FURTHER DATA ANALYSIS WITH R

USING REAL-WORLD DATA FROM HIV, TB & MALARIA

The GRAPH Courses, Global Fund & WHO

This book is a compilation of training materials created by the GRAPH Network under a grant from the Global Fund to Fight AIDS, Tuberculosis and Malaria. The materials aim to build global capacity in epidemiological data analysis and decision-making for public health.

Working with Strings in R

Introduction

Proficiency in string manipulation is a vital skill for data scientists. Tasks like cleaning messy data and formatting outputs rely heavily on the ability to parse, combine, and modify character strings. This lesson focuses on techniques for working with strings in R, utilizing functions from the {stringr} package in the tidyverse. Let's dive in!

Learning Objectives

- Understand the concept of strings and rules for defining them in R
- Use escapes to include special characters like quotes within strings
- Employ {stringr} functions to format strings:
  - Change case with `str_to_lower()`, `str_to_upper()`, `str_to_title()`
  - Trim whitespace with `str_trim()` and `str_squish()`
  - Pad strings to equal width with `str_pad()`
  - Wrap text to a certain width using `str_wrap()`
- Split strings into parts using `str_split()` and `separate()`
- Combine strings together with `paste()` and `paste0()`
- Extract substrings from strings using `str_sub()`

Packages

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

Defining Strings

There are fundamental rules for defining character strings in R.

Strings can be enclosed in either single or double quotes. However, the type of quotation mark used at the start must match the one used at the end. For example:

```
string_1 <- "Hello" # Using double quotes
string_2 <- 'Hello' # Using single quotes
```

You cannot normally include double quotes inside a string that starts and ends with double quotes. The same applies to single quotes inside a string that starts and ends with single quotes. For example:

```
will_not_work <- "Double quotes " inside double quotes"
will_not_work <- 'Single quotes ' inside double quotes'
```

But you can include single quotes inside a string that starts and ends with double quotes, and vice versa:

```
single_inside_double <- "Single quotes ' inside double quotes"
```

Alternatively, you can use the escape character \ to include a literal single or double quote inside a string:

```
single_quote <- 'Single quotes \' inside double quotes'
double_quote <- "Double quotes \" inside double quotes"
```

To display these strings as they would appear in output, such as on a plot, use `cat()`:

```
cat('Single quotes \' inside double quotes')
```

```
## Single quotes ' inside double quotes
```

```
cat("Double quotes \" inside double quotes")
```

```
## Double quotes " inside double quotes
```

`cat()` prints its arguments without additional formatting.

Since \ is the escape character, you must use \\ to include a literal backslash in a string:

```
backslash <- "This is a backslash: \\"
cat(backslash)
```

```
## This is a backslash: \
```

Q: Error Spotting in String Definitions

Below are attempts to define character strings in R, with two out of five lines containing an error. Identify and correct these errors.

```
ex_a <- 'She said, "Hello!" to him.'
ex_b <- "She said \"Let's go to the moon\""
ex_c <- "They've been "best friends" for years."
ex_d <- 'Jane\\'s diary'
ex_e <- "It's a sunny day!
```

String Formatting in R with {stringr}

The {stringr} package in R provides useful functions for formatting strings for analysis and visualization. This includes case changes, whitespace handling, length

standardization, and text wrapping.

Changing Case

Converting case is often needed to standardize strings or prepare them for display. The {stringr} package provides several case-changing functions:

- `str_to_upper()` converts strings to uppercase.

```
str_to_upper("hello world")
```

```
## [1] "HELLO WORLD"
```

- `str_to_lower()` converts strings to lowercase.

```
str_to_lower("Goodbye")
```

```
## [1] "goodbye"
```

- `str_to_title()` capitalizes the first letter of each word. Ideal for titling names, subjects, etc.

```
str_to_title("string manipulation")
```

```
## [1] "String Manipulation"
```

Handling Whitespace

Managing whitespace makes strings neat and uniform. The {stringr} package provides two main functions for this:

- `str_trim()` removes whitespace at the start and end.

```
str_trim(" trimmed ")
```

```
## [1] "trimmed"
```

- `str_squish()` removes whitespace at the start and end, *and* reduces multiple internal spaces to one.

```r
str_squish("   too   much    space   ")
```

```
## [1] "too much space"
```

```r
# notice the difference with str_trim
str_trim("   too   much    space   ")
```

```
## [1] "too   much    space"
```

Text Padding

`str_pad()` pads strings to a fixed width. For example, we can pad the number 7 to force it to have 3 characters:

```r
str_pad("7", width = 3, pad = "0") # Pad left to length 3 with 0
```

```
## [1] "007"
```

The first argument is the string to pad. `width` sets the final string width and `pad` specifies the padding character.

