Dates 1: Recognizing and Formatting Dates

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

Intro

Understanding how to manipulate dates is a crucial skill when working with health data. From patient admission dates to vaccination schedules, date-related data plays a vital role in epidemiological analyses. In this lesson, we will learn how R stores and displays dates, as well as how to effectively manipulate, parse, and format them. Let's get started!

Learning Objectives

- You understand how dates are stored and manipulated in R
- You understand how to coerce strings to dates
- You know how to handle messy dates
- You are able to change how dates are displayed

Packages

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

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

Datasets

IRS Malawi

The first dataset we will be using contains data related to indoor residual spraying (IRS) for malaria control efforts between 2014–2019 in Illovo, Malawi. Notice the dataset is in long format with each row representing a period of time in which IRS occurred within a village. Given the same village is sprayed multiple times at different time points, the village names repeat. Long format datasets are often used when dealing with time series data with repeated measurements, as they are easier to manipulate for analyses and visualization.

```
irs <- read_csv(here("data/Illovo_data.csv"))
```

```
irs
```

```
## # A tibble: 5 × 9
##   village     target_spray sprayed coverage_p
##   <chr>              <dbl>   <dbl>      <dbl>
## 1 Mess                  87      64       73.6
## 2 Nkombedzi            183     169       92.4
## 3 B Compound            16      16      100
## 4 D Compound             3       2       66.7
## 5 Post Office            6       3       50
##   start_date_default end_date_default start_date_typical
##   <date>             <date>           <chr>
## 1 2014-04-07         2014-04-17       07/04/2014
## 2 2014-04-22         2014-04-27       22/04/2014
## 3 2014-05-13         2014-05-13       13/05/2014
## 4 2014-05-13         2014-05-13       13/05/2014
## 5 2014-05-13         2014-05-13       13/05/2014
## # ℹ 2 more variables: start_date_long <chr>,
## #   start_date_messy <chr>
```

The variables included in the dataset are the following:

- `village`: name of the village where the spraying occurred
- `target_spray`: number of structures targeted for spraying
- `sprayed`: number of structures actually sprayed
- `start_date_default`: day the spraying started, formatted as the default "YYYY-MM-DD"
- `end_date_default`: day the spraying ended, formatted as the default "YYYY-MM-DD"
- `start_date_typical`: day the spraying started, formatted "DD/MM/YYYY"
- `start_date_long`: day the spraying started, with the month written out fully, two-digit day, then four-digit year
- `start_date_messy`: day the spraying started with a mix of different formats.

Inpatient stays

The second dataset is mock data of simulated hospital inpatient stays. It contains admission and discharge dates for 150 patients. Similar to the IRS dataset, the admission dates are formatted in multiple different ways so that you can practice your formatting skills.

```
ip <- read_csv(here("data/inpatient_data.csv"))
```

```
ip
```

```
## # A tibble: 5 × 6
##   patient_id adm_date_default adm_date_common
##        <dbl> <date>           <chr>
## 1          1 2021-05-23       05/23/2021
## 2          2 2022-12-07       12/07/2022
## 3          3 2022-03-27       03/27/2022
## 4          4 2022-04-28       04/28/2022
## 5          5 2023-06-28       06/28/2023
##   adm_date_abbrev adm_date_messy disch_date_default
##   <chr>           <chr>          <date>
## 1 May 23, 2021    2021/05/23     2021-06-27
## 2 Dec 07, 2022    12-07-2022     2023-02-19
## 3 Mar 27, 2022    2022/03/27     2022-05-31
## 4 Apr 28, 2022    28.04.2022     2022-07-04
## 5 Jun 28, 2023    06-28-2023     2023-07-31
```

Introduction to dates in R

In R, there is a specific class designed to handle dates, known as `Date`. The default format for this class is "YYYY-MM-DD". For example, December 31st 2000 would be represented as `2000-12-31`.

However, if you just enter in such a date string, R will initially consider this to be a character:

```
class("2000-12-31")
```

```
## [1] "character"
```

If we want to create a `Date`, we can use the `as.Date()` function and write in the date following the default format:

```
my_date <- as.Date("2000-12-31")
class(my_date)
```

```
## [1] "Date"
```

WATCH OUT

Note the capital "D" in the `as.Date()` function!

With the date format, you can now do things like find the difference between two dates:

```
as.Date("2000-12-31") - as.Date("2000-12-20")
```

```
## Time difference of 11 days
```

This would of course not be possible if you had bare characters:

```
"2000-12-31" - "2000-12-20"
```

```
Error in "2000-12-31" - "2000-12-20" :
  non-numeric argument to binary operator
```

Many other operations apply only to the `Date` class. We'll explore these later.

The default format for `as.Date()` is "YYYY-MM-DD". Other common formats like "MM/DD/YYYY" or "Month DD, YYYY" won't work by default:

```
as.Date("12/31/2000")
as.Date("Dec 31, 2000")
```

However, R will also accept "/" instead of "-" as long as the order is still "YYYY/MM/DD". The dates will be displayed in the default "YYYY-MM-DD" format though:

```
as.Date("2000/12/31")
```

```
## [1] "2000-12-31"
```

So in summary, the only formats that work by default are "YYYY-MM-DD" and "YYYY/MM/DD". Later on in this lesson we will learn how to handle different date formats, and give tips on how to coerce dates that are imported as strings to the `Date` class. For

now, the important thing is to understand that dates have their own class which has its own formatting properties.

There is one other data class used for dates, called `POSIXct`. This class specifically handles dates and times together, and the default format is "YYYY-MM-DD HH:MM:SS". However for the purpose of this course, we won't be working with date-times as that level of analysis is much less common in the world of public health.

Coercing strings to dates

Let's return to our IRS dataset and take a look at how R has classified our date variables!

```
irs %>%
  select(contains("date"))
```

```
## # A tibble: 112 × 5
##    start_date_default end_date_default start_date_typical
##    <date>             <date>           <chr>
##  1 2014-04-07         2014-04-17       07/04/2014
##  2 2014-04-22         2014-04-27       22/04/2014
##  3 2014-05-13         2014-05-13       13/05/2014
##  4 2014-05-13         2014-05-13       13/05/2014
##  5 2014-05-13         2014-05-13       13/05/2014
##  6 2014-05-15         2014-05-26       15/05/2014
##  7 2014-05-27         2014-05-27       27/05/2014
##  8 2014-05-27         2014-05-27       27/05/2014
##  9 2014-05-28         2014-06-16       28/05/2014
## 10 2014-06-18         2014-06-27       18/06/2014
##    start_date_long start_date_messy
##    <chr>           <chr>
##  1 April 07 2014   04/07/14
##  2 April 22 2014   April 22 2014
##  3 May 13 2014     May 13 2014
##  4 May 13 2014     13-05-2014
##  5 May 13 2014     May 13 2014
##  6 May 15 2014     May 15 2014
##  7 May 27 2014     May 27 2014
##  8 May 27 2014     2014/05/27
##  9 May 28 2014     May 28 2014
## 10 June 18 2014    18-06-2014
## # ℹ 102 more rows
```

As we can see, the two columns that were recognized as dates are `start_date_default` and `end_date_default`, which follow R's "YYYY-MM-DD" format:

```
  start_date_default  end_date_default  start_date_typical
  👉<date>👈          👉<date>👈         <chr>
1 2014-04-07          2014-04-17        07/04/2014
2 2014-04-22          2014-04-27        22/04/2014
```

All other date columns in our dataset were imported as character strings ("chr"), and if we want to coerce them into dates we will need to tell R that they are dates, as well as specifying the order of the date components.

You may be wondering why it's necessary to specify the order. Well imagine that we have a date written `01-02-03`. Is that January 2nd 2003? February 1st 2003? Or maybe March 2nd 2001? There are so many different conventions for writing dates that if R were to guess the format, there would inevitably be instances where it guessed wrong.

To tackle this, there are two main ways to coerce strings to dates that involve specifying the component order. The first approach relies on base R, and the second uses a package called `lubridate` from the `tidyverse` library. Let's take a look at the base R function first!

Base R

We saw the function used to convert strings to dates using base R in the introduction, the `as.Date()` function. Let's try to use this on our `start_date_typical`column without specifying the order of components to see what happens.

```
irs %>%
  mutate(start_date_typical = as.Date(start_date_typical)) %>%
  select(start_date_typical)
```

```
## # A tibble: 5 × 1
##   start_date_typical
##   <date>
## 1 0007-04-20
## 2 0022-04-20
## 3 0013-05-20
## 4 0013-05-20
## 5 0013-05-20
```

Obviously, this is not at all what we wanted! If we take a look at the original variable, we can see that it's formatted "DD/MM/YYYY". R tried to apply it's default format to these dates, giving us these odd results.

WATCH OUT

Often R will throw an error if you try to coerce ambiguous strings to dates without specifying the order of its components. But, as we have just seen, this isn't always the case! Always double-check that your code has run how you expected and never rely on error messages alone to ensure that your data transformations have worked correctly.

For R to correctly interpret our dates, we have to use the `format` option and specify the components of our date using a series of symbols. The table below shows the symbols for the most common format components:

Component	Symbol	Example
Year (numeric, with century)	%Y	2023
Year (numeric, without century)	%y	23
Month (numeric, 01-12)	%m	01
Month (written out fully)	%B	January
Month (abbreviated)	%b	Jan
Day of the month	%d	31
Day of the week (numeric, 1-7 with Sunday being 1)	%u	5
Day of the week (written out fully)	%A	Friday
Day of the week (abbreviated)	%a	Fri

If we come back to our original `start_date_typical` variable, we see it's formatted as "DD/MM/YYYY" which is the day of the month, followed by the month represented as a number (01-12), and then the four-digit year. If we use these symbols, we should get the results we're looking for.

```
irs %>%
  mutate(start_date_typical = as.Date(start_date_typical, format="%d%m%Y"))
```

```
## # A tibble: 5 × 9
##   village     target_spray sprayed coverage_p
##   <chr>              <dbl>   <dbl>      <dbl>
## 1 Mess                  87      64       73.6
## 2 Nkombedzi            183     169       92.4
## 3 B Compound            16      16      100
## 4 D Compound             3       2       66.7
## 5 Post Office            6       3       50
##   start_date_default end_date_default start_date_typical
##   <date>             <date>           <date>
## 1 2014-04-07         2014-04-17       NA
## 2 2014-04-22         2014-04-27       NA
## 3 2014-05-13         2014-05-13       NA
## 4 2014-05-13         2014-05-13       NA
## 5 2014-05-13         2014-05-13       NA
## # ℹ 2 more variables: start_date_long <chr>,
## #   start_date_messy <chr>
```

Ok so it's still not what we wanted. Do you have an idea why that may be? It's because the components of our dates are separated by a slash "/", which we have to include into our format option. Let's try it again!

```
irs %>%
  mutate(start_date_typical = as.Date(start_date_typical,
         format="%d/%m/%Y"))%>%
  select(start_date_typical)
```

```
## # A tibble: 5 × 1
##   start_date_typical
##   <date>
## 1 2014-04-07
## 2 2014-04-22
## 3 2014-05-13
## 4 2014-05-13
## 5 2014-05-13
```

This time it worked perfectly! Now we know how to coerce strings to dates in base R using the `as.Date()` function with the `format` option.

Coerce long date

Try to coerce the column `start_date_long` from the IRS dataset to the `Date` class. Don't forget to include all elements into the format option, including the symbols that separate the components of the date!

Find code errors

Can you find all the errors in the following code?

```
as.Date("June 26, 1987", format = "%b%d%y")
```

```
## [1] NA
```

Lubridate

The `lubridate` package provides a much more user-friendly way of coercing strings to dates than base R. With this package, all that's necessary is that you specify the order in which the year, month, and day appear using "y", "m", and "d", respectively. With these functions, specifying the characters that separate the different components of the date isn't necessary.

Let's take a look at a few examples:

```
dmy("30/04/2002")
```

```
## [1] "2002-04-30"
```

```
mdy("April 30th 2002")
```

```
## [1] "2002-04-30"
```

```
ymd("2002-04-03")
```

```
## [1] "2002-04-03"
```

That was easy enough! And as we can see, our dates are displayed using the default R format. Now that we understand how to use the functions in the lubridate package, let's try to apply them to the `start_day_long` variable from our dataset.

```
irs %>%
  mutate(start_date_long = mdy(start_date_long)) %>%
  select(start_date_long)
```

```
## # A tibble: 5 × 1
##   start_date_long
##   <date>
## 1 2014-04-07
## 2 2014-04-22
## 3 2014-05-13
## 4 2014-05-13
## 5 2014-05-13
```

Perfect, that's exactly what we wanted!

Coerce typical date

Try to coerce the column `start_date_typical` from the IRS dataset to the `Date` class using the functions in the lubridate package.

Base and lubridate formatting The following table contains the formats found in the `adm_date_abbr` and `adm_date_messy` formats from our inpatient dataset. See if you can fill in the blank cells:

Date example	Base R	Lubridate
Dec 07, 2022		
03-27-2022		mdy
28.04.2022		
	%Y/%m/%d	

Now we know two ways to coerce strings to the date class by specifying the order of the components! But what if we have multiple date formats within the same column? Let's move on to the next section to find out!

Handling messy dates with `lubridate::parse_date_time()`

When working with dates, sometimes you'll have various different formats within the same column. Luckily, lubridate has a useful function just for this purpose! The `parse_date_time()` function is similar to the previous functions we saw in the lubridate package, but with more flexibility and the possibility of including multiple date formats into the same call using the `orders` argument. Let's quickly take a look at how it works with a few simple examples.

To get an understanding of how to use `parse_date_time()`, let's apply it to a single string that we want to coerce to a date.

```
parse_date_time("30/07/2001", orders="dmy")
```

```
## [1] "2001-07-30 UTC"
```

That worked perfectly! Using the function like this is equivalent to using `dmy()`. However, the real power in the function is when we have multiple date strings that have different formats.

The "UTC" part is the default time zone used to parse the date. This can be changed in with the `tz=` argument, but changing the default time zone is rarely necessary when dealing with dates alone, as opposed to date-times.

Let's take a look at another example but with two different formats:

```
parse_date_time(c("1 Jan 2000", "July 30th 2001"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-01-01 UTC" "2001-07-30 UTC"
```

Note that this specific example will still work if you change the order in which you present the formats:

```
parse_date_time(c("1 Jan 2000", "July 30th 2001"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-01 UTC" "2001-07-30 UTC"
```

The last code chunk still works because `parse_date_time()` checks each format specified in the `orders` argument until it finds a match. This means whether you list "dmy" first or "mdy" first, it will try both formats on each date string to see which one fits. The order doesn't matter for distinct date strings that can only match one format.

