Joining 1

GRAPH Network & WHO, supported by the Global Fund to fight HIV, TB & Malaria

October 2023

This document is a draft of a lesson made by the GRAPH Network, a non-profit headquartered at the University of Geneva Global Health Institute, in collaboration with the World Health Organization, under a Global Fund 2023 grant to create e-learning modules to build in-country data capacity for epidemiological and impact analysis for National HIV, TB and malaria programs

Prelude
Learning Objectives
Packages
What is a join and why do we need it?
Joining syntax
Types of joins
  left_join()
  right_join()
  inner_join()
  full_join()
Summary

Prelude

Joining datasets is a crucial skill when working with health-related data as it allows you to combine information from multiple sources, leading to more comprehensive and insightful analyses. In this lesson, you'll learn how to use different joining techniques using R's `dplyr` package. Let's get started!

Learning Objectives

- You understand how each of the different `dplyr` joins work
- You're able to choose the appropriate join for your data
- You can join simple datasets together using functions from `dplyr`

Packages

Please load the packages needed for this lesson with the code below:

```r
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
               countrycode)
```

What is a join and why do we need it?

To illustrate the utility of joins, let's start with a toy example. Consider the following two datasets. The first, `demographic`, contains names and ages of three patients:

```
demographic <-
  tribble(~name,     ~age,
          "Alice",    25,
          "Bob",      32,
          "Charlie",  45)
demographic
```

```
## # A tibble: 3 × 2
##   name      age
##   <chr>   <dbl>
## 1 Alice      25
## 2 Bob        32
## 3 Charlie    45
```

The second, `test_info`, contains tuberculosis test dates and results for those patients:

```
test_info <-
  tribble(~name,     ~test_date,    ~result,
          "Alice",   "2023-06-05",  "Negative",
          "Bob",     "2023-08-10",  "Positive",
          "Charlie", "2023-07-15",  "Negative")
test_info
```

```
## # A tibble: 3 × 3
##   name    test_date  result
##   <chr>   <chr>      <chr>
## 1 Alice   2023-06-05 Negative
## 2 Bob     2023-08-10 Positive
## 3 Charlie 2023-07-15 Negative
```

We'd like to analyze these data together, and so we need a way to combine them.

One option we might consider is the `cbind()` function from base R (`cbind` is short for column bind):

```
cbind(demographic, test_info)
```

name	age	name	test_date	result
Alice	25	Alice	2023-06-05	Negative
Bob	32	Bob	2023-08-10	Positive
Charlie	45	Charlie	2023-07-15	Negative

This successfully merges the datasets, but it doesn't do so very intelligently. The function essentially "pastes" or "staples" the two tables together. So, as you can notice, the "name" column appears twice. This is not ideal and will be problematic for analysis.

Another problem occurs if the rows in the two datasets are not already aligned. In this case, the data will be combined incorrectly with `cbind()`. Consider the `test_info_disordered` dataset below, which now has Bob in the first row:

```
test_info_disordered <-
  tribble(~name,      ~test_date,     ~result,
          "Bob",      "2023-08-10",   "Positive", # Bob in first row
          "Alice",    "2023-06-05",   "Negative",
          "Charlie",  "2023-07-15",   "Negative")
```

What happens if we `cbind()` this with the original `demographic` dataset, where Bob was in the *second* row?

```
cbind(demographic, test_info_disordered)
```

name	age	name	test_date	result
Alice	25	Bob	2023-08-10	Positive
Bob	32	Alice	2023-06-05	Negative
Charlie	45	Charlie	2023-07-15	Negative

Alice's demographic details are now mistakenly aligned with Bob's test info!

