

# Grouping and summarizing

## The GRAPH Courses team

October 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



Introduction	.....
Learning objectives	.....
The Yaounde COVID-19 dataset	.....
What are summary statistics?	.....
Introducing <code>dplyr::summarize()</code>	.....
Grouped summaries with <code>dplyr::group_by()</code>	.....
Grouping by multiple variables (nested grouping)	.....
Ungrouping with <code>dplyr::ungroup()</code> (why and how)	.....
Counting rows	.....
Counting rows that meet a condition	.....
<code>dplyr::count()</code>	.....
Including missing combinations in summaries	.....
Wrap-up	.....

---

## Introduction

You currently know how to keep your data entries of interest, how keep relevant variables and how to modify them or create new ones.

Now, we will take your data wrangling skills one step further by understanding how to easily extract summary statistics, through the verb `summarize()`, such as calculating the mean of a variable.

Moreover, we will begin exploring a crucial verb, `group_by()`, capable of grouping your variables together to perform grouped operations on your data set.

Let's go !

---

## Learning objectives

1. You can use `dplyr::summarize()` to extract summary statistics from datasets.
  2. You can use `dplyr::group_by()` to group data by one or more variables before performing operations on them.
  3. You understand why and how to ungroup grouped data frames.
  4. You can use `dplyr::n()` together with `group_by()-summarize()` to count rows per group.
  5. You can use `sum()` together with `group_by()-summarize()` to count rows that meet a condition.
  6. You can use `dplyr::count()` as a handy function to count rows per group.
- 

## The Yaounde COVID-19 dataset

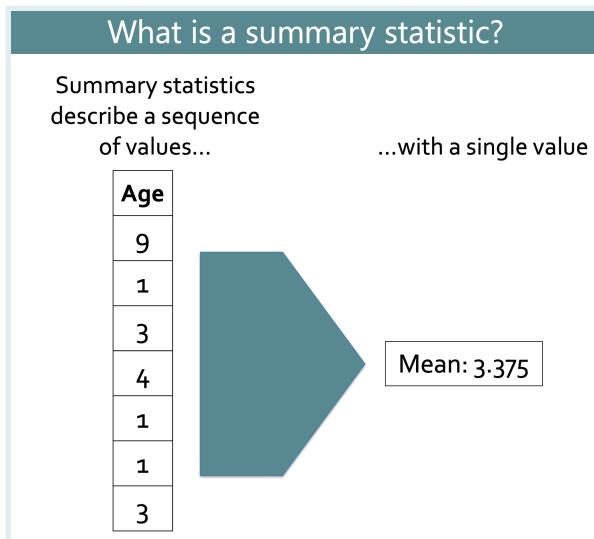
In this lesson, we will again use data from the COVID-19 serological survey conducted in Yaounde, Cameroon.

```
yaounde <- read_csv(here::here('data/yaounde_data.csv'))  
  
# A smaller subset of variables  
yao <- yaounde %>% select(  
  age, age_category_3, sex, weight_kg, height_cm,  
  neighborhood, is_smoker, is_pregnant, occupation,  
  treatment_combinations, symptoms, n_days_miss_work, n_bedridden_days,  
  highest_education, igg_result)  
  
yao  
  
## # A tibble: 971 × 15  
##   age age_category_3 sex    weight_kg height_cm neighborhood  
##   <dbl> <chr>       <chr>     <dbl>      <dbl> <chr>  
## 1    45 Adult        Female     95       169 Briqueterie  
## 2    55 Adult        Male      96       185 Briqueterie  
## 3    23 Adult        Male      74       180 Briqueterie  
## 4    20 Adult        Female    70       164 Briqueterie  
## 5    55 Adult        Female    67       147 Briqueterie  
## # ... with 966 more rows, and 9 more variables: is_smoker <chr>,  
## #   is_pregnant <chr>, occupation <chr>, ...
```

See the first lesson in this chapter for more information about this dataset.

## What are summary statistics?

A summary statistic is a single value (such as a mean or median) that describes a variable (typically a column in your dataset).



Summary statistics can describe the center, spread or range of a variable, or the counts and positions of values within that variable. Some common summary statistics are shown in the diagram below:

Examples of summary statistics

```
age <- (9, 1, 4, 2, 2, 2)
```

Summary statistic	R code	Output
<b>Counts</b>		
No. of elements	dplyr::n(age)	6
No. of distinct elements	dplyr::n_distinct(age)	4
<b>Position</b>		
First element	dplyr::first(age)	9
Last element	dplyr::last(age)	2
3rd element	dplyr::nth(age, 3)	4
<b>Center</b>		
Mean	mean(age)	3.3
Median	median(age)	2
<b>Spread</b>		
Standard deviation	sd(age)	2.9
Interquartile range	IQR(age)	1.5
<b>Range</b>		
Minimum	min(age)	1
Maximum	max(age)	9
25th quantile	quantile(age, 0.25)	2

Computing summary statistics is a very common operation in most data analysis workflows, so it will be important to become fluent in extracting them from your datasets. And for this task, there is no better tool than the `{dplyr}` function `summarize()`! So let's see how to use this powerful function.

## Introducing `dplyr::summarize()`

To get started, it is best to first consider how to get simple summary statistics *without* using `summarize()`, then we will consider why you *should* actually use it.

Imagine you were asked to find the mean age of respondents in the `yao` data frame. How might you do this?

First, recall that the dollar sign function, `$`, allows you to extract a data frame column to a vector:

```
yao$age %>%  
  head(10) # first 10 ages
```

```
## [1] 45 55 23 20 55 17 13 28 30 13
```

To obtain the mean, you simply pass this `yao$age` vector into the `mean()` function:

```
mean(yao$age)
```

```
## [1] 29.01751
```

And that's it! You now have a simple summary statistic. Extremely easy, right?

So why do we need `summarize()` to get summary statistics if the process is already so simple without it. We'll come back to this question soon.