However, when dealing with ambiguous dates like "01/02/2000" and "01/03/2000", which could be interpreted as either January 2nd and January 3rd or February 1st and March 1st respectively, the order in `orders` really does matter:

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-02 UTC" "2000-01-03 UTC"
```

In the example above, because "mdy" is listed first, the function interprets the dates as January 2nd and January 3rd. But, if you switched the order and listed "dmy" first, it would interpret the dates as February 1st and March 1st:

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-02-01 UTC" "2000-03-01 UTC"
```

Hence, when there's potential ambiguity in the date strings, the order in which you specify the formats becomes vital.

Using parse_date_time

The dates in the code below are November 9th 2002, December 4th 2001, and June 5th 2003. Complete the code to coerce them from strings to dates.

```
parse_date_time(c("11/09/2002", "12/04/2001", "2003-06-05"), orders=c(...))
```

Let's return to our dataset, this time to the `start_date_messy` column.

```
irs %>%
  select("start_date_messy")
```

```
## # A tibble: 5 × 1
##   start_date_messy
##   <chr>
## 1 04/07/14
## 2 April 22 2014
## 3 May 13 2014
```

```
## 4 13-05-2014
## 5 May 13 2014
```

Given that this column that was created specifically for this course, we know the different date formats in this column. In your own work, always be sure that you know what format your dates are in, as we know some can be ambiguous.

Here, we're working with four different formats, specifically:

- YYYY/MM/DD
- Month DD YYYY
- DD-MM-YYYY
- MM/DD/YYYY

Let's see how this looks in lubridate compared to base R:

Example date	Base R	Lubridate
2014/05/13	%Y/%m/%d	ymd
May 13 2014	%B %d %Y	mdy
27-05-2014	%d-%m-%Y	dmy
07/21/14	%m/%d/%y	mdy

Here, lubridate considers there to be only three different formats ("ymd", "mdy", and "dmy"). Now that we know how our data is formatted, we can use the `parse_date_time()` function to clean it up.

```
irs %>%
  select(start_date_messy) %>%
  mutate(start_date_messy = parse_date_time(start_date_messy, orders =
         c("mdy", "ymd", "dmy")))
```

start_date_messy
2014-04-07
2014-04-22
2014-05-13
2014-05-13
2014-05-13

That's much better! R has correctly formatted our column and it is now recognized as a date variable. You may be wondering if the ordering of the formats is necessary in this case. Let's try a different order and find out!

```
irs %>%
  select(start_date_messy) %>%
  mutate(start_date_messy = parse_date_time(start_date_messy, orders =
         c("dmy", "mdy", "ymd")))
```

start_date_messy
2014-04-07
2014-04-22
2014-05-13
2014-05-13
2014-05-13

That didn't seem to make a difference, the dates are still correctly formatted! If you're wondering why the order mattered in our previous example but not here, it's to do with how the parse_date_times() function works. When given multiple orders, the function tries to find the best fit for a subset of observations by considering the dates separators' and favoring the order in which the formats were supplied. In our last example, both dates were separated by a "/" and both supplied formats ("dmy" and "mdy") were possible formats, so the function favored the first one given.

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("mdy", "dmy"))
```

```
## [1] "2000-01-02 UTC" "2000-01-03 UTC"
```

```
parse_date_time(c("01/02/2000", "01/03/2000"), orders=c("dmy", "mdy"))
```

```
## [1] "2000-02-01 UTC" "2000-03-01 UTC"
```

In our IRS dataset, we also had formats that could be ambiguous such as DD-MM-YYYY and MM/DD/YYYY. But here the function can use the separators as a hint to find formatting rules and distinguish between different formats. For example, if we have an ambiguous date such as `01-02-2000`, but also a date with the same separator that is not ambiguous such as `30-05-2000`, the function will determine that the most likely answer is that all dates separated by a "-" are in DD-MM-YYYY format, and apply this rule recursively to the input data. If you want to learn more about the details of the `parse_date_time()` function, click here or run `?parse_date_time` in R!

Using parse_date_time with adm_date_messy With the help of the table you completed from the exercise in **Section 6.2 Lubridate**, use the `parse_date_time()` function to clean up the `adm_date_messy` column in the inpatient dataset, `ip`!

Changing how dates are displayed

Up until now we have been coercing strings of various formats to the `Date` class which follows a default "YYYY-MM-DD" format. But what if we want our dates to be displayed in a specific format that's different from this default, such as when we're creating reports or

graphs? This is made possible by converting the dates back into strings using using the `format()` function in base R!

The `format()` function gives you lots of freedom to customize the appearance of your dates. You can do this by using the same symbols we've encountered before with the `as.Date()` function, ordering them to match how you want your date to look. Let's go back to the table to refresh our memory on how different date parts are represented in base R.

Component	Symbol	Example
Year (numeric, with century)	%Y	2023
Year (numeric, without century)	%y	23
Month (numeric, 01-12)	%m	01
Month (written out fully)	%B	January
Month (abbreviated)	%b	Jan
Day of the month	%d	31
Day of the week (numeric, 1-7 with Monday being 1)	%u	5
Day of the week (written out fully)	%A	Friday
Day of the week (abbreviated)	%a	Fri

Great, now let's try to apply this function to a single date. Let's say we want the date `2000-01-31` to be displayed as "Jan 31, 2000".

```
my_date <- as.Date("2000-01-31")
format(my_date, "%b %d, %Y")
```

```
## [1] "Jan 31, 2000"
```

Create date vector Format the date below to MM/DD/YYYY using the format function:

```
my_date <- as.Date("2018-05-07")
```

Now, let's try using it on our dataset. Let's create a new variable called `start_date_char` from the `start_date_default` column in our dataset. We'll format it to be displayed as DD/MM/YYYY.

```
irs %>%
  mutate(start_date_char = format(start_date_default, "%d/%m/%Y")) %>%
  select(start_date_char)
```

```
## # A tibble: 5 × 1
##   start_date_char
##   <chr>
## 1 07/04/2014
## 2 22/04/2014
## 3 13/05/2014
```

```
## 4 13/05/2014
## 5 13/05/2014
```

Looking great! Let's do one last example using our `end_date_default` variable and formatting it as Month DD, YYYY.

```
irs %>%
  mutate(end_date_char = format(end_date_default, "%B %d, %Y")) %>%
  select(end_date_char)
```

```
## # A tibble: 5 × 1
##   end_date_char
##   <chr>
## 1 April 17, 2014
## 2 April 27, 2014
## 3 May 13, 2014
## 4 May 13, 2014
## 5 May 13, 2014
```

Looks great!

WRAP UP!

Congratulations on finishing the first Dates lesson! Now that you understand how Dates are stored, displayed, and formatted in R, you can move on to the next section where you'll learn how to perform manipulations with dates and how to create basic time series graphs.

Answer Key

Coerce long date

```
irs <- irs %>%
  mutate(start_date_long = as.Date(start_date_long, format="%B, %d %Y"))
```

Find code errors

```
as.Date("June 26, 1987", format = "%B %d, %Y")
```

Coerce typical date

```
irs %>%
  mutate(start_date_typical = dmy(start_date_typical))
```

Base and lubridate formatting

Date example	Base R	Lubridate
Dec 07, 2022	%b %d, %Y	mdy
03-27-2022	%m-%d-%Y	mdy
28.04.2022	%d.%m.%Y	dmy
2021/05/23	%Y/%m/%d	ymd

Using parse_date_time

```
parse_date_time(c("11/09/2002", "12/04/2001", "2003-06-05"), orders=c("mdy",
        "ymd"))
```

Using parse_date_time with adm_date_messy

```
ip %>%
  mutate(adm_date_messy = parse_date_time(adm_date_messy, orders = c("mdy",
        "dmy", "ymd")))
```

Create date vector

```
my_date <- as.Date("2018-05-07")
format(my_date, "%m/%d/%Y")
```

Contributors

The following team members contributed to this lesson:

(make sure to update the contributor list accordingly!)

Dates 2: Working with Dates

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

Intro

You now have a solid understanding of how dates are stored, displayed, and formatted in R. In this lesson, you will learn how to perform simple analyses with dates, such as calculating the time between date intervals and creating time series graphs! These skills are crucial for anyone working with health data, as they are the basis to understanding temporal patterns such as the spread of disease, changes in population-level health indicators, and the impact of preventive measures!

Learning Objectives

- You know how to calculate intervals between dates
- You know how to extract components from date columns
- You know how to round dates
- You are able to create simple time series graphs

Packages

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

```
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse,
               lubridate,
               ggplot2)
```

Dataset

The first dataset we will be working with is the IRS dataset from previous the previous lesson. Check out the first Dates lesson for more information about the contents of this dataset.

```
irs <- read_csv(here("data/Illovo_data.csv"))
```

```
irs
```

```
## # A tibble: 5 × 9
##   village     target_spray sprayed coverage_p
##   <chr>              <dbl>   <dbl>      <dbl>
## 1 Mess                  87      64       73.6
## 2 Nkombedzi            183     169       92.4
## 3 B Compound            16      16      100
## 4 D Compound             3       2       66.7
## 5 Post Office            6       3       50
##   start_date_default end_date_default start_date_typical
##   <date>             <date>           <chr>
## 1 2014-04-07         2014-04-17       07/04/2014
## 2 2014-04-22         2014-04-27       22/04/2014
## 3 2014-05-13         2014-05-13       13/05/2014
## 4 2014-05-13         2014-05-13       13/05/2014
## 5 2014-05-13         2014-05-13       13/05/2014
## # ℹ 2 more variables: start_date_long <chr>,
## #   start_date_messy <chr>
```

The second dataset that we will be using contains data from the same study that the IRS data was taken from. Here, we have the average monthly incidence rate of malaria per 1000 people from 2015-2019 for villages that received IRS (cases) and villages that didn't receive IRS (controls). The dataset also contains the average minimum temperature and average maximum temperature in Illovo for each month from 2015-2019.

```
inc_temp <- read_csv(here("data/Illovo_ir_weather.csv"))
```

```
inc_temp
```

```
## # A tibble: 5 × 5
##   date       ir_case ir_control avg_min avg_max
##   <date>       <dbl>      <dbl>   <dbl>   <dbl>
## 1 2015-01-10    42.9       19.6    21.2    31.6
## 2 2015-02-03    61.0       10.1    21.5    32.9
## 3 2015-03-11    74.1       56.8    20.6    33.4
## 4 2015-04-15    95.2       34.7    18.5    32.3
## 5 2015-05-05    89.8       31.9    15.9    31.4
```

The variables included in the dataset are the following:

- `date`: Monthly time points ranging from 2015-2019, with the day of the month randomly generated
- `ir_case`: Average incidence rate of malaria per 1000 people for villages that received IRS
- `ir_control`: Average incidence rate of malaria per 1000 people for villages that did not receive IRS
- `avg_min`: Average monthly minimum temperature in Celsius
- `avg_min`: Average monthly maximum temperature in Celsius

WEATHER

```
weather <- read_csv(here("data/Illovo_weather.csv"))
```

```
weather
```

```
## # A tibble: 5 × 4
##   date       min_temp max_temp  rain
##   <date>        <dbl>    <dbl> <dbl>
## 1 2015-01-01     21.5     29.9  21.7
## 2 2015-01-02     19.6     30.4   2.2
## 3 2015-01-03     21.6     29.9  25.8
## 4 2015-01-04     20       29.5   1
## 5 2015-01-05     20       32.2  53
```

Calculating Date Intervals

To start off, we're going to look at two ways to calculate intervals, the first using the "-" operator in base R, and the second using the interval operator from the `lubridate` package. Let's take a look at both of these and compare.

Using the "-" operator

The first way to calculate time differences is using the "-" operator to subtract one date from another. Let's create two date variables and try it out!

```
date_1 <- as.Date("2000-01-01") # January 1st, 2000
date_2 <- as.Date("2000-01-31") # January 31st, 2000
date_2 - date_1
```

```
## Time difference of 30 days
```

It's that simple! Here we can see that R outputs the time difference in days.

Using the interval operator from `lubridate`

The second way to calculate time intervals is by using the `%--%` operator from the `lubridate` package. We can see here that the output is slightly different to the base R output.

```
date_1 %--% date_2
```

```
## [1] 2000-01-01 UTC--2000-01-31 UTC
```

Our output is an interval between two dates. If we want to know how long has passed in days, we have to use the `days()` function. The `(1)` here tells lubridate to count in increments of one day at a time.

```
date_1 %--% date_2/days(1)
```

```
## [1] 30
```

Technically, specifying `days(1)` isn't actually necessary, we can also leave the parentheses empty (ie. `days()`)and get the same result because lubridate's default is to count in increments of 1. However, if we want to count in increments of 5 days for example, we can specify `days(5)` and the result returned to us will be 6, because 5*6=30.

```
date_1 %--% date_2/days(5)
```

```
## [1] 6
```

Use all the three different methods to calculate the time between the dates below. Use the `weeks()` function in place of `days()` in the lubridate method to return the answer in weeks.

```
oct_31 <- as.Date("2023-10-31")
jul_20 <- as.Date("2023-07-20")
```

Comparison

So which of the methods is the best to use? Well, you can probably tell that lubridate has more flexibility, and thanks to how it handles dates (the specifics of which are beyond the

Let's take a look at an example of this by calculating an interval of 6 years. With base R, the results are returned in days. If we want to get the result in years, we have to use the `as.numeric()` function to transform the results into a number without the days unit, and divide by 365.25. We use 365.25 because there is one extra day every 4 years (i.e. leap years), so on average there is 365.25 days per year. With lubridate, we can just specify `years()`. Let's see how each performs in the following example.

```
date_1 <- as.Date("2000-01-01") # January 1st, 2000
date_2 <- as.Date("2006-01-01") # January 1st, 2006
as.numeric(date_2-date_1)/365.25
```

```
## [1] 6.001369
```

```
date_1 %--% date_2/years()
```

```
## [1] 6
```

Ok, so they're very similar, but not exactly the same! Obviously the correct answer is that 6 years has passed between the two given dates. Yet, dates can get complicated since the number of days in a year, or even a month, isn't always the same. Trying to calculate date intervals with base R is an approximation rather than an exact answer. In the example above, we couldn't get exactly 6 years using the minus oprator in base R, even if we divide by 365, 365.25, or 366.