A third issue arises when an entity appears more than once in one dataset. Perhaps Alice did multiple TB tests:

```
test_info_multiple <-
  tribble(~name,      ~test_date,     ~result,
          "Alice",    "2023-06-05",   "Negative",
          "Alice",    "2023-06-06",   "Negative",
          "Bob",      "2023-08-10",   "Positive",
          "Charlie",  "2023-07-15",   "Negative")
```

If we try to `cbind()` this with the `demographic` dataset, we'll get an error, due to a mismatch in row counts:

```
cbind(demographic, test_info_multiple)
```

```
Error in data.frame(..., check.names = FALSE) :
  arguments imply differing number of rows: 3, 4
```

VOCAB

What we have here is called a `one-to-many` relationship—one Alice in the demographic data, but multiple Alice rows in the test data. Joining in

VOCAB

such cases will be covered in detail in the second joining lesson.

Clearly, we need a smarter way to combine datasets than `cbind()`; we'll need to venture into the world of joining.

Let's start with the most common join, the `left_join()`, which solves the problems we previously encountered.

It works for the simple case, and it does not duplicate the name column:

```
left_join(demographic, test_info)
```

```
## Joining with `by = join_by(name)`
```

```
## # A tibble: 3 × 4
##   name      age test_date  result
##   <chr>   <dbl> <chr>      <chr>
## 1 Alice      25 2023-06-05 Negative
## 2 Bob        32 2023-08-10 Positive
## 3 Charlie    45 2023-07-15 Negative
```

It works where the datasets are not ordered identically:

```
left_join(demographic, test_info_disordered)
```

```
## Joining with `by = join_by(name)`
```

```
## # A tibble: 3 × 4
##   name      age test_date  result
##   <chr>   <dbl> <chr>      <chr>
## 1 Alice      25 2023-06-05 Negative
## 2 Bob        32 2023-08-10 Positive
## 3 Charlie    45 2023-07-15 Negative
```

And it works when there are multiple test rows per patient:

```
left_join(demographic, test_info_multiple)
```

```
## Joining with `by = join_by(name)`
## # A tibble: 4 × 4
##   name      age test_date  result  
##   <chr>   <dbl> <chr>      <chr>   
## 1 Alice      25 2023-06-05 Negative
## 2 Alice      25 2023-06-06 Negative
## 3 Bob        32 2023-08-10 Positive
## 4 Charlie    45 2023-07-15 Negative
```

Simple yet beautiful!

SIDE NOTE

We'll be using the pipe operator as well when joining. Remember that this:

```
demographic %>% left_join(test_info)
```

```
## Joining with `by = join_by(name)`
```

is equivalent to this:

```
left_join(demographic, test_info)
```

```
## Joining with `by = join_by(name)`
```

Joining syntax

Now that we understand *why* we need joins, let's look at their basic syntax.

Joins take two dataframes as the first two arguments: `x` (the *left* dataframe) and `y` (the *right* dataframe). As with other R functions, you can provide these as named or unnamed arguments:

```
# both the same:
left_join(x = demographic, y = test_info) # named
left_join(demographic, test_info) # unnamed
```

Another critical argument is `by`, which indicates the column or **key** used to connect the tables. We don't always need to supply this argument; it can be *inferred* from the datasets. For example, in our original examples, "name" is the only column common to `demographic` and `test_info`. So the join function assumes `by = "name"`:

```
# these are equivalent
left_join(x = demographic, y = test_info)
left_join(x = demographic, y = test_info, by = "name")
```

VOCAB

The column used to connect rows across the tables is known as a "key". In the dplyr join functions, the key is specified in the `by` argument, as seen in `left_join(x = demographic, y = test_info, by = "name")`

What happens if the keys are named differently in the two datasets? Consider the `test_info_different_name` dataset below, where the "name" column has been changed to "test_recipient":

```
test_info_different_name <-
  tribble(~test_recipient,  ~test_date,     ~result,
          "Alice",          "2023-06-05",  "Negative",
          "Bob",            "2023-08-10",  "Positive",
          "Charlie",        "2023-07-15",  "Negative")
test_info_different_name
```

```
## # A tibble: 3 × 3
##   test_recipient test_date  result
##   <chr>          <chr>      <chr>
## 1 Alice          2023-06-05 Negative
## 2 Bob            2023-08-10 Positive
## 3 Charlie        2023-07-15 Negative
```

If we try to join `test_info_different_name` with our original `demographic` dataset, we will encounter an error:

```
left_join(x = demographic, y = test_info_different_name)
```

```
Error in `left_join()`:
! `by` must be supplied when `x` and `y` have no common
  variables.
ℹ Use `cross_join()` to perform a cross-join.
```

The error indicates that there are no common variables, so the join is not possible.