`side` controls whether padding is added on the left or right. The `side` argument defaults to "left", so padding will be added on the left side if not specified. Specifying `side = "right"` pads on the right side instead:

```r
str_pad("7", width = 4, side = "right", pad = "_") # Pad right to length 4
with _
```

```
## [1] "7___"
```

Or we can pad on both sides:

```r
str_pad("7", width = 5, side = "both", pad = "_") # Pad both sides to length
5 with _
```

```
## [1] "__7__"
```

Text Wrapping

Text wrapping helps fit strings into confined spaces like plot titles. The `str_wrap()` function wraps text to a set width.

For example, to wrap text at 10 characters we can write:

```
example_string <- "String Manipulation with str_wrap can enhance readability
in plots."
wrapped_to_10 <- str_wrap(example_string, width = 10)
wrapped_to_10
```

```
## [1] "String\nManipulation\nwith\nstr_wrap\ncan\nenhance\nreadability\nin
plots."
```

The output may appear confusing. The `\n` indicates a line break, and to view the modified properly, we need to use the `cat()` function, which is a special version of `print()`:

```
cat(wrapped_to_10)
```

```
## String
## Manipulation
## with
## str_wrap
## can
## enhance
## readability
## in plots.
```

Notice that the function maintains whole words, so it won't split longer words like "manipulation".

Setting the width to 1 essentially splits the string into individual words:

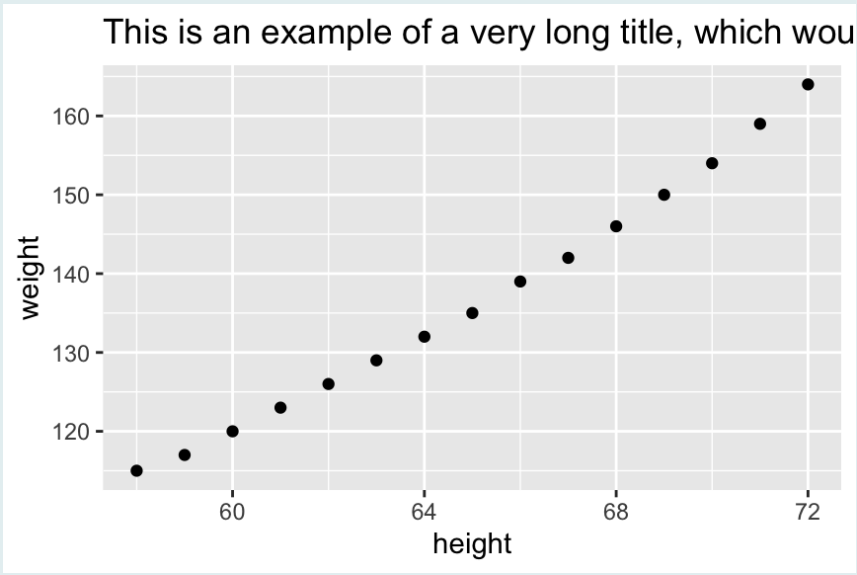
```
cat(str_wrap(example_string, width = 1))
```

```
## String
## Manipulation
## with
## str_wrap
## can
## enhance
## readability
```