First let's see *how* to obtain summary statistics with `summarize()`; we will then tackle the *why*. Going back to the previous example, the correct syntax to get the mean age with `summarize()` would be:

```
yao %>%  
  summarize(mean_age = mean(age))
```

```
## # A tibble: 1 × 1  
##   mean_age  
##       <dbl>  
## 1      29.0
```

The 'anatomy' of this syntax is shown below. You simply need the name of the new column (e.g. `mean_age`), the summary function (e.g. `mean()`), and the column to summarize (e.g. `age`).

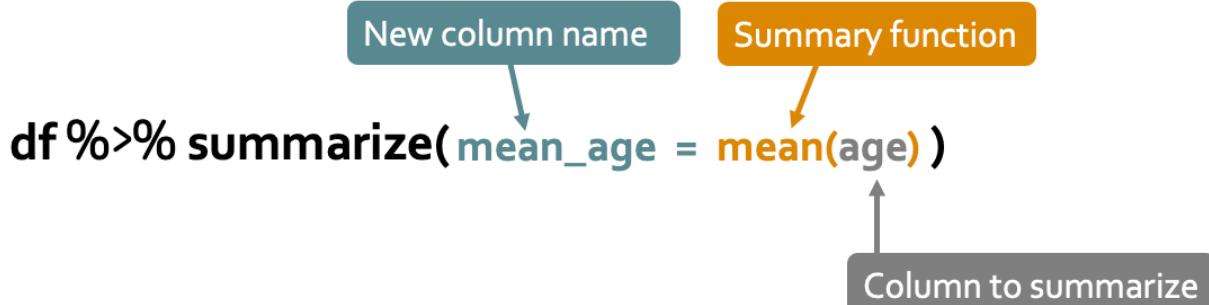


Fig. Basic syntax for the `summarize()` function.

You can also compute multiple summary statistics in a single `summarize()` statement. For example, if you wanted both the mean and the median age, you could run:

```
yao %>%
  summarize(mean_age = mean(age),
            median_age = median(age))
```

```
## # A tibble: 1 × 2
##   mean_age median_age
##       <dbl>      <dbl>
## 1     29.0       26
```

Nice and simple!

Now, you should be wondering why `summarize()` puts the summary statistics into a data frame, with each statistic in a different column.

The main benefit of this data frame structure is to make it easy to produce *grouped* summaries (and creating such grouped summaries will be the primary benefit of using `summarize()`).

We will look at these grouped summaries in the next section. For now, attempt the practice questions below.



Use `summarize()` and the relevant summary functions to obtain the mean, median and standard deviation of respondent weights from the `weight_kg` variable of the `yao` data frame.

Your output should be a data frame with three columns named as shown below:



mean\_weight\_kg median\_weight\_kg sd\_weight\_kg

```
Q_weight_summary <-  
  yao %>%
```



Use `summarize()` and the relevant summary functions to obtain the minimum and maximum respondent heights from the `height_cm` variable of the `yao` data frame.

Your output should be a data frame with two columns named as shown below:

min\_height\_cm max\_height\_cm

```
Q_height_summary <-  
  yao %>%
```

```
.CHECK_Q_height_summary()  
.HINT_Q_height_summary()
```

## Grouped summaries with `dplyr::group_by()`

As its name suggests, `dplyr::group_by()` lets you group a data frame by the values in a variable (e.g. male vs female sex). You can then perform operations that are split according to these groups.

What effect does `group_by()` have on a data frame? Let's try to group the `yao` data frame by sex and observe the effect:

```
yao %>%  
  group_by(sex)
```

```
## # A tibble: 971 × 15  
## # Groups:   sex [2]  
##       age age_category_3 sex    weight_kg height_cm neighborhood  
##     <dbl> <chr>        <chr>    <dbl>      <dbl> <chr>  
## 1     45 Adult         Female     95       169 Briqueterie  
## 2     55 Adult         Male      96       185 Briqueterie  
## 3     23 Adult         Male      74       180 Briqueterie  
## 4     20 Adult         Female    70       164 Briqueterie
```

```
## # ... with 966 more rows, and 9 more variables: is_smoker <chr>,
## #   is_pregnant <chr>, occupation <chr>, ...
```

Hmm. Apparently nothing happened. The one thing you *might* notice is a new section in the header that tells you the grouped-by variable (sex) and the number of groups (2):

```
# A tibble: 971 × 10
# Groups:   sex [2]
```

But apart from this header, the data frame appears unchanged.

Now watch what happens when we chain the `group_by()` with the `summarize()` call we used in the previous section:

```
yao %>%
  group_by(sex) %>%
  summarize(mean_age = mean(age),
            median_age = median(age))
```

```
## # A tibble: 2 × 3
##   sex     mean_age median_age
##   <chr>      <dbl>      <dbl>
## 1 Female     29.5       26
## 2 Male       28.4       25
```

You get a different summary statistic for each group! The statistics for women are in one row and those for men are in another. (From this output data frame, you can tell that, for example, the mean age for female respondents is 29.5, while that for male respondents is 28.4)

As was mentioned earlier, this kind of grouped summary is the primary reason the `summarize()` function is so useful!

---

Let's see another example of a simple `group_by() + summarize()` operation.

Suppose you were asked to obtain the maximum and minimum weights for individuals in different neighborhoods in the `yao` data frame. First you would `group_by()` the `neighborhood` variable, then call the `max()` and `min()` functions inside `summarize()`:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(max_weight = max(weight_kg),
            min_weight = min(weight_kg))
```

```
## # A tibble: 9 × 3
##   neighborhood max_weight min_weight
```

```

##   <chr>      <dbl>      <dbl>
## 1 Briqueterie    128       20
## 2 Carriere       129       14
## 3 Cité Verte     118       16
## 4 Ekoudou        135       15
## 5 Messa           96        19
## # ... with 4 more rows

```

Great! With just a few code lines you are able to extract quite a bit of information.

---

Let's see one more example for good measure. The variable `n_days_miss_work` tells us the number of days that respondents missed work due to COVID-like symptoms. Individuals who reported no COVID-like symptoms have an `NA` for this variable:

```

yao %>%
  select(n_days_miss_work)

```

```

## # A tibble: 971 × 1
##   n_days_miss_work
##   <dbl>
## 1 0
## 2 NA
## 3 NA
## 4 7
## 5 NA
## # ... with 966 more rows

```

To count the total number of work days missed for each sex group, you could try to run the `sum()` function on the `n_days_miss_work` variable:

```

yao %>%
  group_by(sex) %>%
  summarise(total_days_missed = sum(n_days_miss_work))

```

```

## # A tibble: 2 × 2
##   sex   total_days_missed
##   <chr>      <dbl>
## 1 Female      NA
## 2 Male        NA

```

Hmmm. This gives you `NA` results because some rows in the `n_days_miss_work` column have `NAs` in them, and R cannot find the sum of values containing an `NA`. To solve this, the argument `na.rm = TRUE` is needed:

```

yao %>%
  group_by(sex) %>%
  summarise(total_days_missed = sum(n_days_miss_work, na.rm = TRUE))

```

```
## # A tibble: 2 × 2
##   sex      total_days_missed
##   <chr>          <dbl>
## 1 Female           256
## 2 Male             272
```

The output tells us that across all women in the sample, 256 work days were missed due to COVID-like symptoms, and across all men, 272 days.

So hopefully now you see why `summarize()` is so powerful. In combination with `group_by()`, it lets you obtain highly informative grouped summaries of your datasets with very few lines of code.

Producing such summaries is a very important part of most data analysis workflows, so this learning is likely to come in handy soon!

### VOCAB



### `summarize()` produces “Pivot Tables”

The summary data frames created by `summarize()` are often called Pivot Tables in the context of spreadsheet software like Microsoft Excel.

### PRACTICE



(in RMD)

is_smoker	weight_mean
Ex-smoker	
Non-smoker	
Smoker	
NA	

```
Q_height_by_smoking_status <-
  yao %>%
```

Use `group_by()`, `summarize()`, and the relevant summary functions to obtain the minimum and maximum heights for each sex in the `yao` data frame.

**PRACTICE**  
  
(in RMD)

Your output should be a data frame with three columns named as shown below:

sex	min_height_cm	max_height_cm
Female		
Male		

```
Q_min_max_height_by_sex <-
  yao %>%
```

**PRACTICE**  
  
(in RMD)

Use `group_by()`, `summarize()`, and the `sum()` function to calculate the total number of bedridden days (from the `n_bedridden_days` variable) reported by respondents of each sex.

Your output should be a data frame with two columns named as shown below:

neighborhood	total_bedridden_days
Female	
Male	

```
Q_sum_bedridden_days <-
  yao %>%
```

## Grouping by multiple variables (nested grouping)

It is possible to group a data frame by more than one variable. This is sometimes called “nested” grouping.

Let’s see an example. Suppose you want to know the mean age of men and women *in each neighbourhood* (rather than the mean age of *all* women), you could put both `sex` and `neighborhood` in the `group_by()` statement:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 18 × 3
## # Groups:   sex [2]
##   sex     neighborhood mean_age
##   <chr>   <chr>          <dbl>
## 1 Female  Briqueterie    31.6
## 2 Female  Carriere       28.2
## 3 Female  Cité Verte     31.8
## 4 Female  Ekoudou        29.3
## 5 Female  Messa          30.2
## # ... with 13 more rows
```

From this output data frame you can tell that, for example, women from Briqueterie have a mean age of 31.6 years, while men from Briqueterie have a mean age of 33.7 years.

The order of the columns listed in `group_by()` is interchangeable. So if you run `group_by(neighborhood, sex)` instead of `group_by(sex, neighborhood)`, you'll get the same result, although it will be ordered differently:

```
yao %>%
  group_by(neighborhood, sex) %>%
  summarize(mean_age = mean(age))
```

```
## `summarise()` has grouped output by 'neighborhood'. You can override using
## the `.groups` argument.
```

```
## # A tibble: 18 × 3
## # Groups:   neighborhood [9]
##   neighborhood sex     mean_age
##   <chr>        <chr>      <dbl>
## 1 Briqueterie Female    31.6
## 2 Briqueterie Male     33.7
## 3 Carriere     Female    28.2
## 4 Carriere     Male     30.0
## 5 Cité Verte   Female    31.8
## # ... with 13 more rows
```

Now the column order is different: `neighborhood` is the first column, and `sex` is the second. And the row order is also different: rows are first ordered by `neighborhood`, then ordered by `sex` within each neighborhood.

But the actual summary statistics are the same. For example, you can again see that women from Briqueterie have a mean age of 31.6 years, while men from Briqueterie have a mean age of 33.7 years.



Using the `yao` data frame, group your data by gender (`sex`) and treatments (`treatment_combinations`) using `group_by`. Then, using `summarize()` and the relevant summary function, calculate the mean weight (`weight_kg`) for each group.

Your output should be a data frame with three columns named as shown below:

sex	treatment_combinations	mean_weight_kg
-----	------------------------	----------------

```
Q_weight_by_sex_treatments <-  
  yao %>%
```

Using the `yao` data frame, group your data by age category (`age_category_3`), gender (`sex`), and IgG results (`igg_results`) using `group_by`. Then, using `summarize()` and the relevant summary function, calculate the mean number of bedridden days (`n_bedridden_days`) for each group.

Your output should be a data frame with four columns named as shown below:

age_category	sex	igg_result	mean_n_bedridden_days
--------------	-----	------------	-----------------------

```
Q_bedridden_by_age_sex_iggresult <-  
  yao %>%
```

## Ungrouping with `dplyr::ungroup()` (why and how)

When you `group_by()` more than one variable before using `summarize()`, the output data frame is still grouped. This persistent grouping can have unwanted downstream effects, so you will sometimes need to use `dplyr::ungroup()` to ungroup the data before doing further analysis.

To understand *why* you should `ungroup()` data, first consider the following example, where we group by only one variable before summarizing:

```
yao %>%
  group_by(sex) %>%
  summarize(mean_age = mean(age))
```

```
## # A tibble: 2 × 2
##   sex     mean_age
##   <chr>    <dbl>
## 1 Female    29.5
## 2 Male      28.4
```

The data comes out like a normal data frame; it is not grouped. You can tell this because there is no information about groups in the header.