In situations like this, you have two choices: you can rename the column in the second dataframe to match the first, or more simply, specify which columns to join on using `by = c()`.

```r
left_join(x = demographic, y = test_info_different_name,
          by = c("name" = "test_recipient"))
```

```
## # A tibble: 3 × 4
##   name      age test_date  result
##   <chr>   <dbl> <chr>      <chr>
## 1 Alice      25 2023-06-05 Negative
## 2 Bob        32 2023-08-10 Positive
## 3 Charlie    45 2023-07-15 Negative
```

The syntax `c("name" = "test_recipient")` is a bit unusual. It essentially says, "Connect `name` from data frame x with `test_recipient` from data frame y because they represent the same data."

Consider the two datasets below, one with patient details and the other with medical check-up dates for these patients.

```r
patients <- tribble(
  ~patient_id, ~name,     ~age,
  1,           "John",     32,
  2,           "Joy",     28,
  3,           "Khan",      40
)

checkups <- tribble(
  ~patient_id, ~checkup_date,
  1,           "2023-01-20",
  2,           "2023-02-20",
  3,           "2023-05-15"
)
```

Join the `patients` dataset with the `checkups` dataset using `left_join()`

Two datasets are defined below, one with patient details and the other with vaccination records for those patients.

```
# Patient Details
patient_details <- tribble(
  ~id_number,  ~full_name,    ~address,
  "A001",      "Alice",       "123 Elm St",
  "B002",      "Bob",         "456 Maple Dr",
  "C003",      "Charlie",     "789 Oak Blvd"
)

# Vaccination Records
vaccination_records <- tribble(
  ~patient_code, ~vaccine_type,  ~vaccination_date,
  "A001",        "COVID-19",     "2022-05-10",
  "B002",        "Flu",          "2023-09-01",
  "C003",        "Hepatitis B",  "2021-12-15"
)
```

Join the `patient_details` and `vaccination_records` datasets. You will need to use the `by` argument because the patient identifier columns have different names.

Types of joins

The toy examples so far have involved datasets that could be matched perfectly - every row in one dataset had a corresponding row in the other dataset.

Real-world data is usually messier. Often, there will be entries in the first table that do not have corresponding entries in the second table, and vice versa.

To handle these cases of imperfect matching, there are different join types with specific behaviors: `left_join()`, `right_join()`, `inner_join()`, and `full_join()`. In the upcoming sections, we'll look at examples of how each join type operates on datasets with imperfect matches.

`left_join()`

Let's start with `left_join()`, which you've already been introduced to. To see how it handles unmatched rows, we will try to join our original `demographic` dataset with a modified version of the `test_info` dataset.

As a reminder, here is the `demographic` dataset, with Alice, Bob and Charlie:

```
demographic
```

```
## # A tibble: 3 × 2
##   name      age
##   <chr>   <dbl>
## 1 Alice      25
## 2 Bob         32
## 3 Charlie     45
```

For test information, we'll remove `Charlie` and we'll add a new patient, `Xavier`, and his test data:

```
test_info_xavier <- tribble(
  ~name,     ~test_date, ~result,
  "Alice",   "2023-06-05", "Negative",
  "Bob",     "2023-08-10", "Positive",
  "Xavier",  "2023-05-02", "Negative")
test_info_xavier
```

```
## # A tibble: 3 × 3
##   name   test_date  result
##   <chr>  <chr>      <chr>
## 1 Alice  2023-06-05 Negative
## 2 Bob    2023-08-10 Positive
## 3 Xavier 2023-05-02 Negative
```