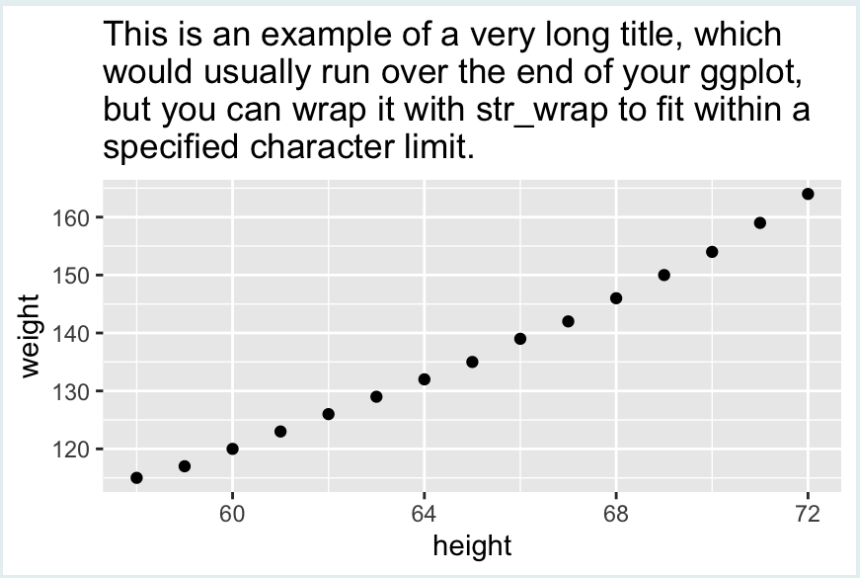
```
## in
## plots.
```

`str_wrap()` is particularly useful in plotting with ggplot2. For example, wrapping a long title to prevent it from spilling over the plot:

```
long_title <- "This is an example of a very long title, which would usually
run over the end of your ggplot, but you can wrap it with str_wrap to fit
within a specified character limit."

# Without wrapping
ggplot(women, aes(height, weight)) +
  geom_point() +
  labs(title = long_title)
```




```
# With wrapping at 80 characters
ggplot(women, aes(height, weight)) +
  geom_point() +
  labs(title = str_wrap(long_title, width = 50))
```



So `str_wrap()` keeps titles neatly within the plot!

Q: Cleaning Patient Name Data

A dataset contains patient names with inconsistent formatting and extra white spaces. Use the {stringr} package to standardize this information:

```
patient_names <- c("  john doe", "ANNA SMITH   ", "Emily Davis")
# 1. Trim white spaces from each name.
# 2. Convert each name to title case for consistency.
```

Q: Standardizing Drug Codes

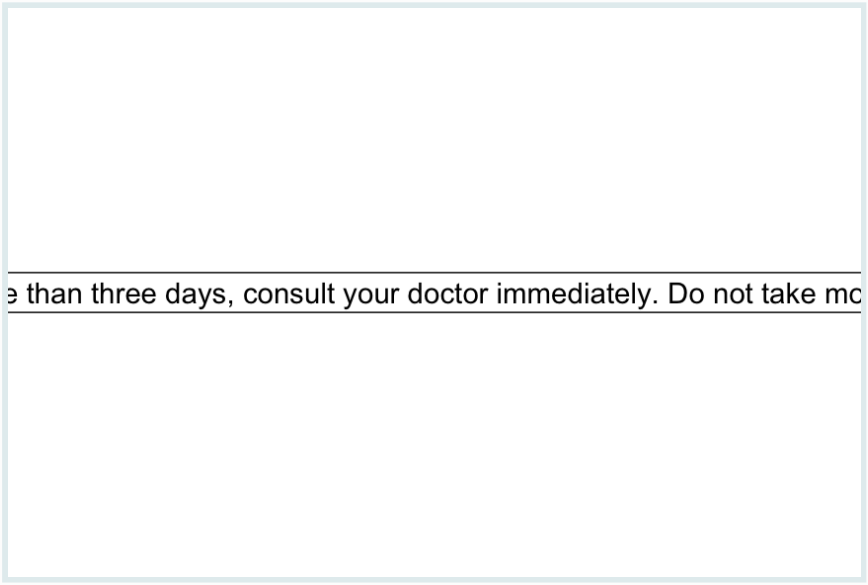
The following (fictional) drug codes are inconsistently formatted. Standardize them by padding with zeros to ensure all codes are 8 characters long:

```
drug_codes <- c("12345", "678", "91011")
# Pad each code with zeros on the left to a fixed width of 8 characters.
```