```
as.numeric(date_2-date_1)/365
```

```
## [1] 6.005479
```

```
as.numeric(date_2-date_1)/365.25
```

```
## [1] 6.001369
```

```
as.numeric(date_2-date_1)/366
```

```
## [1] 5.989071
```

Lubridate on the other hand can handle these complexities, so it will give an exact answer. The differences are slight, but in terms of flexibility and accuracy, lubridate is the clear winner!

SIDE NOTE

SIDE NOTE

Although we don't cover date-times in this course, it's good to know that `lubridate` is particularly handy for dealing with time zones and daylight savings shifts.

Can you apply this to our IRS dataset? Create a new column called `spraying_time` and using lubridates `%--%` operator, calculate the number of days between `start_date_default` and `end_date_default`.

Extracting Date Components

Sometimes during your data cleaning or analysis, you may need to extract a specific component of your date variable. A set of useful functions within the `lubridate` package allows you to do exactly this. For example, if we wanted to create a column with just the month that spraying started at each interval, we could use the `month()` function in the following way:

```
irs %>%
  mutate(month_start = month(start_date_default))%>%
  select(village, start_date_default, month_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default month_start
##   <chr>       <date>                   <dbl>
## 1 Mess        2014-04-07                   4
## 2 Nkombedzi   2014-04-22                   4
## 3 B Compound  2014-05-13                   5
## 4 D Compound  2014-05-13                   5
## 5 Post Office 2014-05-13                   5
```

As we can see here, this function returns the month as a number from 1-12. For our first observation, the spraying started during the fourth month, so in April. It's that simple! .If we want to have R display the month written out rather than the number underneath it, we can use the `label=TRUE` argument.

```
irs %>%
  mutate(month_start = month(start_date_default, label=TRUE))%>%
  select(village, start_date_default, month_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default month_start
##   <chr>       <date>             <ord>
## 1 Mess        2014-04-07         Apr
## 2 Nkombedzi   2014-04-22         Apr
```

```
## 3 B Compound  2014-05-13          May
## 4 D Compound  2014-05-13          May
## 5 Post Office 2014-05-13          May
```

Likewise, if we wanted to extract the year, we would use the `year()` function.

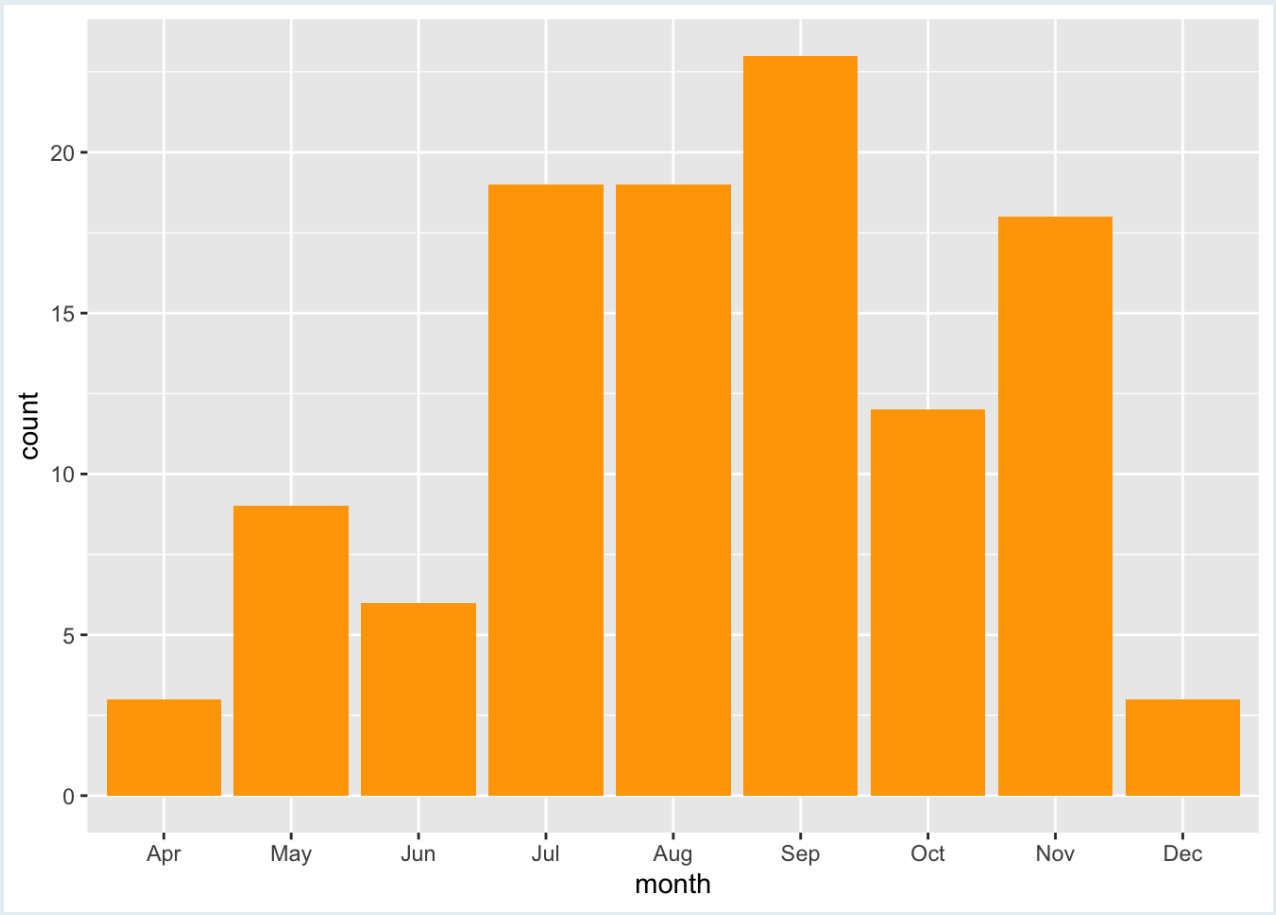
```
irs %>%
  mutate(year_start = year(start_date_default)) %>%
  select(village, start_date_default, year_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default year_start
##   <chr>       <date>                  <dbl>
## 1 Mess        2014-04-07               2014
## 2 Nkombedzi   2014-04-22               2014
## 3 B Compound  2014-05-13               2014
## 4 D Compound  2014-05-13               2014
## 5 Post Office 2014-05-13               2014
```

Create a new variable called `wday_start` and extract the day of the week that the spraying started in the same way as above but with the `wday()` function. Try to display the days of the week written out rather than numerically.

One reason why you may want to extract specific date components is when you want to visualize your data, which is very simple to do in R using the `ggplot2` package! For example, let's say we wanted to compare the months when spraying starts, we can do this by creating a new month variable with the `month()` function, and plotting a bar graph with `geom_bar`.

```
irs %>%
  mutate(month = month(start_date_default, label = TRUE)) %>%
  ggplot(aes(x = month)) +
  geom_bar(fill = "orange")
```



Here we can see that most spraying campaigns started between July and November, with none occurring in the first three months of the year. The authors of the paper that this data was drawn from have stated that spraying campaigns aimed to finish just as the rainy season (November-April) in Malawi started. This was both for practical reasons and in anticipation of higher malaria transmission. We can see this temporal spraying pattern reflected in our graph!

Create a new graph of months when the spraying campaign ended and compare it to the graph of when they started. Does this match your expectations considering the `spraying_time`variable that you created in the exercise of the previous section?

Rounding

Sometimes it's necessary to round our dates up or down if we want to analyze or vizualize our data in a meaningful way. First, let's see what we mean by rounding with a few simple examples.

Let's take the date March 17th 2012. If we wanted to round down to the nearest month, then we would use the `floor_date()` function from `lubridate` with the `unit="month"`

```
my_date_down <- as.Date("2012-03-17")
floor_date(my_date_down, unit="month")
```

```
## [1] "2012-03-01"
```

As we can see, our date is now March 1st, 2012.

If we wanted to round up, we can use the `ceiling_date()` function. Let's try this on out on the date January 3rd 2020.

```
my_date_up <- as.Date("2020-01-03")
ceiling_date(my_date_up, unit="month")
```

```
## [1] "2020-02-01"
```

With `ceiling_date()`, January 3rd had been rounded up to February 1st.

Finally, we can also simply round without specifying up or down and the dates are automatically round to the nearest specified unit.

```
my_dates <- as.Date(c("2000-11-03", "2000-11-27"))
round_date(my_dates, unit="month")
```

```
## [1] "2000-11-01" "2000-12-01"
```

Here we can see that by rounding to the nearest month, November 3rd is round down to November 1st, and November 27th is round up to December 1st.

We can also round up or down to the nearest year. What do you think the output would be if we round down the date November 29th 2001 to the nearest year:

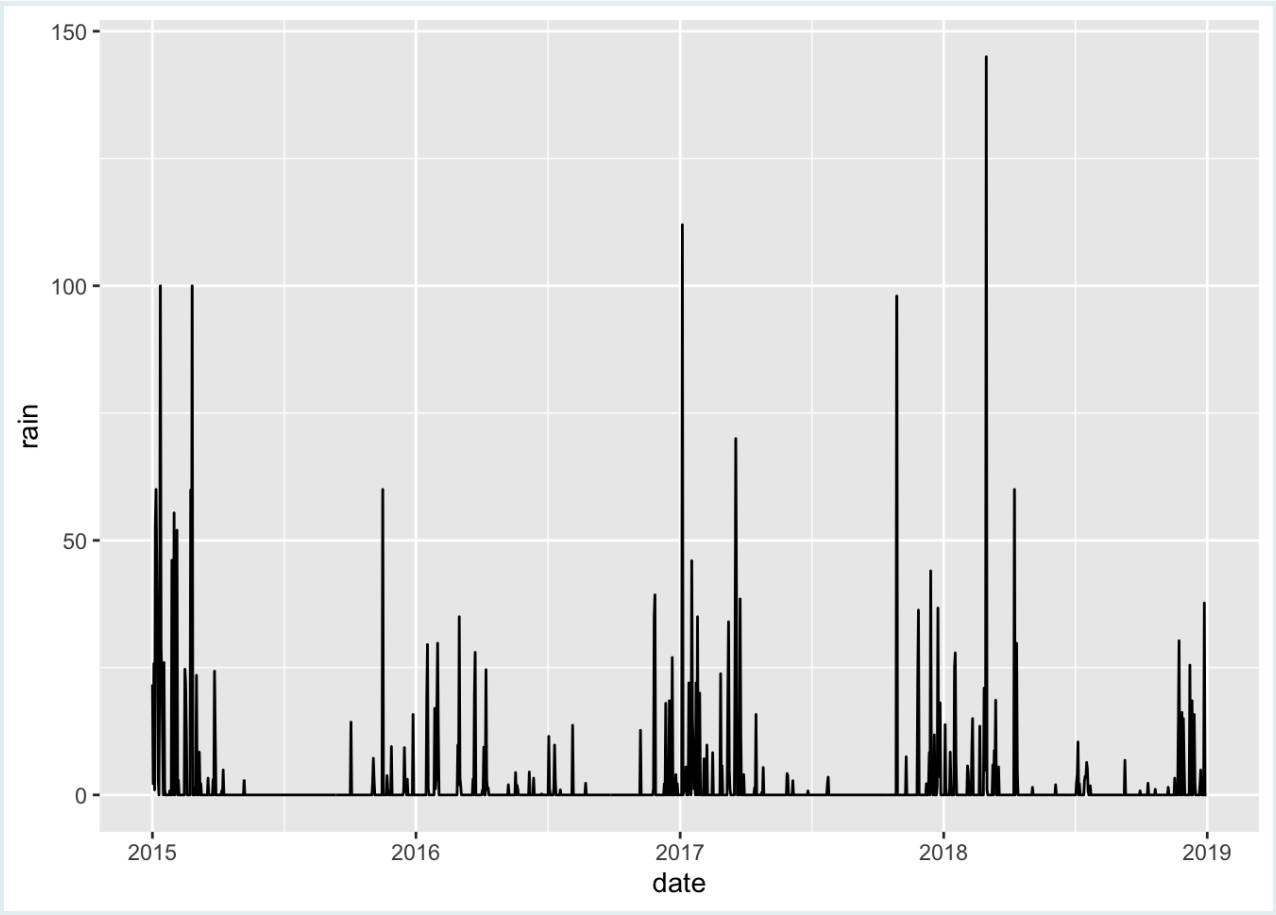
```
date_round <- as.Date("2001-11-29")
floor_date(date_round, unit="year")
```

Hopefully what we mean by rounding is a little bit more clear! So why could this be helpful with our data? Well, let's now turn to our weather data.