But now consider when you group by two variables before summarizing:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.`groups` argument.
```

```
## # A tibble: 18 × 3
## # Groups:   sex [2]
##   sex     neighborhood mean_age
##   <chr>    <chr>        <dbl>
## 1 Female  Briqueterie    31.6
## 2 Female  Carriere       28.2
## 3 Female  Cité Verte     31.8
## 4 Female  Ekoudou        29.3
## 5 Female  Messa          30.2
## # ... with 13 more rows
```

Now the header tells you that the data is still grouped by the first variable in `group_by()`, `sex`:

```
# A tibble: 18 × 3
👉# Groups:   sex [2]👉
```

What is the implication of this persistent grouping in the data frame? It means that the data frame may exhibit what seems like weird behavior when you try to apply some `{dplyr}` functions on it.

For example, if you try to `select()` a single variable, perhaps the `mean_age` variable, you should normally be able to just use `select(mean_age)`:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age)) %>%
  select(mean_age) # doesn't work as expected
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.groups` argument.
## Adding missing grouping variables: `sex`

## # A tibble: 18 × 2
## # Groups:   sex [2]
##   sex     mean_age
##   <chr>    <dbl>
## 1 Female    31.6
## 2 Female    28.2
## 3 Female    31.8
## 4 Female    29.3
## 5 Female    30.2
## # ... with 13 more rows
```

But as you can see, the grouped-by variable, `sex`, is *still* selected, even though we only asked for `mean_age` in the `select()` statement.

This is one of the many examples of unique behaviors of grouped data frames. Other `dplyr` verbs like `filter()`, `mutate()` and `arrange()` also act in special ways on grouped data. We will address this in detail in a future lesson.

---

So you now know *why* you should ungroup data when you no longer need it grouped. Let's now see *how* to ungroup data. It's quite simple: just add the `ungroup()` function to your pipe chain. For example:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age)) %>%
  ungroup()
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 18 × 3
##   sex     neighborhood mean_age
##   <chr>    <chr>          <dbl>
## 1 Female  Briqueterie    31.6
## 2 Female  Carriere       28.2
## 3 Female  Cité Verte     31.8
## 4 Female  Ekoudou        29.3
```

```
## 5 Female Messa           30.2
## # ... with 13 more rows
```

Now that the data frame is ungrouped, it will behave like a normal data frame again. For example, you can `select()` any column(s) you want; you won't have some unwanted columns tagging along:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(mean_age = mean(age)) %>%
  ungroup() %>%
  select(mean_age)
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.`groups` argument.
```

```
## # A tibble: 18 × 1
##   mean_age
##   <dbl>
## 1 31.6
## 2 28.2
## 3 31.8
## 4 29.3
## 5 30.2
## # ... with 13 more rows
```

---

## Counting rows

You can do a lot of data science by just *counting* and occasionally *dividing*. -  
Hadley Wickham, Chief Scientist at RStudio

A common data summarization task is counting how many observations (rows) there are for each group. You can achieve this with the special `n()` function from `{dplyr}`, which is specifically designed to be used within `summarise()`.

For example, if you want to count how many individuals are in each neighborhood group, you would run:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(count = n())
```

```
## # A tibble: 9 × 2
##   neighborhood count
##   <chr>        <int>
```

```
## 1 Briqueterie    106
## 2 Carriere        236
## 3 Cité Verte      72
## 4 Ekoudou         190
## 5 Messa            48
## # ... with 4 more rows
```

As you can see, the `n()` function does not require any arguments. It just “knows its job” in the data frame!

Of course, you can include other summary statistics in the same `summarize()` call. For example, below we also calculate the mean age per neighborhood.

```
yao %>%
  group_by(neighborhood) %>%
  summarize(count = n(),
            mean_age = mean(age))
```

```
## # A tibble: 9 × 3
##   neighborhood count mean_age
##   <chr>       <int>    <dbl>
## 1 Briqueterie    106     32.5
## 2 Carriere        236     28.9
## 3 Cité Verte      72      29.9
## 4 Ekoudou         190     27.6
## 5 Messa            48      27.3
## # ... with 4 more rows
```

Group your `yao` data frame by the respondents' occupation (`occupation`) and use `summarize()` to count the different occupations (think of the `n()` function) and calculate, by occupation, the mean number of work days missed (`n_days_miss_work`).

### PRACTICE



Your output should be a data frame with three columns named as shown below:

occupation	count	mean_n_days_miss_work
------------	-------	-----------------------

```
Q_occupation_summary <-
  yao %>%
```

### Counting rows that meet a condition

within the `sum()` function.

For example, to count the number of people under 18 in each neighborhood, you place the condition `age < 18` inside `sum()`:

```
yao %>%  
  group_by(neighborhood) %>%  
  summarize(count_under_18 = sum(age < 18))
```

```
## # A tibble: 9 × 2  
##   neighborhood count_under_18  
##   <chr>           <int>  
## 1 Briqueterie      28  
## 2 Carriere          58  
## 3 Cité Verte        19  
## 4 Ekoudou          66  
## 5 Messa             18  
## # ... with 4 more rows
```

---

Similarly, to count the number of people with doctorate degrees in each neighborhood, you place the condition `highest_education == "Doctorate"` inside `sum()`:

```
yao %>%  
  group_by(neighborhood) %>%  
  summarize(count_with_doctorates = sum(highest_education == "Doctorate"))
```

```
## # A tibble: 9 × 2  
##   neighborhood count_with_doctorates  
##   <chr>           <int>  
## 1 Briqueterie      2  
## 2 Carriere          1  
## 3 Cité Verte        1  
## 4 Ekoudou          1  
## 5 Messa             2  
## # ... with 4 more rows
```

### Under the hood: counting with conditions



Why are you able to use `sum()` which is meant to add numbers, on a condition like `highest_education == "Doctorate"`?

Using `sum()` on a condition works because the condition evaluates to the Boolean values `TRUE` and `FALSE`. And these Boolean values are treated as numbers (where `TRUE` equals 1 and `FALSE` equals 0), and numbers can, of course, be summed.

The code below demonstrates what is going on under the hood in a step-by-step way. Run through it and see if you can follow.