Q: Wrapping Medical Instructions

Use `str_wrap()` to format the following for better readability:

```
instructions <- "Take two tablets daily after meals. If
symptoms persist for more than three days, consult your doctor
immediately. Do not take more than the recommended dose. Keep
out of reach of children."

ggplot(data.frame(x = 1, y = 1), aes(x, y, label =
instructions)) +
  geom_label() +
  theme_void()
```




```
# Now, wrap the instructions to a width of 50 characters then
plot again.
```

Applying String Formatting to a Dataset

Now let's apply the {stringr} package's string formatting functions to clean and standardize a dataset. Our focus is on a dataset from a study on HIV care and treatment services in Zambézia Province, Mozambique, available here. The original dataset had various formatting inconsistencies, but we've added additional mistakes for educational purposes.

First, we load the dataset and examine specific variables for potential issues.

```r
# Load the dataset
hiv_dat_messy_1 <- openxlsx::read.xlsx(here("data/hiv_dat_messy_1.xlsx"))
%>%
  as_tibble()

# These four variables contain formatting inconsistencies:
hiv_dat_messy_1 %>%
  select(district, health_unit, education, regimen)
```

```
## # A tibble: 1,413 × 4
##    district health_unit                education  regimen
##    <chr>    <chr>                      <chr>      <chr>
##  1 "Rural"  District Hospital Maganj… MISSING    AZT+3TC+NVP
##  2 "Rural"  District Hospital Maganj… secondary  TDF+3TC+EFV
##  3 "Urban"  24th  Of  July  Health  … MISSING    tdf+3tc+efv
##  4 "Urban"  24th  Of  July  Health  … MISSING    TDF+3TC+EFV
##  5 " Urban" 24th  Of  July  Health  … University tdf+3tc+efv
##  6 "Urban"  24th Of July Health Faci… Technical  AZT+3TC+NVP
##  7 "Rural"  District Hospital Maganj… Technical  TDF+3TC+EFV
##  8 "Urban"  24th Of July Health Faci… Technical  azt+3tc+nvp
##  9 "Urban"  24th Of July Health Faci… Technical  AZT+3TC+NVP
## 10 "Urban"  24th Of July Health Faci… Technical  TDF+3TC+EFV
## # ℹ 1,403 more rows
```

Using the `tabyl` function, we can identify and count unique values, revealing the inconsistencies:

```r
# Counting unique values
hiv_dat_messy_1 %>% tabyl(health_unit)
```

```
##                           health_unit   n    percent
##       24th  Of  July  Health  Facility 239 0.16914367
##           24th Of July Health Facility 249 0.17622081
##  District  Hospital  Maganja  Da  Costa 342 0.24203822
##      District Hospital Maganja Da Costa 336 0.23779193
##                 Nante  Health  Facility 119 0.08421798
##                   Nante Health Facility 128 0.09058740
```

```r
hiv_dat_messy_1 %>% tabyl(education)
```

```
##   education   n     percent
##     MISSING 776 0.549186129
##        None 128 0.090587403
##     Primary 178 0.125973107
##   Secondary  82 0.058032555
##   Technical  17 0.012031139
##  University   4 0.002830856
```

```
##     primary 157 0.111111111
##   secondary  71 0.050247700
```

```
hiv_dat_messy_1 %>% tabyl(regimen)
```

```
##       regimen   n      percent valid_percent
##   AZT+3TC+EFV  24 0.0169851380  0.0179910045
##   AZT+3TC+NVP 229 0.1620665251  0.1716641679
##   D4T+3TC+ABC   1 0.0007077141  0.0007496252
##   D4T+3TC+EFV   2 0.0014154282  0.0014992504
##   D4T+3TC+NVP  16 0.0113234253  0.0119940030
##         OTHER   1 0.0007077141  0.0007496252
##   TDF+3TC+EFV 404 0.2859164897  0.3028485757
##   TDF+3TC+NVP   3 0.0021231423  0.0022488756
##   azt+3tc+efv  16 0.0113234253  0.0119940030
##   azt+3tc+nvp 231 0.1634819533  0.1731634183
##   d4t+3tc+efv   9 0.0063694268  0.0067466267
##   d4t+3tc+nvp  18 0.0127388535  0.0134932534
##   d4t+4tc+nvp   1 0.0007077141  0.0007496252
##  d4t6+3tc+nvp   2 0.0014154282  0.0014992504
##         other   2 0.0014154282  0.0014992504
##   tdf+3tc+efv 374 0.2646850672  0.2803598201
##   tdf+3tc+nvp   1 0.0007077141  0.0007496252
##          <NA>  79 0.0559094126            NA
```

```
hiv_dat_messy_1 %>% tabyl(district)
```

```
##  district   n    percent
##     Rural 234 0.16560510
##     Urban 118 0.08351026
##     Rural 691 0.48903043
##     Urban 370 0.26185421
```