```
weather
```

As we can see, our weather data is recorded daily, but this level of detail isn't ideal for studying how weather patterns affect malaria transmission, which follows a seasonal pattern. Daily weather data can be quite noisy given the significant variation from one day to the next. For example, a graph of daily rainfall wouldn't be very informative. Let's try it out to see:

```
weather %>%
  ggplot()+
  geom_line(aes(date, rain))
```



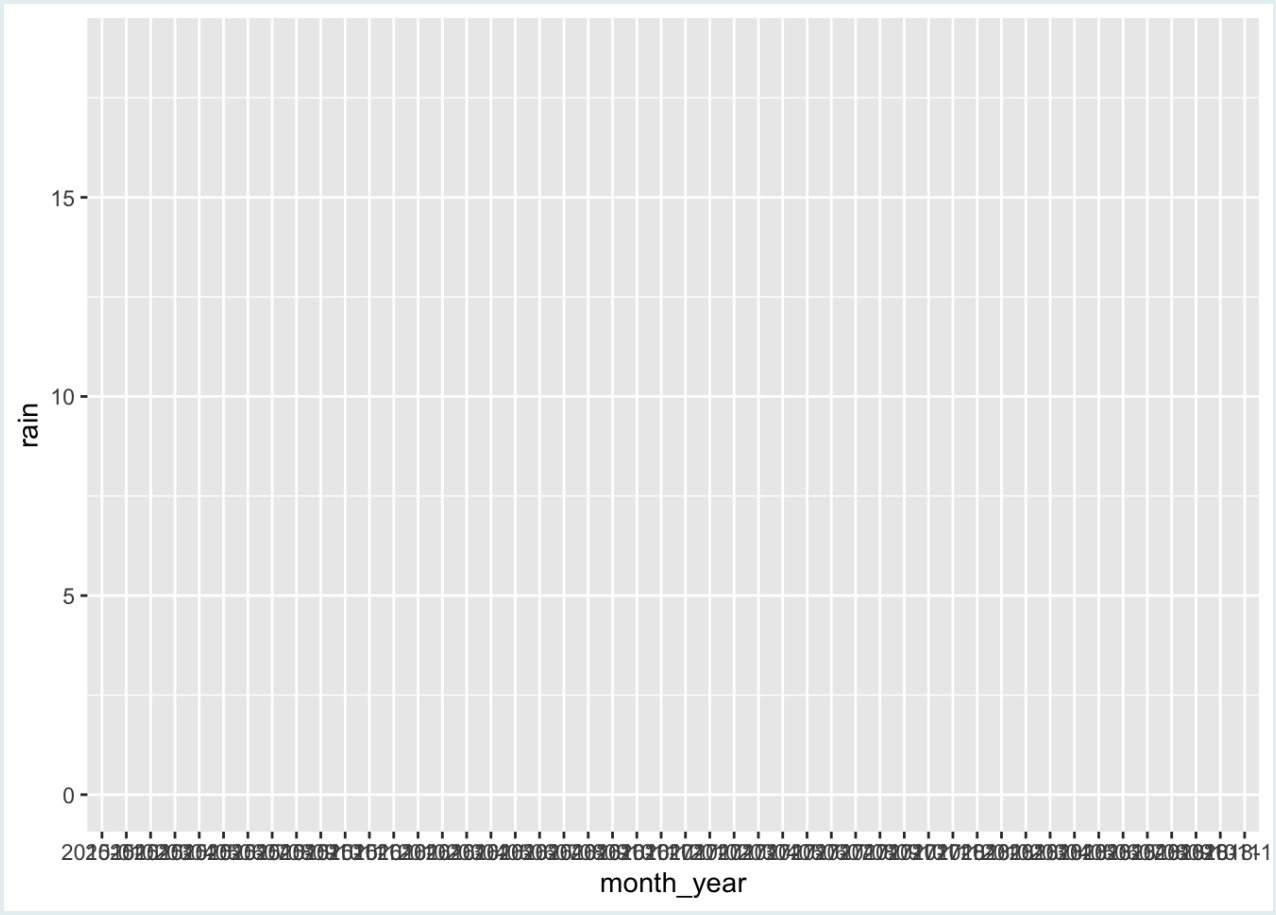
Aside from being visually messy, this graph fails to illustrate seasonal trends effectively. Monthly aggregation is a more effective approach for capturing seasonal variations and reducing noise. If we wanted to plot the monthly average rainfall, our first attempt may be to use the `str_sub()` function to extract the first seven characters of our date (the month and year component).

```
weather_bad <- weather %>%
  mutate(month_year=str_sub(date, 1, 7)) %>%
  group_by(month_year) %>%
  summarise(rain=mean(rain))
weather_bad
```

However, if we try to plot this, our new variable `month_year` is no longer a date variable, so we can't plot graphs over time as it's not continuous!

```
 weather_bad %>%
  ggplot() +
  geom_line(aes(month_year, rain))
```

```
## `geom_line()`: Each group consists of only one
## observation.
## ℹ Do you need to adjust the group aesthetic?
```

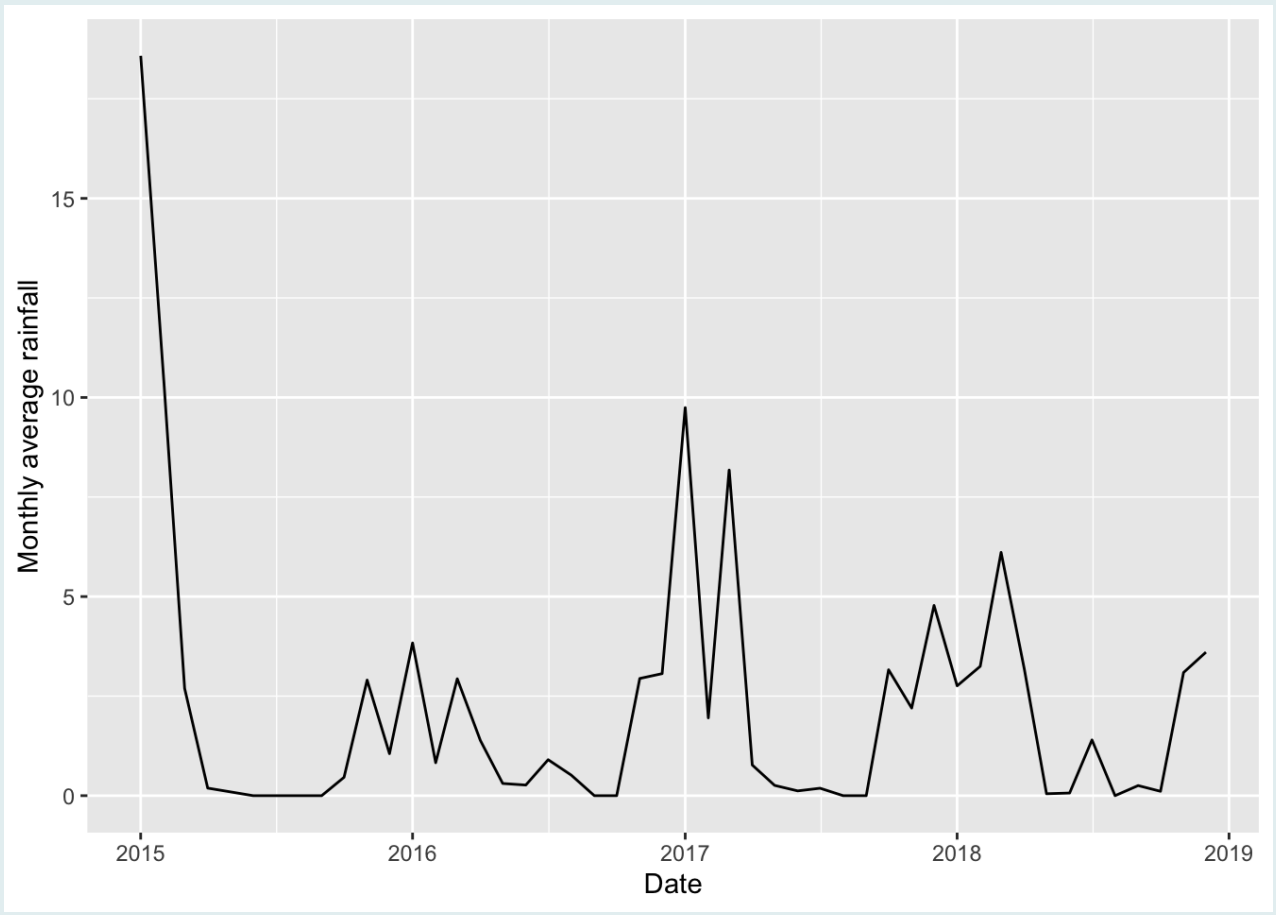


The best way to do this is to first round our dates down to the month using `floor_date()` function, then group our data by our new `month_year` variable, and then calculate the monthly average. Let's try it out now.

```
weather <- weather %>%
  mutate(month_year=floor_date(date, unit="month")) %>%
  group_by(month_year) %>%
  summarise(rain=mean(rain))
weather
```

Now we can plot our data and we'll have a graph of the average monthly rainfall over the 4 year spraying period.

```
 weather %>%
  ggplot() +
  geom_line(aes(month_year, rain)) +
  labs(x="Date", y="Monthly average rainfall")
```



That looks much better! Now we get a much clearer picture of seasonal trends and yearly variations.

Using the weather data, create a new graph plotting the average monthly minimum and maximum temperatures from 2015-2019.

WRAP UP!

[XXX NICE WRAP UP MESSAGE OR SUMMARY IF NEEDED HERE XXX]

Answer Key

```
irs %>%
  mutate(wday_start= wday(start_date_default, label=TRUE)) %>%
  select(village, start_date_default, wday_start)
```

```
## # A tibble: 5 × 3
##   village     start_date_default wday_start
##   <chr>       <date>             <ord>
## 1 Mess        2014-04-07         Mon
## 2 Nkombedzi   2014-04-22         Tue
## 3 B Compound  2014-05-13         Tue
## 4 D Compound  2014-05-13         Tue
## 5 Post Office 2014-05-13         Tue
```

Contributors

The following team members contributed to this lesson:

(make sure to update the contributor list accordingly!)

Factors in R

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

Introduction

Factors are an important data class for representing and working with categorical variables in R. In this lesson, we will learn how to create factors and how to manipulate them with functions from the `forcats` package, a part of the tidyverse. Let's dive in!

Learning Objectives

- You understand what factors are and how they differ from characters in R.
- You are able to modify the **order** of factor levels.
- You are able to modify the **value** of factor levels.

Packages

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

Dataset: HIV Mortality

We will use a dataset containing information about HIV mortality in Colombia from 2010 to 2016, which is hosted on the open data platform 'Datos Abiertos Colombia.' You can learn more and access the full dataset here.

Each row corresponds to an individual who passed away from AIDS or AIDS-related-complications.

```
hiv_mort <- read_csv(here("data/colombia_hiv_deaths_2010_to_2016"))
```

```
## # A tibble: 5 × 25
##   municipality_type death_location  birth_date birth_year
##   <chr>             <chr>           <date>          <dbl>
## 1 Municipal head    Hospital/clinic 1956-05-26       1956
## 2 Municipal head    Hospital/clinic 1983-10-10       1983
## 3 Municipal head    Hospital/clinic 1967-11-22       1967
## 4 Municipal head    Home/address    1964-03-14       1964
## 5 Municipal head    Hospital/clinic 1960-06-27       1960
##   birth_month birth_day death_year death_month death_day
##   <chr>           <dbl>      <dbl> <chr>           <dbl>
## 1 May                26       2012 Sep                14
## 2 Oct                10       2012 Mar                17
## 3 Nov                22       2011 Oct                19
## 4 Mar                14       2012 Nov                19
## 5 Jun                27       2012 Jan                13
## # ℹ 16 more variables: age_at_death <dbl>, gender <chr>,
## #   education_level <chr>, occupation <chr>, …
```

See the appendix at the bottom for the data dictionary describing all variables.

What are Factors?

Factors are an important data class in R used to represent categorical variables.

A categorical variable takes on a limited set of possible values or levels. For example, country, race or political affiliation. These differ from free-form string variables that take arbitrary values, like person names, book titles or doctor's comments.

RECAP

Review of the Main Data Classes in R

- `Numeric`: Represents continuous numerical data, including decimal numbers.
- `Integer`: Specifically for whole numbers without decimal places.
- `Character`: Used for text or string data.
- `Logical`: Represents boolean values (TRUE or FALSE).
- `Factor`: Used for categorical data with predefined levels or categories.
- `Date`: Represents dates without times.

Factors have a few key advantages over character vectors for working with categorical data in R:

- Factors are stored in R slightly more efficiently than characters.
- Certain statistical functions, such as `lm()`, require categorical variables to be input as factors
- Factors allow control over the order of categories or levels. This allows properly sorting and plotting of categorical data.

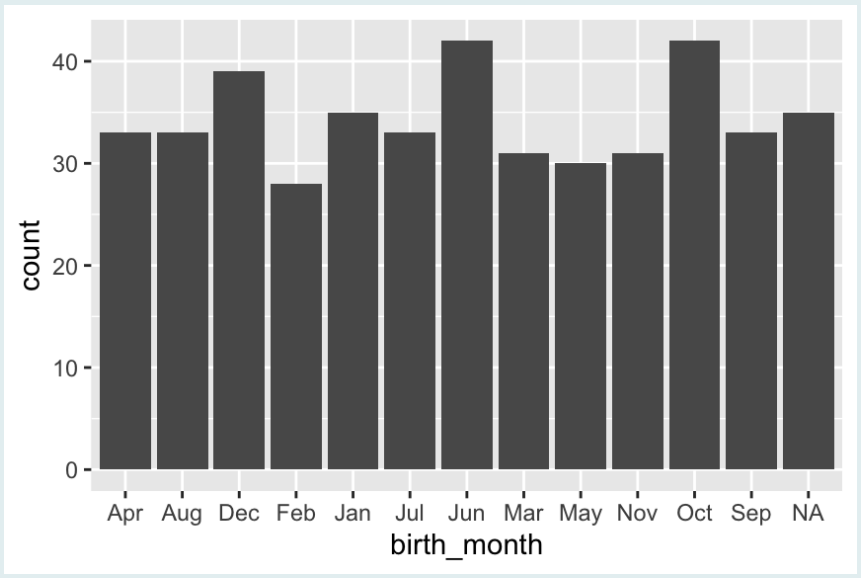
This last point, controlling the order of factor levels, will be our primary focus.

Factors in Action

Let's see a practical example of the value of factors using the `hiv_mort` dataset we loaded above.

Suppose you are interested in visualizing the patients in the dataset by their birth month. We can do this with ggplot:

```
ggplot(hiv_mort) +
  geom_bar(aes(x = birth_month))
```



However, there's a hiccup: the x-axis (representing the months) is arranged alphabetically, with April first on the left, then August, and so on. But months should follow a specific chronological order!

We can arrange the plot in the desired order by creating a factor using the `factor()` function:

```
hiv_mort_modified <-
  hiv_mort %>%
  mutate(birth_month = factor(x = birth_month,
                              levels = c("Jan", "Feb", "Mar", "Apr",
                                         "May", "Jun", "Jul", "Aug",
                                         "Sep", "Oct", "Nov", "Dec")))
```

The syntax is straightforward: the `x` argument takes the original character column, `birth_month`, and the `levels` argument takes in the desired sequence of months.

When we inspect the data type of the birth_month variable, we can see its transformation:

```
# Modified dataset
class(hiv_mort_modified$birth_month)
```

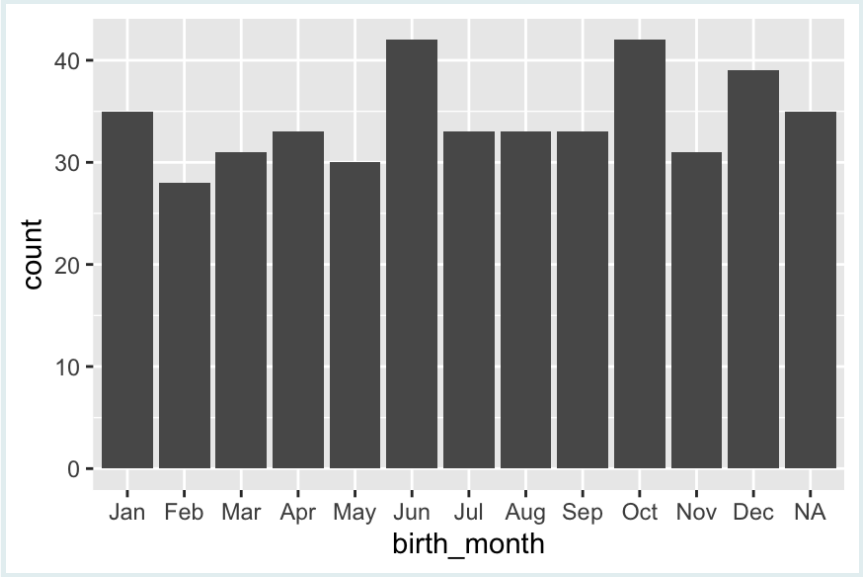
```
## [1] "factor"
```