If we perform a `left_join()` using `demographic` as the left dataset (`x = demographic`) and `test_info_xavier` as the right dataset (`y = test_info_xavier`), what should we expect? Recall that Charlie is only present in the left dataset, and Xavier is only present in the right. Well, here's what happens:

```
left_join(x = demographic, y = test_info_xavier, by = "name")
```

As you can see, with the *LEFT* join, all records from the *LEFT* dataframe (`demographic`) are retained. So, even though `Charlie` doesn't have a match in the `test_info_xavier` dataset, he's still included in the output. (But of course, since his test information is not available in `test_info_xavier` those values were left as `NA`.)

Xavier, on the other hand, who was only present in the right dataset, gets dropped.

The graphic below shows how this join worked:

KEY POINT

In a join function like `left_join(x, y)`, the dataset provided to the `x` argument can be termed the "left" dataset, while the dataset assigned to the `y` argument can be called the "right" dataset.

Now what if we flip the datasets? Let's see the outcome when `test_info_xavier` is the left dataset and `demographic` is the right one:

```
left_join(x = test_info_xavier, y = demographic, by = "name")
```

Once again, the `left_join()` retains all rows from the *left* dataset (now `test_info_xavier`). This means Xavier's data is included this time. Charlie, on the other hand, is excluded.

The graphic below illustrates how this works:

VOCAB

Primary Dataset: In the context of joins, the primary dataset refers to the main or prioritized dataset in an operation. In a left join, the left dataset is considered the primary dataset because all of its rows are retained in the output, regardless of whether they have a matching row in the other dataset.

Try out the following. Below are two datasets - one with disease diagnoses (`disease_dx`) and another with patient demographics (`patient_demographics`).

```
disease_dx <- tribble(
  ~patient_id, ~disease,       ~date_of_diagnosis,
  1,            "Influenza",    "2023-01-15",
  3,            "COVID-19",     "2023-03-05",
  8,            "Influenza",    "2023-02-20",
)

patient_demographics <- tribble(
  ~patient_id, ~name,       ~age,  ~gender,
  1,            "Fred",       28,   "Female",
  2,            "Genevieve", 45,   "Female",
  3,            "Henry",      32,   "Male",
  5,            "Irene",      55,   "Female",
  8,            "Jules",      40,   "Male"
)
```

Use `left_join()` to merge these datasets, keeping only patients for whom we have demographic information. Think carefully about which dataset to put on the left.

Let's try another example, this time with a more realistic set of data.

First, we have data on the TB incidence rate per 100,000 people for 47 African countries, from the WHO:

```
tb_2019_africa <- read_csv(here("data/tb_incidence_2019.csv"))
```

```
## Rows: 47 Columns: 3
## ── Column specification ────────────────────────────────
## Delimiter: ","
## chr (2): country, conf_int_95
## dbl (1): cases
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```
tb_2019_africa
```

We want to analyze how TB incidence in African countries varies with government health expenditure per capita. For this, we have data on health expenditure per capita in USD, also from the WHO:

```
health_exp_2019 <- read_csv(here("data/health_expend_per_cap_2019.csv"))
```

```
## Rows: 185 Columns: 2
## ── Column specification ────────────────────────────────
## Delimiter: ","
## chr (1): country
## dbl (1): expend_usd
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```
health_exp_2019
```

Which dataset should we use as the left dataframe for the join?

Since our goal is to analyze African countries, we should use `tb_2019_africa` as the left dataframe. This will ensure we keep all the African countries in the final joined dataset.

Let's join them:

```
tb_health_exp_joined <-
  tb_2019_africa %>%
  left_join(health_exp_2019, by = "country")
tb_health_exp_joined
```

Now in the joined dataset, we have just the 47 rows for African countries, which is exactly what we wanted!

All rows from the left dataframe `tb_2019_africa` were kept, while non-African countries from `health_exp_2019` were discarded.