```
demo_of_condition_sums <- yao %>%
  select(highest_education) %>%
  mutate(with_doctorate = highest_education == "Doctorate") %>%
  mutate(numeric_with_doctorate = as.numeric(with_doctorate))

demo_of_condition_sums
```

**CHALLENGE**



```
## # A tibble: 971 × 3
##   highest_education with_doctorate numeric_with_doctorate
##   <chr>           <lgl>                <dbl>
## 1 Secondary        FALSE                 0
## 2 University       FALSE                 0
## 3 University       FALSE                 0
## 4 Secondary        FALSE                 0
## 5 Primary          FALSE                 0
## # ... with 966 more rows
```

The numeric values can then be added to produce a count of rows fulfilling the condition `highest_education == "Doctorate"`:

```
demo_of_condition_sums %>%
  summarize(count_with_doctorate = sum(numeric_with_doctorate))
```

```
## # A tibble: 1 × 1
##   count_with_doctorate
##   <dbl>
## 1 17
```

For a final illustration of counting with conditions, consider the `treatment_combinations` variable, which lists the treatments received by people with COVID-like symptoms. People who received no treatments have an `NA` value:

```
yao %>%
  select(treatment_combinations)
```

```
## # A tibble: 971 × 1
##   treatment_combinations
##   <chr>
## 1 Paracetamol
## 2 <NA>
```

```
## 3 <NA>
## 4 Antibiotics
## 5 <NA>
## # ... with 966 more rows
```

If you want to count the number of people who received *no treatment*, you would sum up those who meet the `is.na(treatment_combinations)` condition:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(unknown_treatments = sum(is.na(treatment_combinations)))
```

```
## # A tibble: 9 × 2
##   neighborhood unknown_treatments
##   <chr>                <int>
## 1 Briqueterie            82
## 2 Carriere                192
## 3 Cité Verte                 46
## 4 Ekoudou                133
## 5 Messa                     35
## # ... with 4 more rows
```

These are the people with NA values for the `treatment_combinations` column.

To count the people who *did* receive some treatment, you can simply negate the `is.na()` function with `!`:

```
yao %>%
  group_by(neighborhood) %>%
  summarize(known_treatments = sum(!is.na(treatment_combinations)))
```

```
## # A tibble: 9 × 2
##   neighborhood known_treatments
##   <chr>                <int>
## 1 Briqueterie            24
## 2 Carriere                44
## 3 Cité Verte                 26
## 4 Ekoudou                57
## 5 Messa                     13
## # ... with 4 more rows
```



Group your `yao` data frame by the respondents' symptoms (`symptoms`) and count how many respondents have these symptoms, **only within the adult respondents** (i.e. within the `sum()` function, set a condition on the `age_category_3` variable).

**PRACTICE****(in RMD)**

Your output should be a data frame with two columns named as shown below:

symptoms	sum_adults
----------	------------

```
Q_symptoms_adults <-  
  yao %>%
```

## dplyr::count()

The `dplyr::count()` function wraps a bunch of things into one beautiful friendly line of code to help you find counts of observations by group.

Let's use `dplyr::count()` on our `occupation` variable:

```
yao %>%  
  count(occupation)
```

```
## # A tibble: 28 × 2  
##   occupation          n  
##   <chr>              <int>  
## 1 Farmer                  5  
## 2 Farmer--Other            1  
## 3 Home-maker                65  
## 4 Home-maker--Farmer        2  
## 5 Home-maker--Informal worker    3  
## # ... with 23 more rows
```

Note that this is the same output as:

```
yao %>%  
  group_by(occupation) %>%  
  summarize(n = n())
```

```
## # A tibble: 28 × 2  
##   occupation          n  
##   <chr>              <int>  
## 1 Farmer                  5  
## 2 Farmer--Other            1  
## 3 Home-maker                65  
## 4 Home-maker--Farmer        2  
## 5 Home-maker--Informal worker    3  
## # ... with 23 more rows
```

You can also apply `dplyr::count()` in a nested fashion:

```
yao %>%
  count(sex, occupation)
```

```
## # A tibble: 40 × 3
##   sex     occupation      n
##   <chr>   <chr>        <int>
## 1 Female  Farmer          3
## 2 Female  Home-maker      65
## 3 Female  Home-maker--Farmer  2
## 4 Female  Home-maker--Informal worker  3
## 5 Female  Home-maker--Informal worker--Farmer  1
## # ... with 35 more rows
```

The `count()` verb gives you key information about your dataset in a very quick manner. Let's look at our IgG results stratified by age category and sex in one line of code.

Using the `yao` data frame, count the different combinations of gender (`sex`), age categories (`age_category_3`) and IgG results (`igg_results`).

Your output should be a data frame with four columns named as shown below:



```
Q_count_iggresults_stratified_by_sex_agescategories <-
  yao %>%
```

Using the `yao` data frame, count the different combinations of age categories (`age_category_3`) and number of bedridden days (`n_bedridden_days`).

Your output should be a data frame with three columns named as shown below:

```
Q_count_bedridden_age_categories <-
  yao %>%
```

The downside of `count()` is that it can only give you a single summary statistic in the data frame. When you use `summarize()` and `n()` you can include multiple summary statistics. For example:

```
yao %>%
  group_by(sex, neighborhood) %>%
  summarize(count = n(),
            median_age = median(age))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 18 × 4
## # Groups:   sex [2]
##   sex     neighborhood count median_age
##   <chr>   <chr>      <int>     <dbl>
## 1 Female  Briqueterie    61      28
## 2 Female  Carriere       140     25.5
## 3 Female  Cité Verte     44      28
## 4 Female  Ekoudou        110     26.5
## 5 Female  Messa          26      27.5
## # ... with 13 more rows
```