Another useful function for visualizing these issues is `tbl_summary` from the {gtsummary} package:

```
hiv_dat_messy_1 %>%
  select(district, health_unit, education, regimen) %>%
  tbl_summary()
```

Characteristic	N = 1,413[1]
district	
Rural	234 (17%)
Urban	118 (8.4%)
Rural	691 (49%)
Urban	370 (26%)
health_unit	
24th Of July Health Facility	239 (17%)
24th Of July Health Facility	249 (18%)
District Hospital Maganja Da Costa	342 (24%)
District Hospital Maganja Da Costa	336 (24%)
Nante Health Facility	119 (8.4%)
Nante Health Facility	128 (9.1%)
education	
MISSING	776 (55%)
None	128 (9.1%)
primary	157 (11%)
Primary	178 (13%)
secondary	71 (5.0%)
Secondary	82 (5.8%)
Technical	17 (1.2%)
University	4 (0.3%)
regimen	
azt+3tc+efv	16 (1.2%)
AZT+3TC+EFV	24 (1.8%)

Characteristic	N = 1,413[1]
AZT+3TC+NVP	229 (17%)
D4T+3TC+ABC	1 (<0.1%)
d4t+3tc+efv	9 (0.7%)
D4T+3TC+EFV	2 (0.1%)
d4t+3tc+nvp	18 (1.3%)
D4T+3TC+NVP	16 (1.2%)
d4t+4tc+nvp	1 (<0.1%)
d4t6+3tc+nvp	2 (0.1%)
other	2 (0.1%)
OTHER	1 (<0.1%)
tdf+3tc+efv	374 (28%)
TDF+3TC+EFV	404 (30%)
tdf+3tc+nvp	1 (<0.1%)
TDF+3TC+NVP	3 (0.2%)
Unknown	79
[1] n (%)	

The output clearly shows inconsistencies in casing, spacing, and format, so we need to standardize them.

Next, we address these issues systematically:

```
hiv_dat_clean_1 <- hiv_dat_messy_1 %>%
  mutate(
    district = str_to_title(str_trim(district)), # Standardizing district
names
    health_unit = str_squish(health_unit),       # Removing extra spaces
    education = str_to_title(education),         # Standardizing education
levels
    regimen = str_to_upper(regimen)              # Consistency in regimen
column
  )
```

And we can verify the effectiveness of these changes by rerunning the `tbl_summary()` function:

```
hiv_dat_clean_1 %>%
  select(district, health_unit, education, regimen) %>%
  tbl_summary()
```

Characteristic	N = 1,413[1]
district	
Rural	925 (65%)
Urban	488 (35%)
health_unit	
24th Of July Health Facility	488 (35%)
District Hospital Maganja Da Costa	678 (48%)
Nante Health Facility	247 (17%)
education	
Missing	776 (55%)
None	128 (9.1%)
Primary	335 (24%)
Secondary	153 (11%)
Technical	17 (1.2%)
University	4 (0.3%)
regimen	
AZT+3TC+EFV	40 (3.0%)
AZT+3TC+NVP	460 (34%)
D4T+3TC+ABC	1 (<0.1%)
D4T+3TC+EFV	11 (0.8%)
D4T+3TC+NVP	34 (2.5%)
D4T+4TC+NVP	1 (<0.1%)
D4T6+3TC+NVP	2 (0.1%)
OTHER	3 (0.2%)
TDF+3TC+EFV	778 (58%)