We can check if any rows in `tb_2019_africa` did not have a match in `health_exp_2019` by filtering for `NA` values:

```
tb_health_exp_joined %>%
  filter(is.na(!expend_usd))
```

```
## # A tibble: 3 × 4
##   country      cases conf_int_95 expend_usd
##   <chr>        <dbl> <chr>            <dbl>
## 1 Mauritius       12 [9 - 15]            NA
## 2 South Sudan    227 [147 - 324]         NA
## 3 Comoros         35 [23 - 50]           NA
```

This shows that 3 countries - Mauritius, South Sudan, and Comoros - did not have expenditure data in `health_exp_2019`. But because they were present in `tb_2019_africa`, and that was the left dataframe, they were still included in the joined data.

To be sure, we can quickly confirm that those countries are absent from the expenditure dataset with a filter statement:

```
health_exp_2019 %>%
  filter(country %in% c("Mauritius", "South Sudan", "Comoros"))
```

```
## # A tibble: 0 × 2
## # ℹ 2 variables: country <chr>, expend_usd <dbl>
```

Indeed, these countries aren't present in `health_exp_2019`.

Copy the code below to define two datasets.

The first, `tb_cases_children` contains the number of TB cases in under 15s in 2012, by country:

```
tb_cases_children <- tidyr::who %>%
  filter(year == 2012) %>%
  transmute(country, tb_cases_smear_0_14 = new_sp_m014 + new_sp_f014)

tb_cases_children
```

```
## # A tibble: 5 × 2
##   country        tb_cases_smear_0_14
##   <chr>                        <dbl>
## 1 Afghanistan                    588
## 2 Albania                          0
## 3 Algeria                         89
## 4 American Samoa                  NA
## 5 Andorra                          0
```

And `country_continents`, from the {countrycode} package, lists all countries and their corresponding region and continent:

```
country_continents <-
  countrycode::codelist %>%
  select(country.name.en, continent, region)

country_continents
```

```
## # A tibble: 5 × 3
##   country.name.en continent region
##   <chr>           <chr>     <chr>
## 1 Afghanistan     Asia      South Asia
## 2 Albania         Europe    Europe & Central Asia
## 3 Algeria         Africa    Middle East & North Africa
## 4 American Samoa  Oceania   East Asia & Pacific
## 5 Andorra         Europe    Europe & Central Asia
```

Your goal is to add the continent and region data to the TB cases dataset.

Which dataset should be the left dataframe, `x`? And which should be the right, `y`? Once you've decided, join the datasets appropriately using `left_join()`.

`right_join()`

A `right_join()` can be thought of as a mirror image of a `left_join()`. The mechanics are the same, but now all rows from the *RIGHT* dataset are retained, while only those rows from the left dataset that find a match in the right are kept.

Let's look at an example to understand this. We'll use our original `demographic` and modified `test_info_xavier` datasets:

```
demographic
```

```
test_info_xavier
```

Now let's try `right_join()`, with `demographic` as the right dataframe:

```
right_join(x = test_info_xavier, y = demographic)
```

```
## Joining with `by = join_by(name)`
```

Hopefully you're getting the hang of this, and could predict that output! Since `demographic` was the *right* dataframe, and we are using *right*-join, all the rows from `demographic` are kept—Alice, Bob and Charlie. But only matching records in the left data frame `test_info_xavier`!

The graphic below illustrates this process:

An important point—the same final dataframe can be created with either `left_join()` or `right_join()`, it just depends on what order you provide the data frames to these functions:

```
# here, RIGHT_join prioritizes the RIGHT df, demographic
right_join(x = test_info_xavier, y = demographic)
```

```
## Joining with `by = join_by(name)`
```

```
# here, LEFT_join prioritizes the LEFT df, again demographic
left_join(x = demographic, y = test_info_xavier)
```

```
## Joining with `by = join_by(name)`
```

SIDE NOTE The one difference you might notice between left and right-join is that the final column orders are different. But columns can easily be rearranged, so worrying about column order is not really worth your time.