But `count()` can only yield counts:

```
yao %>%
  group_by(sex, neighborhood) %>%
  count()
```

```
## # A tibble: 18 × 3
## # Groups:   sex, neighborhood [18]
##   sex     neighborhood     n
##   <chr>   <chr>      <int>
## 1 Female  Briqueterie    61
## 2 Female  Carriere       140
## 3 Female  Cité Verte     44
## 4 Female  Ekoudou        110
## 5 Female  Messa          26
## # ... with 13 more rows
```

## Including missing combinations in summaries

When you use `group_by()` and `summarize()` on multiple variables, you obtain a summary statistic for every unique combination of the grouped variables. For instance,

consider the code and output below, which counts the number of individuals in each age-sex group:

```
yao %>%
  group_by(sex, age_category_3) %>%
  summarise(number_of_individuals = n())
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `groups` argument.
```

```
## # A tibble: 6 × 3
## # Groups:   sex [2]
##   sex     age_category_3 number_of_individuals
##   <chr>   <chr>           <int>
## 1 Female  Adult            368
## 2 Female  Child             155
## 3 Female  Senior            26
## 4 Male    Adult            267
## 5 Male    Child             136
## # ... with 1 more row
```

In the output data frame, there is one row for each combination of sex and age group (Female–Adult, Female–Child and so on).

But what happens if one of these combinations is not present in the data?

Let's create an artificial example to observe this. With the code below, we artificially drop all male children from the `yao` data frame:

```
yao_no_male_children <-
yao %>%
  filter(!(sex == "Male" & age_category_3 == "Child"))
```

Now if you run the same `group_by()` and `summarize()` call on `yao_no_male_children`, you'll notice the missing combination:

```
yao_no_male_children %>%
  group_by(sex, age_category_3) %>%
  summarise(number_of_individuals = n())
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `groups` argument.
```

```
## # A tibble: 5 × 3
## # Groups:   sex [2]
##   sex     age_category_3 number_of_individuals
##   <chr>   <chr>           <int>
```

```

## 1 Female Adult          368
## 2 Female Child          155
## 3 Female Senior          26
## 4 Male   Adult          267
## 5 Male   Senior           19

```

Indeed, there is no row for male children.

But sometimes it is useful to include such missing combinations in the output data frame, with an `NA` or `0` value for the summary statistic.

To do this, you can run the following code instead:

```

yao_no_male_children %>%
  # convert variables to factors
  mutate(sex = as.factor(sex),
         age_category_3 = as.factor(age_category_3)) %>%
  # Note the .drop = FALSE argument
  group_by(sex, age_category_3, .drop = FALSE) %>%
  summarise(number_of_individuals = n())

```

```

## `summarise()` has grouped output by 'sex'. You can override using the
## `groups` argument.

```

```

## # A tibble: 6 × 3
## # Groups:   sex [2]
##   sex    age_category_3 number_of_individuals
##   <fct>  <fct>                <int>
## 1 Female Adult              368
## 2 Female Child              155
## 3 Female Senior              26
## 4 Male   Adult              267
## 5 Male   Child               0
## # ... with 1 more row

```

What does the code do?

- First it converts the grouping variables to factors with `as.factor()` (inside a `mutate()` call)
- Then it uses the argument `.drop = FALSE` in the `group_by()` function to avoid dropping the missing combinations.

Now you have a clear `0` count for the number of male children!

Let's see one more example, this time without artificially modifying our data.

The code below calculates the average age by sex and education group:

```
yao %>%
  group_by(sex, highest_education) %>%
  summarise(mean_age = mean(age))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 13 × 3
## # Groups:   sex [2]
##   sex   highest_education     mean_age
##   <chr> <chr>             <dbl>
## 1 Female Doctorate          28
## 2 Female No formal instruction 45.6
## 3 Female No response        35
## 4 Female Primary            26.8
## 5 Female Secondary          28.8
## # ... with 8 more rows
```

Notice that in the output data frame, there are 7 rows for men but only 6 rows for women, because no woman answered “Other” to the question on highest education level.

If you nonetheless want to include the “Female–Other” row in the output data frame, you would run:

```
yao %>%
  mutate(sex = as.factor(sex),
        highest_education = as.factor(highest_education)) %>%
  group_by(sex, highest_education, .drop = FALSE) %>%
  summarise(mean_age = mean(age))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 14 × 3
## # Groups:   sex [2]
##   sex   highest_education     mean_age
##   <fct> <fct>             <dbl>
## 1 Female Doctorate          28
## 2 Female No formal instruction 45.6
## 3 Female No response        35
## 4 Female Other              NaN
## 5 Female Primary            26.8
## # ... with 9 more rows
```

## PRACTICE





Using the `yao` data frame, let's calculate the median age when grouping by neighborhood, `age_category`, and gender

Note, we want all possible combinations of these three variables (not just those present in our data).

Pay attention to two data wrangling imperatives! - convert your grouping variables to factors beforehand using `mutate()` - calculate your statistic, the median, while removing any `NA` values.

Your output should be a data frame with four columns named as shown below:

neighborhood	age_category_3	sex	median_age
--------------	----------------	-----	------------

```
Q_median_age_by_neighborhood_agecategory_sex <-
  yao %>%
```

### Why include missing combinations?

Above, we mentioned that including missing combinations is often useful in the data analysis workflow. Let's see one use case: plotting with `{ggplot}`. If you have not yet learned `{ggplot}`, that is okay, just focus on the plot outputs.