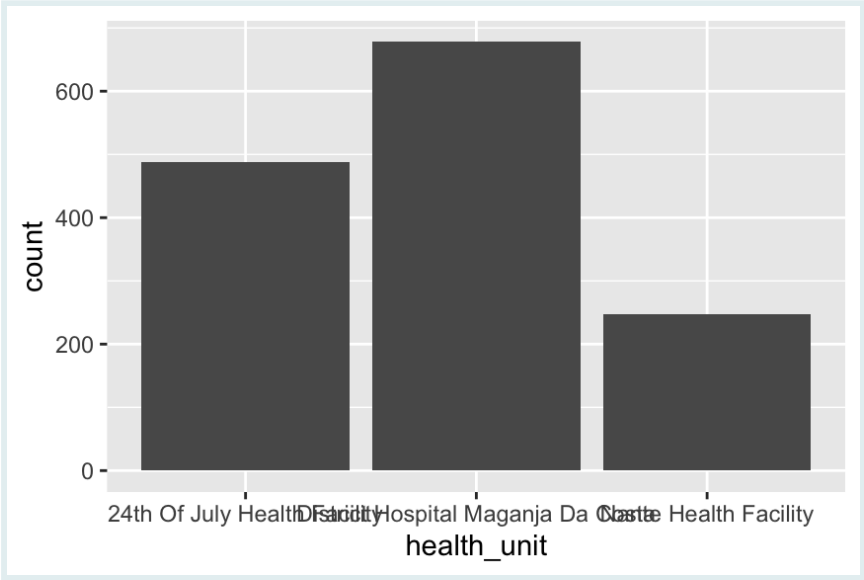
Characteristic	N = 1,413[1]
Unknown	79

[1] n (%)

Great!

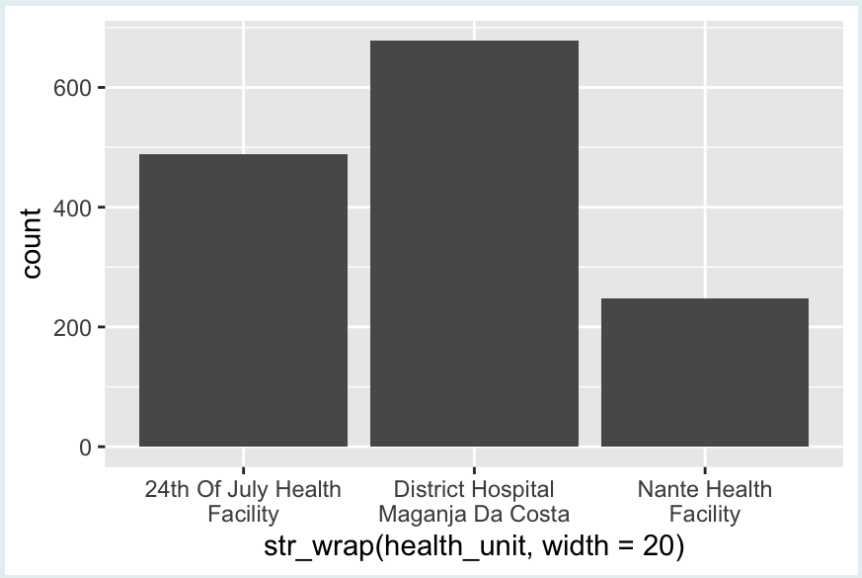
Finally, let's attempt to plot counts of the `health_unit` variable. For the plot style below, we encounter an issue with lengthy labels:

```
ggplot(hiv_dat_clean_1, aes(x = health_unit)) +
  geom_bar()
```



