwork was adapted from:

- Horst, A. (2022). *R & stats illustrations by Allison Horst*. https://github.com/allisonhorst/stats-illustrations (Original work published 2018)

Appendix 1: FAO Food Groups

code	meaning
fao_fgw1	Consumed amount from Foods made from grains
fao_fgw2	Consumed amount from White roots and tubers and plantain
fao_fgw3	Consumed amount from Pulses
fao_fgw4	Consumed amount from Nuts and seeds
fao_fgw5	Consumed amount from Milk and milk products
fao_fgw6	Consumed amount from Organ meat
fao_fgw7	Consumed amount from Meat and poultry
fao_fgw8	Consumed amount from Fish and seafood
fao_fgw9	Consumed amount from Eggs
fao_fgw10	Consumed amount from Dark green leafy vegetables
fao_fgw11	Consumed amount from Vitamin A-rich vegetables, roots and tubers
fao_fgw12	Consumed amount from Vitamin A-rich fruits
fao_fgw13	Consumed amount from Other vegetables
fao_fgw14	Consumed amount from Other fruits
fao_fgw15	Consumed amount from Insects and other small protein foods
fao_fgw16	Consumed amount from Other oils and fats
fao_fgw17	Consumed amount from Savoury and fried snacks
fao_fgw18	Consumed amount from Sweets
fao_fgw19	Consumed amount from Sugar sweetened beverages
fao_fgw20	Consumed amount from Condiments and seasonings
fao_fgw21	Consumed amount from Other beverages and foods

Appendix 2: Blood sample composition

Abbreviation	Complete Name
WBC	white bloodcell
RBC	red bloodcell
HGB	hemoglobin
PLT	platelet
NEUT_A	neutrophils
LYMP_A	lymphocytes
MONO_A	monocytes
EOSI_A	eosinophils
BASO_A	basophils
NRBC_A	nucleated red blood cells
IG_A	immature granulocytes
RET_A	reticulocytes
ASLYMP_A	antibody-synthesizing lymphocytes
RELYMP_A	reactive lymphocytes