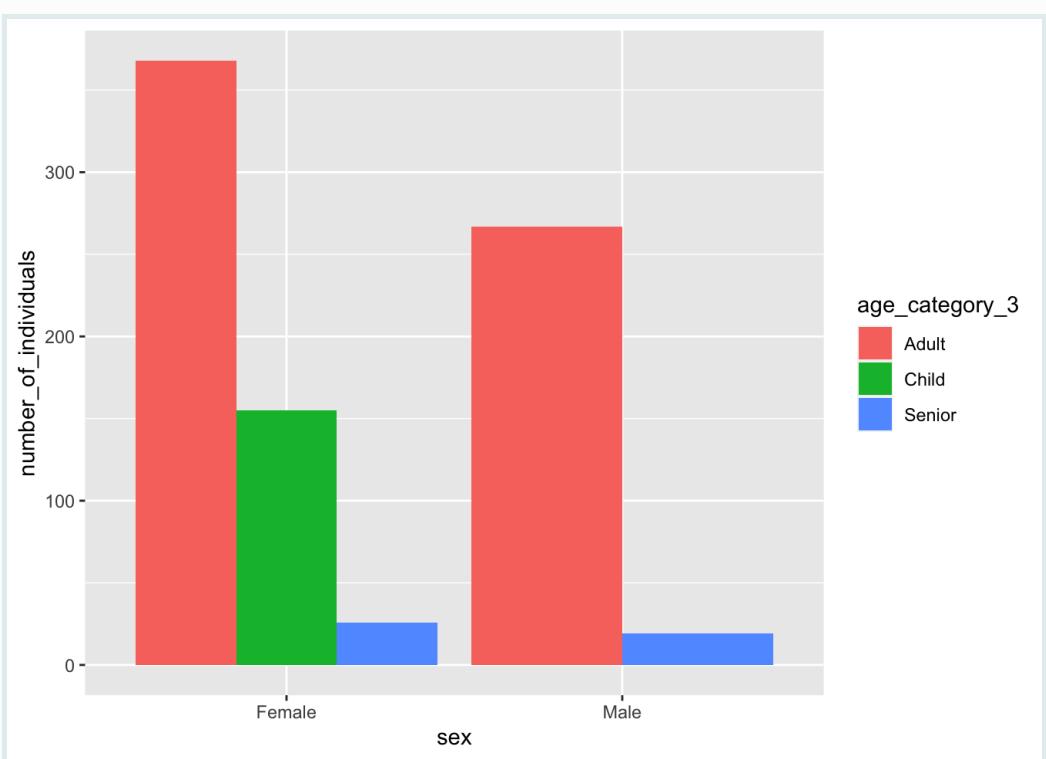
To make a dodged bar chart with the age-sex counts of `yao_no_male_children`, you could run:



```
yao_no_male_children %>%
  group_by(sex, age_category_3) %>%
  summarise(number_of_individuals = n()) %>%
  ungroup() %>%

  # pass the output to ggplot
  ggplot() +
  geom_col(aes(x = sex, y = number_of_individuals, fill =
    age_category_3),
    position = "dodge")
```

```
## `summarise()` has grouped output by 'sex'. You can override
using the
## ``.groups` argument.
```



#### SIDE NOTE



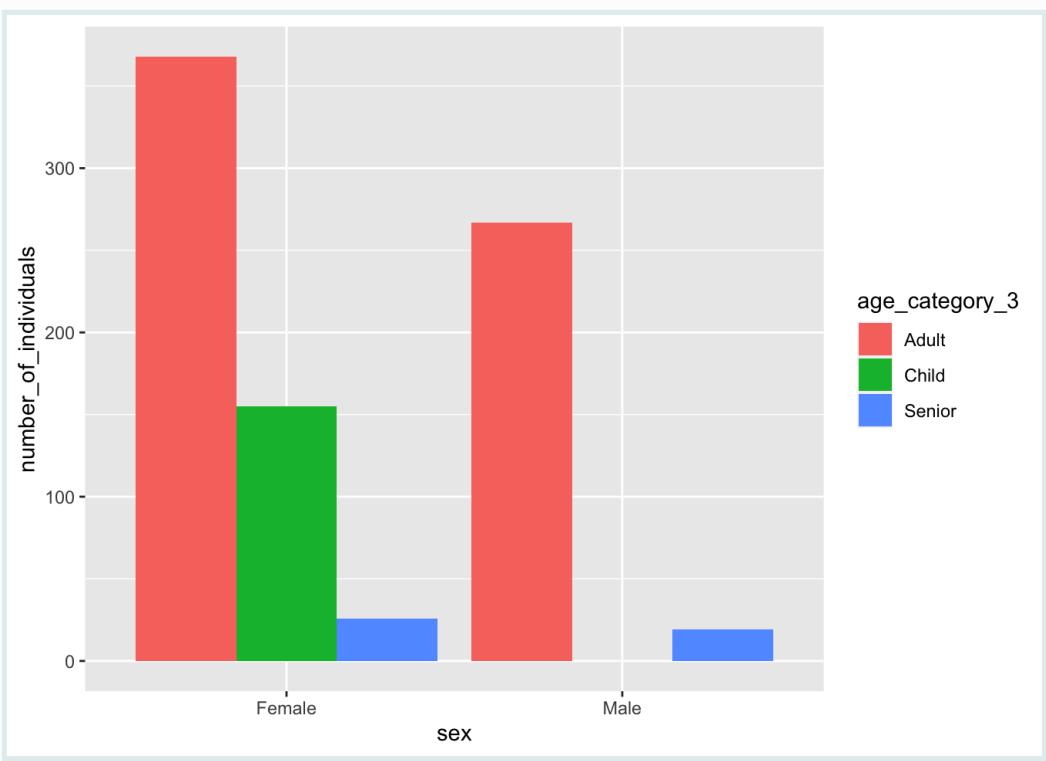
Not very elegant! Ideally there should be an empty space indicating 0 for the number of male children.

If you instead implement the procedure to include missing combinations, you get a more natural dodged bar plot, with an empty space for male children:

```
yao_no_male_children %>%
  mutate(sex = as.factor(sex),
        age_category_3 = as.factor(age_category_3)) %>%
  group_by(sex, age_category_3, .drop = FALSE) %>%
  summarise(number_of_individuals = n()) %>%
  ungroup() %>%

  # pass the output to ggplot
  ggplot() +
  geom_col(aes(x = sex, y = number_of_individuals, fill =
    age_category_3),
    position = "dodge")
```