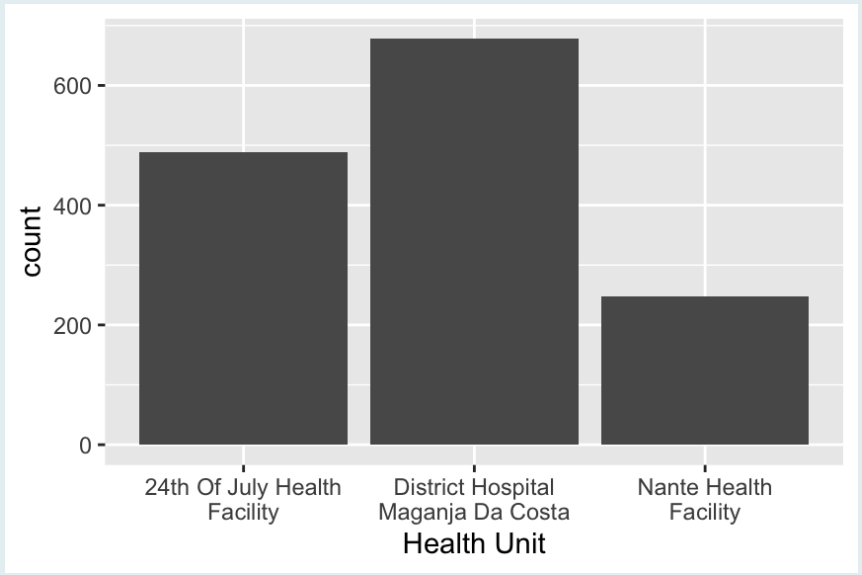
To resolve this, we can adjust the labels using `str_wrap()`:

```
hiv_dat_clean_1 %>%
  ggplot(aes(x = str_wrap(health_unit, width = 20))) +
  geom_bar()
```



Much cleaner, though we should probably fix the axis title:

```
hiv_dat_clean_1 %>%
  ggplot(aes(x = str_wrap(health_unit, width = 20))) +
  geom_bar() +
  labs(x = "Health Unit")
```



Now try your hand on similar cleaning operations in the practice questions below.

Q: Formatting a Tuberculosis Dataset

In this exercise, you will clean a dataset, `lima_messy`, originating from a tuberculosis (TB) treatment adherence study in Lima, Peru. More details about the study and the dataset are available here.

Begin by importing the dataset:

```
lima_messy_1 <-
openxlsx::read.xlsx(here("data/lima_messy_1.xlsx")) %>%
  as_tibble()
lima_messy_1
```

```
## # A tibble: 1,293 × 18
##    id      age          sex   marital_status
##    <chr>   <chr>        <chr> <chr>
##  1 pe-1008 38 and older M     Single
##  2 lm-1009 38 and older M     Married   /   cohabitating
##  3 pe-1010 27 to 37     m     Married   /  cohabitating
##  4 lm-1011 27 to 37     m     Married   /  cohabitating
##  5 pe-1012 38 and older m     Married  /  cohabitating
##  6 lm-1013 27 to 37     M     Single
##  7 pe-1014 27 To 37     m     Married   / cohabitating
##  8 lm-1015 22 To 26     m     Single
##  9 pe-1016 27 to 37     m     Single
## 10 lm-1017 22 to 26     m     Single
## # ℹ 1,283 more rows
## # ℹ 14 more variables: poverty_level <chr>, …
```

Your task is to clean the `marital_status`, `sex`, and `age` variables in `lima_messy`. Following the cleaning process, generate a summary table using the `tbl_summary()` function. Aim for your output to align with this structure:

Characteristic	N = 1,293
marital_status	
Divorced / Separated	93 (7.2%)
Married / Cohabitating	486 (38%)
Single	677 (52%)
Widowed	37 (2.9%)
sex	
F	503 (39%)
M	790 (61%)

Characteristic	N = 1,293
21 and younger	338 (26%)
22 to 26	345 (27%)
27 to 37	303 (23%)
38 and older	307 (24%)

Implement the cleaning and summarize:


```
# Create a new object for cleaned data
lima_clean <- lima_messy %>%
  mutate(
    # Clean marital_status

    # Clean sex

    # Clean age

  )

# Check cleaning
lima_clean %>%
  select(marital_status, sex, age) %>%
  tbl_summary()
```

Q: Wrapping Axis Labels in a Plot

Using the cleaned dataset `lima_clean` from the previous task, create a bar plot to display the count of participants by `marital_status`. Then wrap the axis labels on the x-axis to a maximum of 15 characters per line for readability.

```
# Create your bar plot with wrapped text here:
```

Splitting Strings with `str_split()` and `separate()`

Splitting strings is common task in data manipulation. The tidyverse offers efficient functions for this task, notably `stringr::str_split()` and `tidyr::separate()`.