As we previously mentioned, data scientists typically favor `left_join()` over `right_join()`. It makes more sense to specify your primary dataset first, in the left position. Opting for a `left_join()` is a common best practice due to its clearer logic, making it less error-prone.

Great, now we understand how `left_join()` and `right_join()` work, let's move on to `inner_join()` and `full_join()`!

`inner_join()`

What makes an `inner_join` distinct is that rows are only kept if the joining values are present in *both* dataframes. Let's return to our example of patients and their COVID test results. As a reminder, here are our datasets:

```
demographic
```

```
test_info_xavier
```

Now that we have a better understanding of how joins work, we can already picture what the final dataframe would look like if we used an `inner_join()` on our two dataframes above. If only rows with joining values that are in *both* dataframes are kept, and the only patients that are in both `Demographic` and `test_info` are `Alice` and `Bob`, then they should be the only patients in our final dataset! Let's try it out.

```
inner_join(demographic, test_info_xavier, by="name")
```

Perfect, that's exactly what we expected! Here, `Charlie` was only in the `Demographic` dataset, and `Xavier` was only in the `test_info` dataset, so both of them were removed. The graphic below shows how this join works:

It makes sense that the order in which you specify your datasets doesn't change the information that's retained, given that you need joining values in both datasets for a row to be kept. To illustrate this, let's try changing the order of our datasets.

```
inner_join(test_info_xavier, demographic, by="name")
```

As expected, the only difference here is the order of our columns, otherwise the information retained is the same.

The following data is on foodborne-outbreaks in the US in 2019, from the CDC. Copy the code below to create two new dataframes:

```
total_inf <- tribble(
  ~pathogen,          ~total_infections,
  "Campylobacter",     9751,
  "Listeria",          136,
  "Salmonella",        8285,
  "Shigella",          2478,
)

outcomes <- tribble(
  ~pathogen,         ~n_hosp,     ~n_deaths,
  "Listeria",          128,         30,
  "STEC",              582,         11,
  "Campylobacter",     1938,        42,
  "Yersinia",          200,         5,
)
```

Which pathogens are common between both datasets? Use an `inner_join()` to join the dataframes, in order to keep only the pathogens that feature in both datasets.

Let's return to our health expenditure and income level data and apply what we've learnt to these datasets.

```
tb_2019_africa
```

```
health_exp_2019
```

Here, we can create a new dataframe called `inner_exp_tb` using an `inner_join()` to retain only the countries that we have both health expenditure and TB incidence rates data on. Let's try it out now:

```
inner_exp_tb <- tb_2019_africa %>%
  inner_join(health_exp_2019)
```

```
## Joining with `by = join_by(country)`
```

```
inner_exp_tb
```

That looks great! Along with `left_join()`, the `inner_join()` is one of the most common joins when working with data, so it's likely you will come across it a lot. It's a powerful and often-used tool, but it's also the join that excludes the most information, so be sure that you only want matching records in your final dataset or you may end up accidentally losing a lot of data! In contrast, `full_join()` is the most inclusive join, let's take a look at it in the next section.

Use an `inner_join()` to join the 2019 health expenditure and . Which countries are left in your final dataset?

`full_join()`

The peculiarity of `full_join()` is that it retains *all* records, regardless of whether or not there is a match between the two datasets. Where there is missing information in our final dataset, cells are set to `NA` just as we have seen in the `left_join()` and `right_join()`. Let's take a look at our `Demographic` and `test_info` datasets to illustrate this.