## `summarise()` has grouped output by 'sex'. You can override using the  
## `groups` argument.



**SIDE NOTE** Much better!

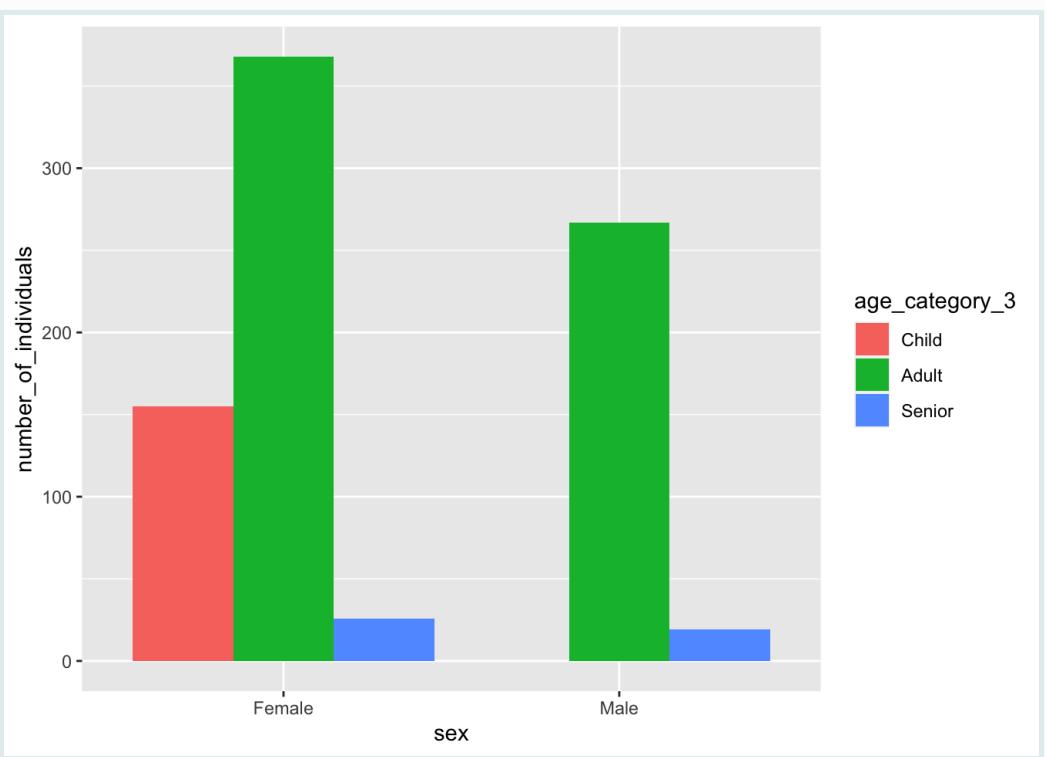


By the way, this output can be improved slightly by setting the factor levels for age to their proper ascending order: first "Child", then "Adult" then "Senior":

```
yao_no_male_children %>%
  mutate(sex = as.factor(sex),
        age_category_3 = factor(age_category_3,
                                 levels = c("Child",
                                           "Adult",
                                           "Senior"))) %>%
  group_by(sex, age_category_3, .drop = FALSE) %>%
  summarise(number_of_individuals = n()) %>%
  ungroup() %>%

  # pass the output to ggplot
  ggplot() +
  geom_col(aes(x = sex, y = number_of_individuals, fill =
                age_category_3),
           position = "dodge")
```

## `summarise()` has grouped output by 'sex'. You can override using the  
## `groups` argument.

**SIDE NOTE**

## Wrap-up

You have now seen how to obtain quick summary statistics from your data, either for exploratory data or for further data presentation or plotting.

Additionally, you have discovered one of the marvels of {dplyr}, the possibility to group your data using `group_by()`.

`group_by()` combined with `summarize()` is a one of the most common grouping manipulations.

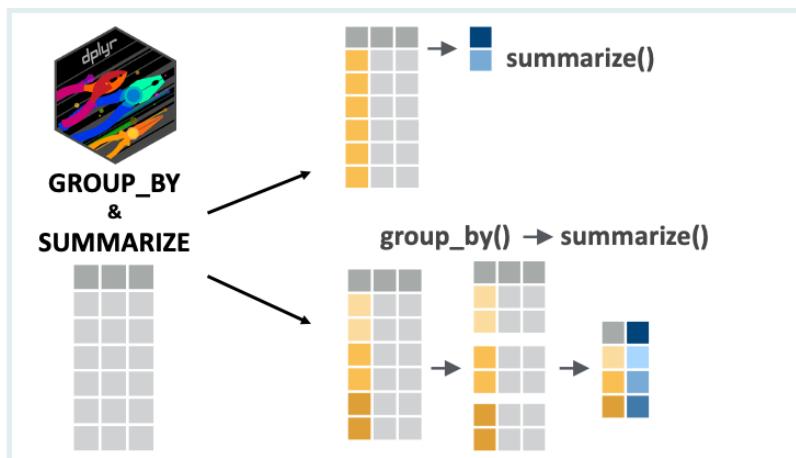


Fig: `summarize()` and `group_by()`

However, you can also combine `group_by()` with many of the other `{dplyr}` verbs: this is what we will cover in our next lesson. 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



### ANDREE VALLE CAMPOS

R Developer and Instructor, the GRAPH Network  
Motivated by reproducible science and education



### KENE DAVID NWOSU

Data analyst, the GRAPH Network  
Passionate about education

## References

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

- Horst, A. (2022). *Dplyr-learnr*. <https://github.com/allisonhorst/dplyr-learnr> (Original work published 2020)
- *Group by one or more variables*. (n.d.). Retrieved 21 February 2022, from [https://dplyr.tidyverse.org/reference/group\\_by.html](https://dplyr.tidyverse.org/reference/group_by.html)
- *Summarise each group to fewer rows*. (n.d.). Retrieved 21 February 2022, from <https://dplyr.tidyverse.org/reference/summarize.html>
- The Carpentries. (n.d.). *Grouped operations using `dplyr`*: Grouped operations using `dplyr` - Introduction to R/tidyverse for Exploratory Data Analysis. Retrieved July 28, 2022, from [https://tavareshugo.github.io/r-intro-tidyverse-gapminder/06-grouped\\_operations\\_dplyr/index.html](https://tavareshugo.github.io/r-intro-tidyverse-gapminder/06-grouped_operations_dplyr/index.html)

Artwork was adapted from:

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