```
# Original dataset
class(hiv_mort$birth_month)
```

```
## [1] "character"
```

Now we can regenerate the ggplot with the modified dataset:

```
ggplot(hiv_mort_modified) +
  geom_bar(aes(x = birth_month))
```



The months on the x-axis are now displayed in the order we specified.

The new factor variable will respect the defined order in other contexts as well. For example, compare how the `count()` function displays the two frequency tables below:

```
# Original dataset
count(hiv_mort, birth_month)
```

```
## # A tibble: 13 × 2
##    birth_month     n
##    <chr>       <int>
##  1 Apr            33
##  2 Aug            33
##  3 Dec            39
##  4 Feb            28
##  5 Jan            35
##  6 Jul            33
##  7 Jun            42
##  8 Mar            31
##  9 May            30
## 10 Nov            31
## 11 Oct            42
```

```
## 12 Sep             33
## 13 <NA>            35
```

```
# Modified dataset
count(hiv_mort_modified, birth_month)
```

```
## # A tibble: 13 × 2
##    birth_month     n
##    <fct>       <int>
##  1 Jan            35
##  2 Feb            28
##  3 Mar            31
##  4 Apr            33
##  5 May            30
##  6 Jun            42
##  7 Jul            33
##  8 Aug            33
##  9 Sep            33
## 10 Oct            42
## 11 Nov            31
## 12 Dec            39
## 13 <NA>           35
```

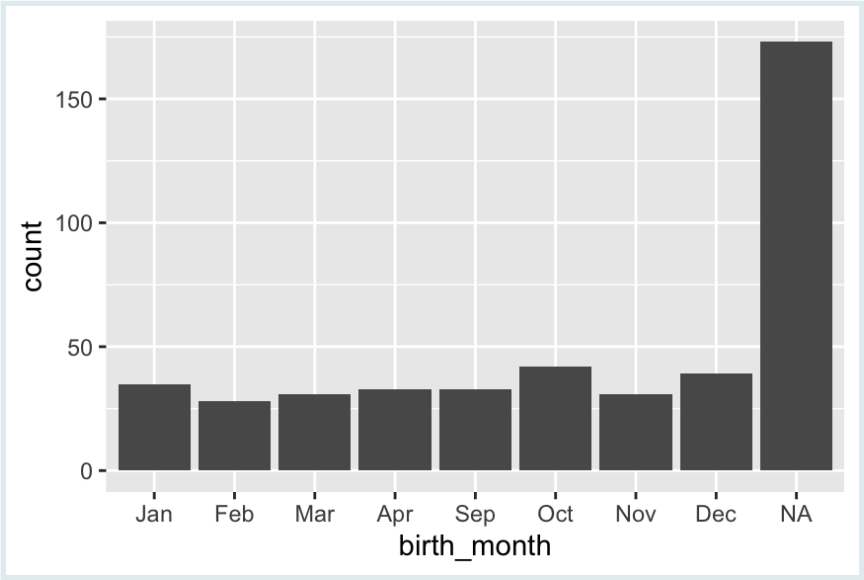
Be mindful when creating factor levels! Any values in the variable that are *not* included in the set of levels provided to the `levels` argument will be converted to NA.

For instance, if we missed some months in our example:

```
hiv_mort_missing_months <-
  hiv_mort %>%
  mutate(birth_month = factor(x = birth_month,
                              levels = c("Jan", "Feb", "Mar",
         "Apr",
                                         # missing months
                                        "Sep", "Oct", "Nov",
         "Dec")))
```

We end up with a lot of NA values:

```
ggplot(hiv_mort_missing_months) +
  geom_bar(aes(x = birth_month))
```

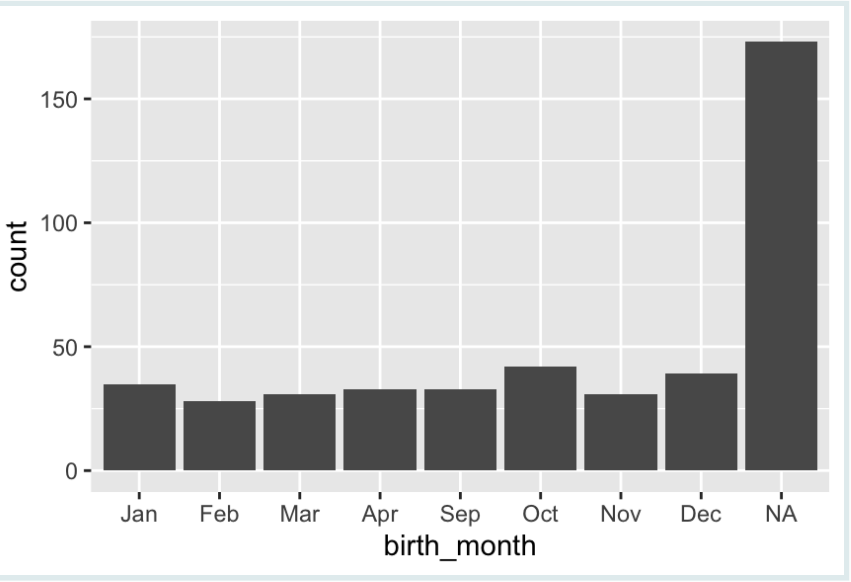


You will have the same problem if there are typographical errors:

```
hiv_mort_with_typos <-
  hiv_mort %>%
  mutate(birth_month = factor(x = birth_month,
                              levels = c("Jan", "Feb", "Mar",
         "Apr",
                                         "Moy", "Jon", "Jol",
         "Aog", # typos
                                         "Sep", "Oct", "Nov",
         "Dec")))
```

```
ggplot(hiv_mort_with_typos) +
  geom_bar(aes(x = birth_month))
```

WATCH OUT



You can use factor without levels. It just uses default (alphabetical) arrangement of levels

```
hiv_mort_default_factor <- hiv_mort %>%
  mutate(birth_month = factor(x = birth_month))
```

SIDE NOTE

```
class(hiv_mort_default_factor$birth_month)
```

```
## [1] "factor"
```

```
levels(hiv_mort_default_factor$birth_month)
```

```
##  [1] "Apr" "Aug" "Dec" "Feb" "Jan" "Jul" "Jun" "Mar"
##  [9] "May" "Nov" "Oct" "Sep"
```

Q: Gender factor

Using the `hiv_mort` dataset, convert the `gender` variable to a factor with the levels "Female" and "Male", in that order.

Q: Error spotting

What errors are you able to spot in the following code chunk? What are the consequences of these errors?

```
hiv_mort <-
  hiv_mort %>%
  mutate(birth_month = factor (x = birth_month,
                               levels = c("Jan", "Feb", "Mar", "Apr",
                                          "Mai", "Jun", "Jul", "Sep",
                                          "Oct", "Nov.", "Dec")))
```

Q: Advantage of factors

What is one main advantage of using factors over characters for categorical data in R?

a. It is easier to perform string manipulation on factors.
b. Factors allow better control over ordering of categorical data.
c. Factors increase the accuracy of statistical models.

Manipulating Factors with `forcats`

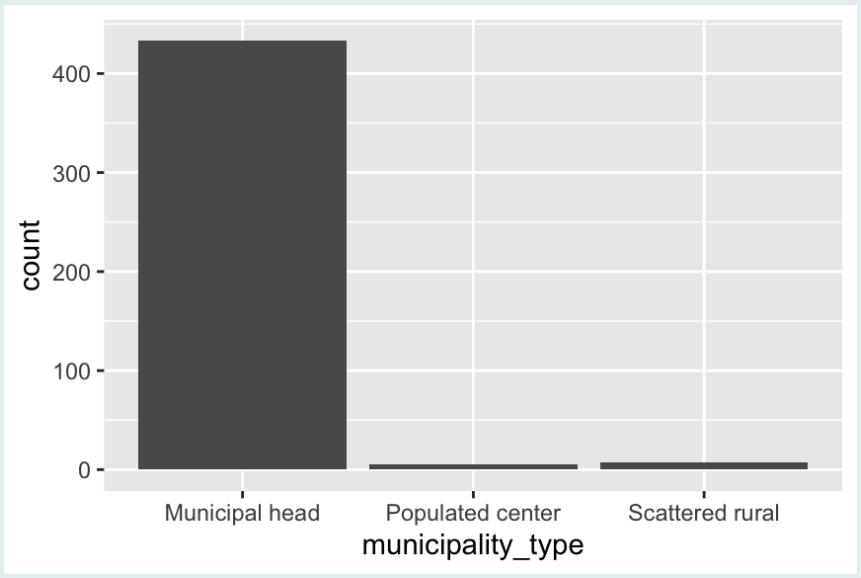
Factors are very useful, but they can sometimes be a little tedious to manipulate using base R functions alone. Thankfully, the `forcats` package, a member of the tidyverse, offers a set of functions that make factor manipulation much simpler. We'll consider four functions here, but there are many others, so we encourage you to explore the *forcats* website on your own time here!

fct_relevel

The `fct_relevel()` function is used to manually change the order of factor levels.

For example, let's say we want to visualize the frequency of individuals in our dataset by municipality type. When we create a bar plot, the values are ordered alphabetically by default:

```
ggplot(hiv_mort) +
  geom_bar(aes(x = municipality_type))
```



But what if we want a specific value, say "Populated center", to appear first in the plot?

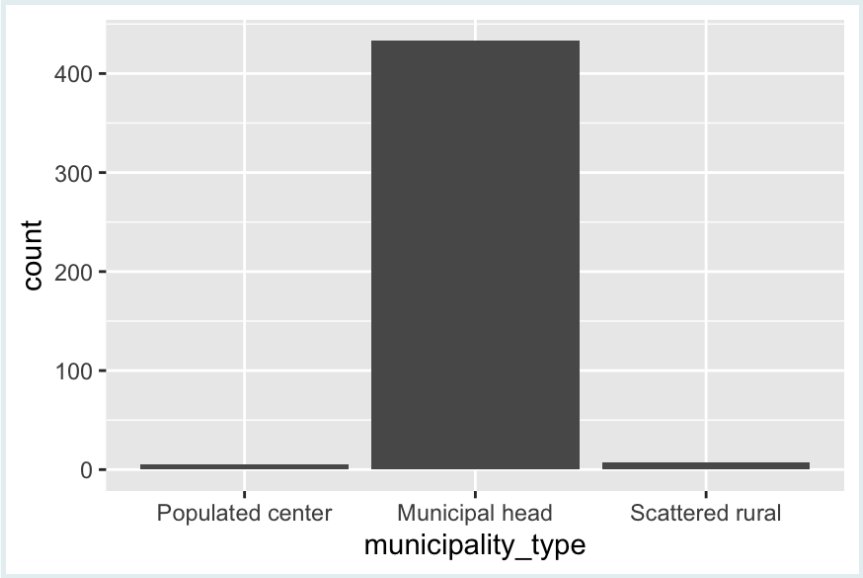
This can be achieved using `fct_relevel()`. Here's how:

```
hiv_mort_pop_center_first <-
  hiv_mort %>%
  mutate(municipality_type = fct_relevel(municipality_type, "Populated
         center"))
```

The syntax is straightforward: we pass the factor variable as the first argument, and the level we want to move to the front as the second argument.

Now when we plot:

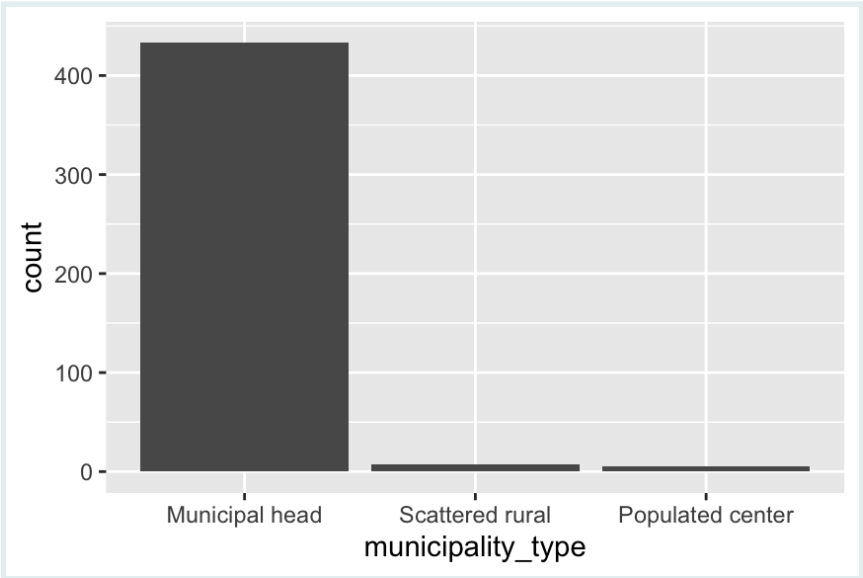
```
ggplot(hiv_mort_pop_center_first) +
  geom_bar(aes(x = municipality_type))
```



The "Populated center" level is now first.

We can move the "Populated center" level to a different position with the `after` argument:

```
hiv_mort %>%
  mutate(municipality_type = fct_relevel(municipality_type, "Populated
         center",
                                         after = 2)) %>%
 # pipe directly into to plot to visualize change
  ggplot() +
  geom_bar(aes(x = municipality_type))
```

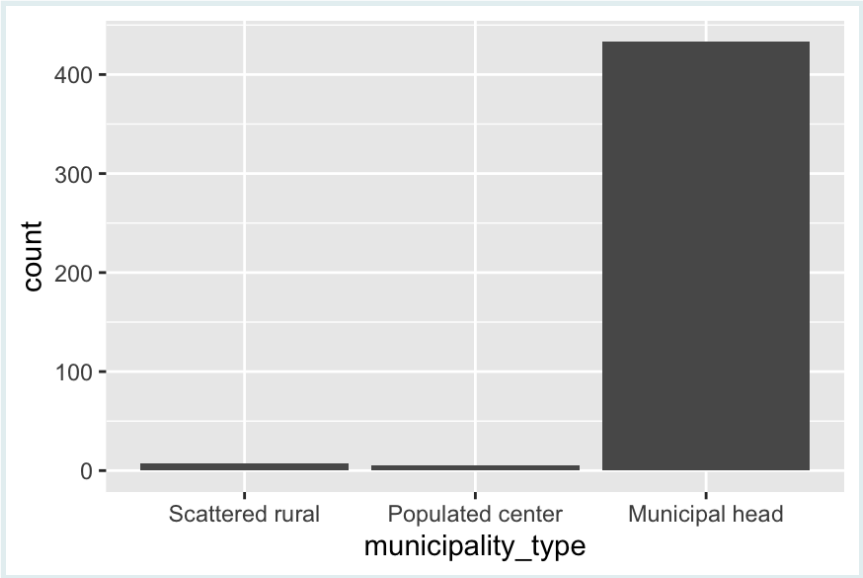


The syntax is: specify the factor, the level to move, and use the `after` argument to define what position to place it after.

We can also move multiple levels at a time by providing these levels to `fct_relevel()`:

Below we arrange all the factor levels for municipality type in our desired order:

```
hiv_mort %>%
  mutate(municipality_type = fct_relevel(municipality_type,
                                         "Scattered rural",
                                         "Populated center",
                                         "Municipal head")) %>%
  ggplot() +
  geom_bar(aes(x = municipality_type))
```



This is similar to creating a factor from scratch with levels in that order:

```
hiv_mort %>%
  mutate(municipality = factor(municipality_type,
                               levels = c("Scattered rural",
                                          "Populated center",
                                          "Municipal head")))
```

Q: Using fct_relevel

Using the `hiv_mort` dataset, convert the `death_location` variable to a factor such that 'Home/address' is the first level. Then create a bar plot that shows the count of individuals in the dataset by `death_location`.

fct_reorder

`fct_reorder()` is used to reorder the levels of a factor based on the values of another variable.

To illustrate, let's make a summary table with number of deaths, mean and median age at death for each municipality:

```
summary_per_muni <-
  hiv_mort %>%
  group_by(municipality_name) %>%
  summarise(n_deceased = n(),
            mean_age_death = mean(age_at_death, na.rm = T),
            med_age_death = median(age_at_death, na.rm = T))

summary_per_muni
```

```
## # A tibble: 25 × 4
##    municipality_name n_deceased mean_age_death
##    <chr>                  <int>          <dbl>
##  1 Aguadas                    2           42
##  2 Anserma                   15           37.4
##  3 Aranzazu                   2           37.5
##  4 Belalcázar                 4           38.8
##  5 Chinchiná                 62           43.6
##  6 Filadelfia                 5           42.6
##  7 La Dorada                 46           41.0
##  8 La Merced                  3           27
##  9 Manizales                199           41.0
## 10 Manzanares                 3           38.3
##    med_age_death
##            <dbl>
##  1          42
##  2          37.5
##  3          37.5
##  4          41
##  5          42.5
##  6          43
##  7          41
##  8          28
##  9          41
## 10          34
## # ℹ 15 more rows
```

When plotting one of the variables, we may want to arrange the factor levels by that numeric variable. For example, to order municipality by the mean age column:

```
summary_per_muni_reordered <-
  summary_per_muni %>%
```

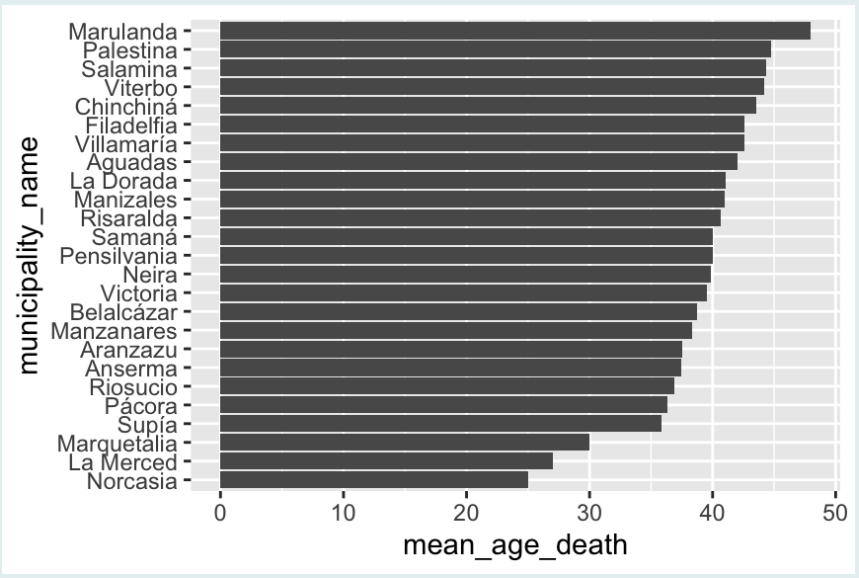
```
  mutate(municipality_name = fct_reorder(.f = municipality_name,
                                         .x = mean_age_death))
```

The syntax is:

- `.f` - the factor to reorder
- `.x` - the numeric vector determining the new order

We can now plot a nicely arranged bar chart:

```
ggplot(summary_per_muni_reordered) +
  geom_col(aes(y = municipality_name, x = mean_age_death))
```



Q: Using fct_reorder

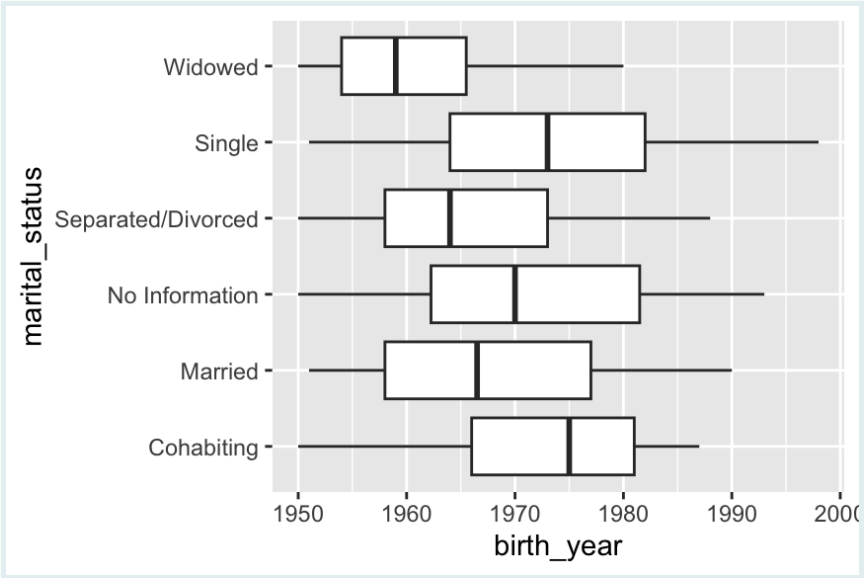
Starting with the `summary_per_muni` data frame, reorder the municipality (`municipality_name`) by the `med_age_death` column and plot the reordered bar chart.

The `.fun` argument

Sometimes we want the categories in our plot to appear in a specific order that is determined by a summary statistic. For example, consider the box plot of `birth_year` by `marital_status`:

```
ggplot(hiv_mort, aes(y = marital_status, x = birth_year)) +
```

```
  geom_boxplot()
```



The boxplot displays the median `birth_year` for each category of marital status as a line in the middle of each box. We might want to arrange the marital_status categories in order of these medians. But if we create a summary table with medians, like we did before with `summary_per_muni`, we can't create a box plot with it (go look at the `summary_per_muni` data frame to verify this yourself).

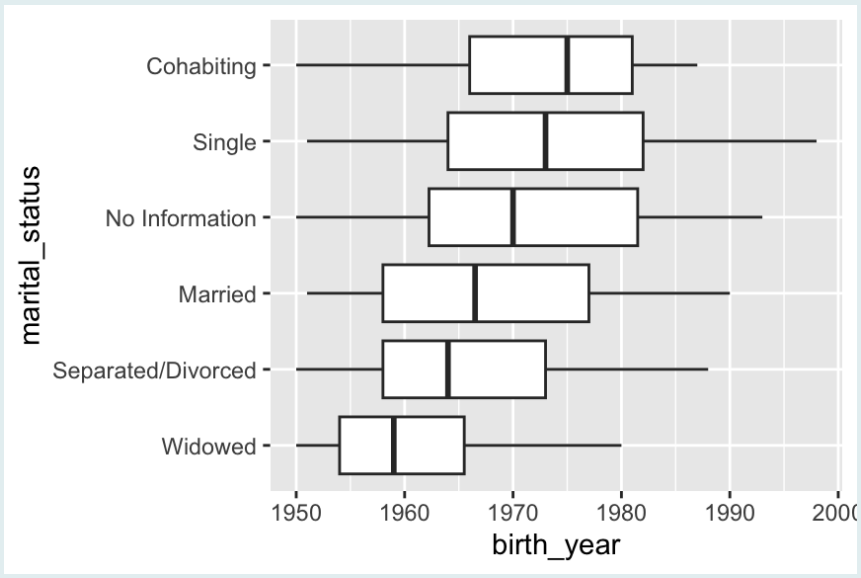
This is where the `.fun` argument of `fct_reorder()` comes in. The `.fun` argument allows us to specify a summary function that will be used to calculate the new order of the levels:

```
hiv_mort_arranged_marital <-
  hiv_mort %>%
  mutate(marital_status = fct_reorder(.f = marital_status,
                                      .x = birth_year,
                                      .fun = median,
                                      na.rm = TRUE))
```

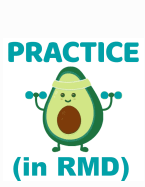
In this code, we are reordering the `marital_status` factor based on the median of `birth_year`. We include the argument `na.rm = TRUE` to ignore NA values when calculating the median.

Now, when we create our box plot, the `marital_status` categories are ordered by the median `birth_year`:

```
ggplot(hiv_mort_arranged_marital, aes(y = marital_status, x = birth_year)) +
  geom_boxplot()
```



We can see that individuals with the marital status "cohabiting" tend to be the youngest (they were born in the latest years).



Q: Using .fun

Using the `hiv_mort` dataset, make a boxplot of `birth_year` by `health_insurance_status`, where the `health_insurance_status` categories are arranged by the median `birth_year`.

fct_recode

The `fct_recode()` function allows us to manually change the values of factor levels. This function can be especially helpful when you need to rename categories or when you want to merge multiple categories into one.

For example, we can rename 'Municipal head' to 'City' in the `municipality_type` variable:

```
hiv_mort_muni_recode <- hiv_mort %>%
  mutate(municipality_type = fct_recode(municipality_type,
                                        "City" = "Municipal head"))

# View the change
levels(hiv_mort_muni_recode$municipality_type)
```

```
## [1] "City"             "Populated center"
```

```
## [3] "Scattered rural"
```

In the above code, `fct_recode()` takes two arguments: the factor variable you want to change (`municipality_type`), and the set of name-value pairs that define the recoding. The new level ("City") is on the left of the equals sign, and the old level ("Municipal head") is on the right.

`fct_recode()` is particularly useful for compressing multiple categories into fewer levels:

We can explore this using the education_level variable. Currently it has six categories:

```
count(hiv_mort, education_level)
```

```
## # A tibble: 6 × 2
##   education_level     n
##   <chr>           <int>
## 1 No information     88
## 2 None               22
## 3 Post-secondary     29
## 4 Preschool           3
## 5 Primary           187
## 6 Secondary         116
```

For simplicity, let's group them into just three categories - primary & below, secondary & above and other:

```
hiv_mort_educ_simple <-
  hiv_mort %>%
  mutate(education_level = fct_recode(education_level,
                                      "primary & below" = "Primary",
                                      "primary & below" = "Preschool",
                                      "secondary & above" = "Secondary",
                                      "secondary & above" = "Post-
        secondary",
                                      "others" = "No information",
                                      "others" = "None"))
```

This condenses the categories nicely:

```
count(hiv_mort_educ_simple, education_level)
```

```
## # A tibble: 3 × 2
##   education_level       n
##   <fct>             <int>
## 1 others              110
## 2 secondary & above   145
## 3 primary & below     190
```

For good measure, we can arrange the levels in a reasonable order, with "others" as the last level:

```
hiv_mort_educ_sorted <-
  hiv_mort_educ_simple %>%
  mutate(education_level = fct_relevel(education_level,
                                       "primary & below",
                                       "secondary & above",
                                       "others"))
```

This condenses the categories nicely:

```
count(hiv_mort_educ_sorted, education_level)
```

```
## # A tibble: 3 × 2
##   education_level       n
##   <fct>             <int>
## 1 primary & below     190
## 2 secondary & above   145
## 3 others              110
```

Q: Using fct_recode

Using the `hiv_mort` dataset, convert death_location to a factor.

Then use `fct_recode()` to rename 'Public way' in death_location to 'Public place'. Plot the frequency counts of the updated variable.

fct_recode vs case_when/if_else

You might question why we need `fct_recode()` when we can utilize `case_when()` or `if_else()` or even `recode()` to substitute specific values. The issue is that these other functions can disrupt your factor variable.

To illustrate, let's say we choose to use `case_when()` to make a modification to the `education_level` variable of the `hiv_mort_educ_sorted` data frame.

As a quick reminder, that the `education_level` variable is a factor with three levels, arranged in a specified order, with "primary & below" first and "others" last:

```
count(hiv_mort_educ_sorted, education_level)
```

```
## # A tibble: 3 × 2
##   education_level       n
##   <fct>             <int>
## 1 primary & below     190
## 2 secondary & above   145
## 3 others              110
```

Say we wanted to replace the "others" with "other", removing the "s". We can write:

SIDE NOTE

```
hiv_mort_educ_other <-
  hiv_mort_educ_sorted %>%
  mutate(education_level = if_else(education_level ==
         "others",
                                   "other", education_level))
```

After this operation, the variable is no longer a factor:

```
class(hiv_mort_educ_other$education_level)
```

```
## [1] "character"
```

If we then create a table or plot, our order is disrupted and reverts to alphabetical order, with "other" as the first level:

```
count(hiv_mort_educ_other, education_level)
```

```
## # A tibble: 3 × 2
##   education_level       n
##   <chr>             <int>
## 1 other               110
## 2 primary & below     190
## 3 secondary & above   145
```

However, if we had used `fct_recode()` for recoding, we wouldn't face this issue:

```
hiv_mort_educ_other_fct <-
  hiv_mort_educ_simple %>%
  mutate(education_level = fct_recode(education_level, "other"
         = "others"))
```

The variable remains a factor:

```
class(hiv_mort_educ_other_fct$education_level)
```

SIDE NOTE

```
## [1] "factor"
```

And if we create a table or a plot, our order is preserved: primary, secondary, then other:

```
count(hiv_mort_educ_other_fct, education_level)
```

```
## # A tibble: 3 × 2
##   education_level       n
##   <fct>             <int>
## 1 other               110
## 2 secondary & above   145
## 3 primary & below     190
```

fct_lump

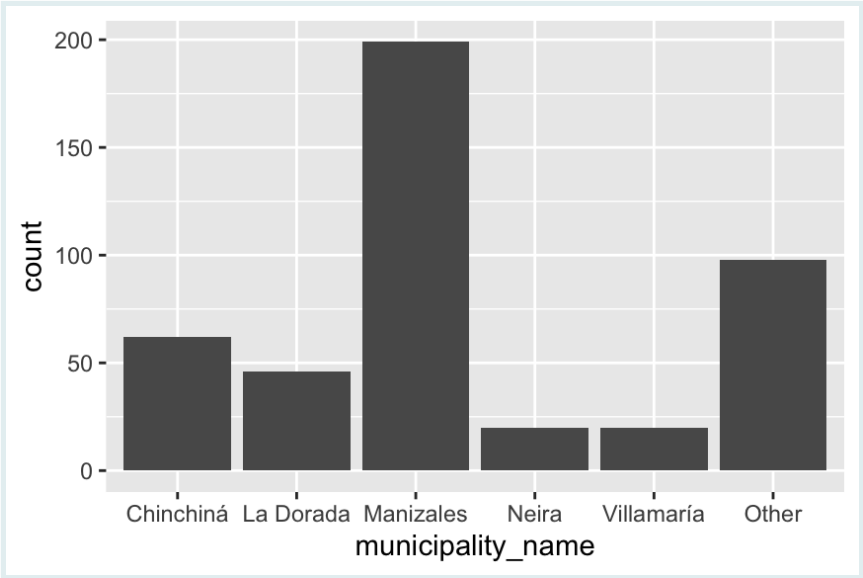
Sometimes, we have too many levels for a display table or plot, and we want to lump the least frequent levels into a single category, typically called 'Other'.

This is where the convenience function `fct_lump()` comes in.

In the below example, we lump less frequent municipalities into 'Other', preserving just the top 5 most frequent municipalities:

```
hiv_mort_lump_muni <- hiv_mort %>%
  mutate(municipality_name = fct_lump(municipality_name, n = 5))
```

```
ggplot(hiv_mort_lump_muni, aes(x = municipality_name)) +  geom_bar()
```

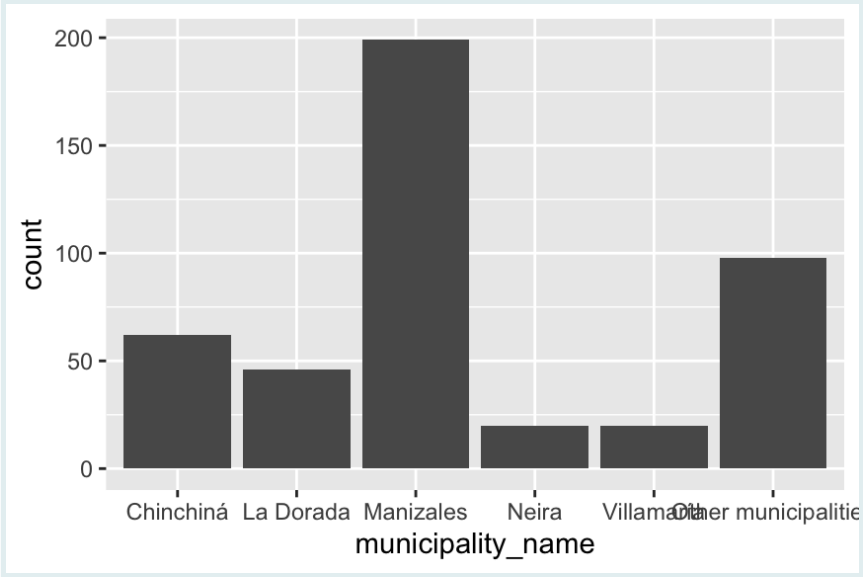


In the usage above, the parameter `n = 5` means that the five most frequent municipalities are preserved, and the rest are lumped into 'Other'.

We can provide a custom name for the other category with the `other_level` argument. Below we use the name `"Other municipalities"`.

```
hiv_mort_lump_muni_other_name <- hiv_mort %>%
  mutate(municipality_name = fct_lump(municipality_name, n = 5,
                                      other_level = "Other municipalities"))

ggplot(hiv_mort_lump_muni_other_name, aes(x = municipality_name)) +
  geom_bar()
```



In this way, `fct_lump()` is a handy tool for condensing factors with many infrequent levels into a more manageable number of categories.

Q: Using fct_lump

Starting with the `hiv_mort` dataset, use `fct_lump()` to create a bar chart with the frequency of the 10 most common occupations.

Lump the remaining occupation into an 'Other' category.

Put `occupation` on the y-axis, not the x-axis, to avoid label overlap.

Wrap up

Congrats on getting to the end. In this lesson, you learned details about the data class, **factors**, and how to manipulate them using basic operations such as `fct_relevel()`, `fct_reorder()` `fct_recode()`, and `fct_lump()`.

While these covered common tasks such as reordering, recoding, and collapsing levels, this introduction only scratches the surface of what's possible with the forcats package. Do explore more on the forcats website.

Now that you understand the basics of working with factors, you are equipped to properly represent your categorical data in R for downstream analysis and visualization.

Answer Key

Q: Gender factor

```
hiv_mort_q1 <- hiv_mort %>%
  mutate(gender = factor(x = gender,
                         levels = c("Female", "Male")))
```

Q: Error spotting

Errors:

- "Mai" should be "May".
- "Nov." has an extra period.
- "Aug" is missing from the list of months.

Consequences:

Any rows with the values "May", "Nov", or "Aug" for death_month will be converted to NA in the new death_month variable. If you create plots, ggplot will drop these levels with only NA values.

Q: Advantage of factors

b. Factors allow better control over ordering of categorical data.

The other two statements are not true.

If you want to apply string operations like substr(), strsplit(), paste(), etc., it's actually more straightforward to use character vectors than factors.

And while many statistical functions expect factors, not characters, for categorical predictors, this does not make them more "accurate".

Q: Using fct_relevel

Q: Using fct_reorder

Q: Using .fun

Q: Using fct_recode

Q: Using fct_lump

Appendix: Codebook

The variables in the dataset are:

- `municipality`: general municipal location of the patient [chr]
- `death_location`: location where the patient died [chr]

- `birth_date`: full date of birth, formatted "YYYY-MM-DD" [date]
- `birth_year`: year when the patient was born [dbl]
- `birth_month`: month when the patient was born [chr]
- `birth_day`: day when the patient was born [dbl]
- `death_year`: year when the patient died [dbl]
- `death_month`: month when the patient died [chr]
- `death_day`: day when the patient died [dbl]
- `gender`: gender of the patient [chr]
- `education_level`: highest level of education attained by patient [chr]
- `occupation`: occupation of patient [chr]
- `racial_id`: race of the patient [chr]
- `municipality_code`: specific municipal location of the patient [chr]
- `primary_cause_death_description`: primary cause of the patient's death [chr]
- `primary_cause_death_code`: code of the primary cause of death [chr]
- `secondary_cause_death_description`: secondary cause of the patient's death [chr]
- `secondary_cause_death_code`: code of the secondary cause of death [chr]
- `tertiary_cause_death_description`: tertiary cause of the patient's death [chr]
- `tertiary_cause_death_code`: code of the tertiary cause of death [chr]
- `quaternary_cause_death_description`: quaternary cause of the patient's death [chr]
- `quaternary_cause_death_code`: code of the quaternary cause of death [chr]

Contributors

The following team members contributed to this lesson:

CAMILLE BEATRICE VALERA

Project Manager and Scientific Collaborator, The GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

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:

```
# 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
```

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

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")
```

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()`.

```
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.

```
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:

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)`
```

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`.

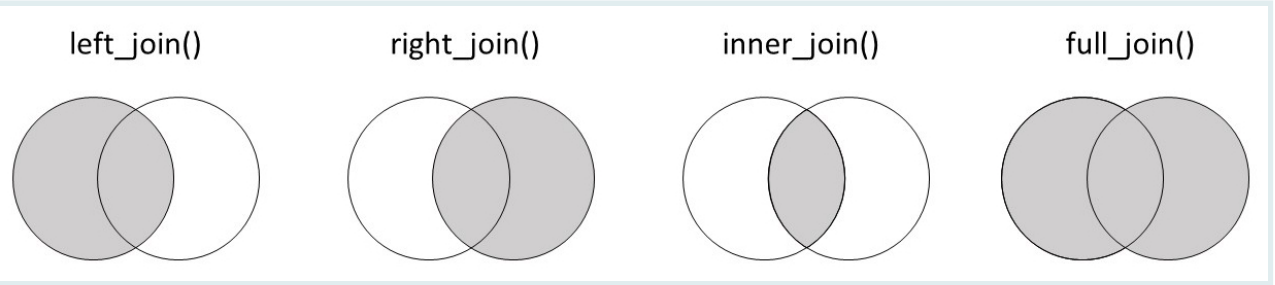
Just as for the previous exercise, obtain all countries and their corresponding region and continent from the {countrycode} package:

```
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!



Joining 2: Joining Real-World Datasets

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

Introduction

Now that we have a solid grasp on the different types of joins and how they work, we can look at how to manage messier and more complex datasets. Joining real-world data from different sources often requires a bit of thought and cleaning ahead of time.

Learning Objectives

- You know how to check for mismatched values between dataframes
- You understand how to join using a one-to-many match
- You know how to join on multiple key columns

Packages

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

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

Pre-cleaning data

Often you will need to pre-clean your data when you draw it from different sources before you're able to join it. This is because there can be discrepancies in ways that values are recorded in different tables such as spelling errors, differences in capitalization, and extra

spaces. In order to join values, we need them to match perfectly. If there are any differences, R considers them to be different values.

To illustrate this, let's return to our mock patient data from the first lesson. If you recall, we had two dataframes, one called `demographic` and the other called `test_info`. We can recreate these datasets but change `Alice` to `alice` in the `demographic` dataset and keep all other values the same.

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

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

Now let's try an `inner_join()` on our two datasets.

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

As we can see, R didn't recognize `Alice` and `alice` as the same person, so the only common value between the datasets was `Bob`. How do we deal with this? Well, there are multiple functions we can use to modify our string values. In this case, using `str_to_title()` would work to ensure that all values are the same. If we apply this to our `name` column in our `demographic` dataset, we'll be able to join the tables appropriately.

```
demographic <- demographic %>%
  mutate(name=str_to_title(name))
demographic
```

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

That worked perfectly! We won't go into detail about all the different functions we can use to modify strings, since they're covered extensively in the strings lesson. The important part of this lesson is that we will learn how to identify mismatched values between dataframes.

The following two datasets contain data for India, Indonesia, and the Philippines. What are the differences between the values in the key columns that would have to be changed before joining the datasets?

```
df1 <- tribble(
  ~Country,      ~Capital,
  "India",       "New Delhi",
  "Indonesia",   "Jakarta",
  "Philippines", "Manila"
)

df2 <- tribble(
  ~Country,      ~Population,    ~Life_Expectancy,
  "India ",       1393000000,    69.7,
  "indonesia",    273500000,     71.7,
  "Philipines",   113000000,     72.7
)
```

In small datasets such as our mock data above, it's quite easy to notice the differences between values in our key columns. But what if we have much bigger datasets? To illustrate this, let's take a look at two real-world datasets on TB in India.

Our first dataset contains data on notification of TB cases in 2022 for all Indian states and Union Territories, taken from the Government of India Tubrculosis Report. Our variables include the state/Union Territory name, the type of healthcare system the patients were detected in (pulic or private), the target number of patients to be notified of their TB status, and the actual number of TB patients notified.

```
notification <- read_csv(here("data/notif_TB_india.csv"))
```

```
notification
```

```
## # A tibble: 5 × 4
##   state_UT                  hc_type target_notify
##   <chr>                     <chr>           <dbl>
## 1 Andaman & Nicobar Islands public            520
## 2 Andaman & Nicobar Islands private            10
## 3 Andhra Pradesh            public           85000
## 4 Andhra Pradesh            private          30000
## 5 Arunachal Pradesh         public            3450
##   actual_notified
##             <dbl>
## 1             510
## 2              24
## 3           62075
## 4           30112
## 5            2722
```

Our second dataset, taken from the same TB Report, contains the state/Union Territory name, the type of healthcare system, the number of TB patients screened for COVID, and the number of TB patients diagnosed with COVID.

```
covid <- read_csv(here("data/COVID_TB_india.csv"))
```

```
covid
```

```
## # A tibble: 5 × 4
##   state_UT                  hc_type covid_screened
##   <chr>                     <chr>            <dbl>
## 1 Andaman & Nicobar Islands public             322
## 2 Andaman & Nicobar Islands private              1
## 3 Andhra Pradesh            public           63319
## 4 Andhra Pradesh            private          26410
## 5 ArunachalPradesh          public            1761
##   covid_dx
##      <dbl>
## 1        0
## 2        0
## 3       97
## 4       17
## 5        0
```

For the sake of this lesson, we've modified some of the state/Union Territory names in the `covid` dataset. Our goal is to have them match the names from our `notification` dataset so to do this we need to compare values between them. For large dataframes, if we want to compare which values are in one but not the other, we can use the `setdiff()` function and state which dataframes & columns we want to compare. Let's start by comparing the `state_UT` values from `notification` dataframe to the `state_UT` values from the `covid` dataframe.

```
setdiff(notification$state_UT, covid$state_UT)
```

```
## [1] "Arunachal Pradesh"
## [2] "Dadra and Nagar Haveli and Daman and Diu"
## [3] "Tamil Nadu"
## [4] "Tripura"
```

So what does the list above tell us? Well by putting the `notification` dataset first, we are asking R "which values are in `notification` but *not* in `covid`?" We can (and should!) also switch the order of the datasets to check the reverse, asking "which values are in `covid` but not in `notification`?" Let's do this and compare the two lists.

```
setdiff(covid$state_UT, notification$state_UT)
```

```
## [1] "ArunachalPradesh"
## [2] "Dadra & Nagar Haveli and Daman & Diu"
## [3] "tamil nadu"
## [4] "Tri pura"
```

`covid` data using the `case_when()` function to have our two dataframes match. Let's clean this up and then compare our datasets again.

```
covid <- covid %>%
  mutate(state_UT =
           case_when(state_UT == "ArunachalPradesh" ~ "Arunachal Pradesh",
                     state_UT == "tamil nadu" ~ "Tamil Nadu",
                     state_UT == "Tri pura" ~ "Tripura",
                     state_UT == "Dadra & Nagar Haveli and Daman & Diu" ~
         "Dadra and Nagar Haveli and Daman and Diu",
                     TRUE ~ state_UT))

setdiff(notification$state_UT, covid$state_UT)
```

```
## character(0)
```

```
setdiff(covid$state_UT, notification$state_UT)
```

```
## character(0)
```

REMINDER

For the sake of illustration purposes, we've re-written the original values in our `covid` dataframe. However, in practice whenever you're transforming your variables, it's always best to create a new clean variable and drop old ones later if you no longer need them!