Here is a reminder of our datasets:

```
demographic
```

```
test_info_xavier
```

Now let's perform a `full_join`, with `Demographic` as our primary dataset.

```
full_join(demographic, test_info_xavier, by="name")
```

As we can see, all rows were kept so there was no loss in information! The graphic below illustrates this process:

Because this join isn't selective, everything ends up in the final dataset, so changing the order of our datasets won't change the information that's retained. It will only change the order of the columns in our final dataset. We can see this below when we specify `test_info` as our primary dataset and `Demographic` as our secondary dataset.

```
full_join(test_info_xavier, demographic, by="name")
```

Just as we saw above, all of the data from both of the original datasets are still there, with any missing information set to `NA`.

The following dataframes contain global malaria incidence rates per 100'000 people and global death rates per 100'000 people from malaria, from Our World in Data. Copy the code to create two small dataframes:

```
malaria_inc <- tribble(
  ~year, ~inc_100k,
  2010, 69.485344,
  2011, 66.507935,
  2014, 59.831020,
  2016, 58.704540,
  2017, 59.151703,
)

malaria_deaths <- tribble(
  ~year, ~deaths_100k,
  2011, 12.92,
  2013, 11.00,
  2015, 10.11,
  2016, 9.40,
  2019, 8.95
)
```

Then, join the above tables using a `full_join()` in order to retain all information from the two datasets.

Let's turn back to our TB dataset and our health expenditure dataset.

```
tb_2019_africa
```

```
## # A tibble: 5 × 3
##   country                cases conf_int_95
##   <chr>                  <dbl> <chr>
## 1 Burundi                  107 [69 - 153]
## 2 Sao Tome and Principe    114 [45 - 214]
## 3 Senegal                  117 [83 - 156]
## 4 Mauritius                 12 [9 - 15]
## 5 Côte d'Ivoire            137 [88 - 197]
```

```
health_exp_2019
```

```
## # A tibble: 5 × 2
##   country              expend_usd
##   <chr>                     <dbl>
## 1 Nigeria                    11.0
## 2 Bahamas                  1002
## 3 United Arab Emirates     1015
## 4 Nauru                    1038
## 5 Slovakia                 1058
```

Now let's create a new dataframe called `full_tb_health` using a full_join!

```
full_tb_health <- tb_2019_africa %>%
 full_join(health_exp_2019)
```

```
## Joining with `by = join_by(country)`
```

```
full_tb_health
```

Just as we saw earlier, all rows were kept between both datasets with missing values set to `NA`.

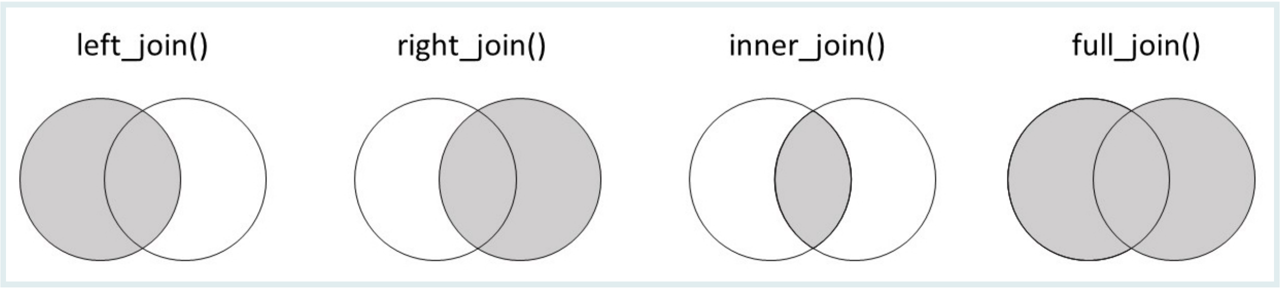
Just as for the previous exercise, obtain all countries and their corresponding region and continent from the {countrycode} package:

```
country_continents <-
  countrycode::codelist %>%
  select(country.name.en, continent, region)
```

Then, use a `full_join()` to join with the `tb_2019_africa` dataset.

Summary

Way to go, you now understand the basics of joining! The Venn diagram below gives a helpful summary of the different joins and the information that each one retains. It may be helpful to save this image for future reference!


left_join()
right_join()
inner_join()
full_join()