Great! As we can see, there are no longer differences in values between our dataframes. Now that we've ensured that our data is clean, we can move on to joining! Since we understand joining basics from our first lesson, we can move onto more complex topics.

The following dataframe, also taken from the TB Report, contains information on the number of pediatric TB cases and the number of pediatric patients initiated on treatment.

```
child <- read_csv(here("data/child_TB_india.csv"))
```

```
child
```

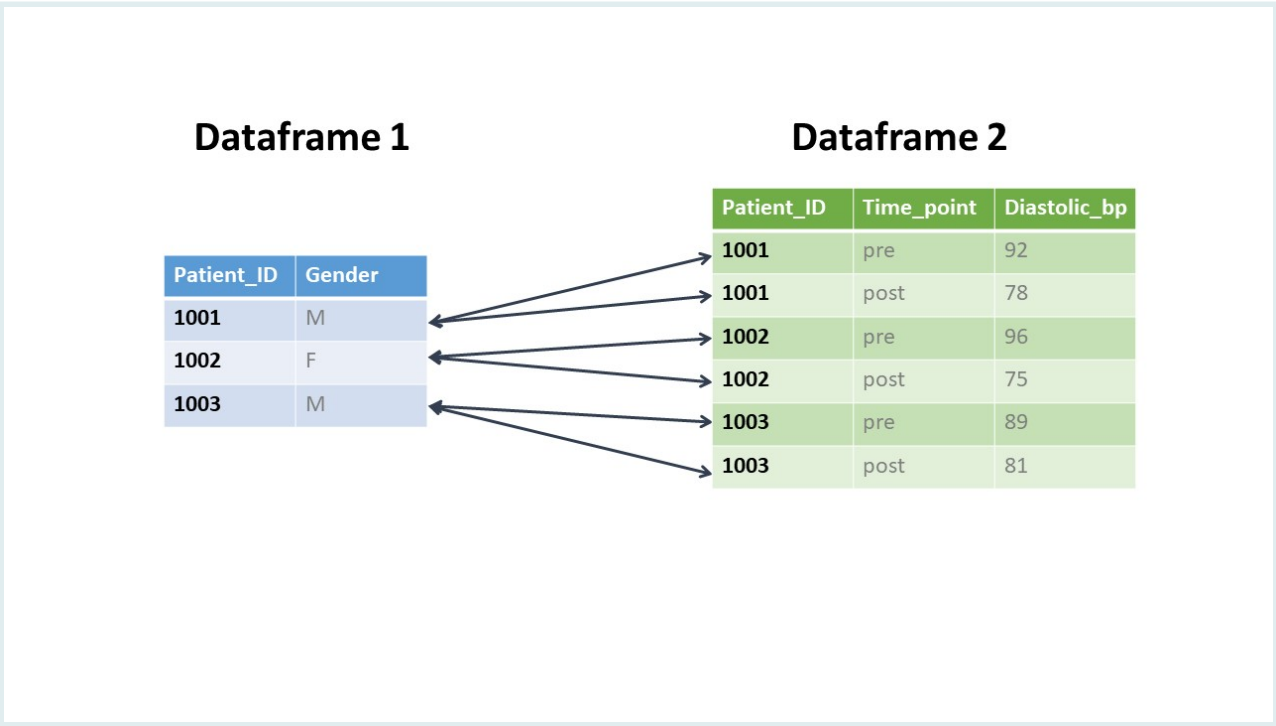
```
## # A tibble: 5 × 4
##   state_UT                  hc_type child_notify
##   <chr>                     <chr>          <dbl>
## 1 Andaman & Nicobar Islands public            18
## 2 Andaman & Nicobar Islands private            1
## 3 Andhra Pradesh            public          1347
## 4 Andhra Pradesh            private         1333
## 5 Arunachal Pradesh         public           256
```

```
##   child_treatment
##              <dbl>
## 1              19
## 2               0
## 3            1684
## 4             993
## 5             282
```

Using `set_diff()` compare the merging values from the `child` dataframe with those from the `notification` dataframe and make any necessary changes to the `child` dataframe to ensure that the values match.

One-to-many relationships

In the previous lesson, we looked at one-to-one joins, where an observation in one dataframe corresponded to no more than one observation in the other dataframe. In a one-to-many join, an observation one dataframe corresponds to multiple observations in the other dataframe. The image below illustrates this concept:



Let's take a look at a one-to-many join with a `left_join()` first!

`left_join()`

To illustrate a one-to-many join, let's return to our patients and their COVID test data. Let's imagine that in our dataset, `Alice` and `Xavier` got tested multiple times for COVID.

We can add two more rows to our `test_info` dataframe with their new test information:

```
test_info_many <- tribble(
  ~name,    ~test_date, ~result,
  "Alice",  "2023-06-05", "Negative",
  "Alice",  "2023-06-10", "Positive",
  "Bob",    "2023-08-10", "Positive",
  "Xavier", "2023-05-02", "Negative",
  "Xavier", "2023-05-12", "Negative",
)
```

Next, let's take a look at what happens when we use a `left_join()`, first with `demographic` as the dataset to the left of the call:

```
left_join(demographic, test_info_many)
```

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

What´s happened above? Well as we know, `Alice` was in our left dataframe so that row was retained. But she featured twice in the right dataset, so her demographic information was duplicated in the final dataset.`Xavier` wasn't in the left dataframe, so he was dropped entirely. Basically, when you perform a one-to-many join, the data from the "one" side are duplicated for each matching row of the "many" side. The graphic below illustrates this process:

We can see the same thing happen when the order of the datasets are switched, putting `test_info_many` to the left of the call..

```
left_join(test_info_many, demographic)
```

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

Once again, the demographic data for `Alice` has been duplicated! `Xavier` was in the left dataframe `test_info_many` so his rows were kept, but since he wasn't in the `demographic` dataframe, those cells are set to `NA`.

Copy the code below to create two small dataframes:

```
patient_info <- tribble(
  ~patient_id, ~name,      ~age,
  1,           "Liam",     32,
  2,           "Manny",    28,
  3,           "Nico",     40
)

conditions <- tribble(
  ~patient_id, ~disease,
  1,           "Diabetes",
  1,           "Hypertension",
  2,           "Asthma",
  3,           "High Cholesterol",
  3,           "Arthritis"
)
```

If you use a `left_join()` to join these datasets, how many rows will be in the final dataframe? Try to figure it out and then perform the join to see if you were right!

Let's apply this to our real-world datasets. The first dataset that we'll work with is the `notification` dataset. As reminder, here is what it looks like:

```
notification
```

```
## # A tibble: 5 × 4
##   state_UT                  hc_type target_notify
##   <chr>                     <chr>           <dbl>
## 1 Andaman & Nicobar Islands public            520
## 2 Andaman & Nicobar Islands private            10
## 3 Andhra Pradesh            public           85000
## 4 Andhra Pradesh            private          30000
## 5 Arunachal Pradesh         public            3450
##   actual_notified
##             <dbl>
## 1             510
## 2              24
## 3           62075
## 4           30112
## 5            2722
```

Our second dataset contains 32 of the 36 Indian state and Union Territories, as well as their subdivision category, and the zonal council they're situated in.

```
regions <- read_csv(here("data/region_data_india.csv"))
```

```
regions
```

```
## # A tibble: 5 × 3
##   zonal_council          subdivision_category
##   <chr>                  <chr>
## 1 No Zonal Council       Union Territory
## 2 North Eastern Council  State
## 3 North Eastern Council  State
## 4 Eastern Zonal Council  State
## 5 Northern Zonal Council Union Territory
##   state_UT
##   <chr>
## 1 Andaman & Nicobar Islands
## 2 Arunachal Pradesh
## 3 Assam
## 4 Bihar
## 5 Chandigarh
```

First, we can check if their are any differences between the datasets.

```
setdiff(notification$state_UT, regions$state_UT)
```

```
## [1] "Andhra Pradesh" "Chhattisgarh"   "Ladakh"
## [4] "Tamil Nadu"
```

```
setdiff(regions$state_UT, notification$state_UT)
```

```
## character(0)
```

As we can see, there are four states in the `notification` dataset that are not in the `regions` dataset. These aren't errors that need cleaning, rather we just don't have the complete information in our `regions` dataset. If we want to keep all the `notification` cases, we will put it in the left position of our join. Let's try it out now!

```
notif_regions <- notification %>%
  left_join(regions)
```

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

As expected, the data from the `regions` dataframe was duplicated for every matching value of the `notification` dataframe. For states that aren't in the `regions` dataset, such as `Andhra Pradesh`, the corresponding cells are set to `NA`.

Great! Now we know how to use a `left_join()` when joining datasets with a one-to-many match. Let's take a look at the differences and similarities with an `inner_join()`.

Using a `left_join()`, join the cleaned `child` TB dataset with the `regions` dataset whilst keeping all of the values from the `child` dataframe.

`inner_join()`

When using an `inner_join()` with a one-to-many relationship, the same principles apply as with a `left_join()`. To illustrate this, let's take a look at our COVID patient data and their test info.

```
demographic
```

```
test_info_many
```

Now, let's take a look at what happens when we use an `inner_join()` to join these two datasets.

```
inner_join(demographic, test_info_many)
```

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

With an `inner_join()`, the values that are common between the datasets are kept and those from the "one" side are duplicated for every row of the "many" side. Since `Alice` and `Bob` are common between the two datasets, those are the only ones that are kept. And since `Alice` is featured twice in the `test_info`, her row from the `demographic` dataset is duplicated!

Let's try it out with our `covid` TB dataset, and our `regions` dataset. As a reminder, here are our datasets.

```
covid
```

```
regions
```

As we saw previously, the regions dataset is missing 4 states/Union Territories, so we can expect them to be excluded from our final dataframe with an `inner_join()`. Let's create a new dataframe called `inner_covid_regions`.

```
inner_covid_regions <- covid %>%
  inner_join(regions)
```

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

```
inner_covid_regions
```

Great, that's exactly what we wanted!

Use `set_diff()` to compare values between the `child` and `regions` datasets. Then, use an `inner_join()` to join the two. How many observations are left?

Multiple key columns

Sometimes we have more than one column that uniquely identify the observations that we want to match on. For example, let's imagine that we have systolic and diastolic blood pressure measures for three patients before (pre) and after (post) taking a new blood pressure drug.

```
blood_pressure <- tribble(
  ~name,    ~time_point,  ~systolic,  ~diastolic,
  "David",  "pre",         139,        87,
  "David",  "post",        121,        82,
  "Eamon",  "pre",         137,        86,
  "Eamon",  "post",        128,        79,
  "Flavio", "pre",         137,        81,
  "Flavio", "post",        130,        73
)
blood_pressure
```

Now, let's imagine we have another dataset with the same 3 patients and their serum creatinine levels before and after taking the drug. Creatinine is a waste product that is normally processed by the kidneys. If creatinine levels in the blood increase, it can signify that the kidneys aren't functioning properly, which can be a side effect of blood pressure drugs.

```
kidney <- tribble(
  ~name,    ~time_point,  ~creatinine,
  "David",  "pre",         0.9,
  "David",  "post",        1.3,
  "Eamon",  "pre",         0.7,
  "Eamon",  "post",        0.8,
  "Flavio", "pre",         0.6,
  "Flavio", "post",        1.4
)
kidney
```

We want to join the two datasets so that each patient has two rows, one row for their blood pressure and creatinine levels before taking the drug, and one row for their blood pressure and creatinine levels after the drug. To do this, our first instinct may be to join on the patients name. Let's try it out and see what happens:

```
bp_kidney_dups <- blood_pressure %>%
  left_join(kidney, by="name")
```

```
## Warning in left_join(., kidney, by = "name"): Detected an unexpected many-
to-many relationship between
## `x` and `y`.
## ℹ Row 1 of `x` matches multiple rows in `y`.
## ℹ Row 1 of `y` matches multiple rows in `x`.
## ℹ If a many-to-many relationship is expected, set
##   `relationship = "many-to-many"` to silence this
##   warning.
```

```
bp_kidney_dups
```

As we can see, this isn't what we wanted at all! We can join on the patient names, but R gives a warning message that this is considered a "many-to-many" relationship because multiple rows in one dataframe correspond to multiple rows in the other dataframe, meaning we end up with 4 rows per patient. As a general rule,you should avoid many-to-many joins whenever possible! Also note that since we have two columns called `time_point` (one from each dataframe), these columns in the new dataframe are differentiated by `.x` and `.y`.

What we want to do is match on both `name` and `time_point`. To do this we have to specify to R that there are two columns to match on. In reality, this is very simple! All we have to do is use the `c()` function and specify both column names.

```
bp_kidney <- blood_pressure %>%
  left_join(kidney, by = c("name", "time_point"))
bp_kidney
```

That looks great! Now let's apply this to our real-world `notification` and `covid` datasets.

```
notification
```

```
covid
```

Let's think about how we want our final dataframe to look. We want to have two rows for each state, one with the TB notification and COVID data for the public sector, and one with the TB notification and COVID data for the private sector. That means we have to match on both `state_UT` and `hc_type`. Just as we did for the patient data, we have to specify both key values in the `by=` statement using `c()`. Let's try it out!

```
notif_covid <- notification %>%
  left_join(covid, by=c("state_UT", "hc_type"))
notif_covid
```

Great, that's exactly what we wanted!

Create a new dataframe called `all_tb_data` together the `notif_covid` dataset with the `child` dataset. Then, join this dataframe with the `regions` data for a final combined dataframe, ensuring that no TB data is lost.

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

(make sure to update the contributor list accordingly!)