Using `str_split()`

The `str_split()` function is useful for dividing strings into parts. For example:

```
example_string <- "split-this-string"
str_split(example_string, pattern = "-")
```

```
## [[1]]
## [1] "split"  "this"   "string"
```

This code splits `example_string` at each hyphen.

However, applying `str_split()` directly to a dataframe can be more complex.

Let's try it with the IRS dataset from Malawi as a case study. You should already be familiar with this dataset from a previous lesson. It is available here. For now, we'll focus on the `start_date_long` column:

```
irs <- read_csv(here("data/Illovo_data.csv"))
irs_dates_1 <- irs %>% select(village, start_date_long)
irs_dates_1
```

```
## # A tibble: 112 × 2
##    village           start_date_long
##    <chr>             <chr>
##  1 Mess              April 07 2014
##  2 Nkombedzi         April 22 2014
##  3 B Compound        May 13 2014
##  4 D Compound        May 13 2014
##  5 Post Office       May 13 2014
##  6 Mangulenje        May 15 2014
##  7 Mangulenje Senior May 27 2014
##  8 Old School        May 27 2014
##  9 Mwanza            May 28 2014
## 10 Alumenda          June 18 2014
## # ℹ 102 more rows
```

Suppose we want to split the `start_date_long` variable to extract the day, month, and year. We can write:

```
irs_dates_1 %>%
  mutate(start_date_parts = str_split(start_date_long, " "))
```

```
## # A tibble: 112 × 3
##    village           start_date_long start_date_parts
```

```
##    <chr>             <chr>           <list>
##  1 Mess              April 07 2014   <chr [3]>
##  2 Nkombedzi         April 22 2014   <chr [3]>
##  3 B Compound        May 13 2014     <chr [3]>
##  4 D Compound        May 13 2014     <chr [3]>
##  5 Post Office       May 13 2014     <chr [3]>
##  6 Mangulenje        May 15 2014     <chr [3]>
##  7 Mangulenje Senior May 27 2014     <chr [3]>
##  8 Old School        May 27 2014     <chr [3]>
##  9 Mwanza            May 28 2014     <chr [3]>
## 10 Alumenda          June 18 2014    <chr [3]>
## # ℹ 102 more rows
```

This results in a list column, which can be difficult to work with. To make it more readable, we can use `unnest_wider()`:

```
irs_dates_1 %>%
  mutate(start_date_parts = str_split(start_date_long, " ")) %>%
  unnest_wider(start_date_parts, names_sep = "_")
```

```
## # A tibble: 112 × 5
##    village           start_date_long start_date_parts_1
##    <chr>             <chr>           <chr>
##  1 Mess              April 07 2014   April
##  2 Nkombedzi         April 22 2014   April
##  3 B Compound        May 13 2014     May
##  4 D Compound        May 13 2014     May
##  5 Post Office       May 13 2014     May
##  6 Mangulenje        May 15 2014     May
##  7 Mangulenje Senior May 27 2014     May
##  8 Old School        May 27 2014     May
##  9 Mwanza            May 28 2014     May
## 10 Alumenda          June 18 2014    June
## # ℹ 102 more rows
## # ℹ 2 more variables: start_date_parts_2 <chr>, …
```

It works! Our date parts are now split. However, this approach is quite cumbersome. A better solution for splitting components is the `separate()` function.

Using `separate()`

Let's try the same task using `separate()`:

```
irs_dates_1 %>%
  separate(start_date_long, into = c("month", "day", "year"), sep = " ")
```

```
## # A tibble: 112 × 4
##    village           month day   year
##    <chr>             <chr> <chr> <chr>
```

```
##  1 Mess              April 07    2014
##  2 Nkombedzi         April 22    2014
##  3 B Compound        May   13    2014
##  4 D Compound        May   13    2014
##  5 Post Office       May   13    2014
##  6 Mangulenje        May   15    2014
##  7 Mangulenje Senior May   27    2014
##  8 Old School        May   27    2014
##  9 Mwanza            May   28    2014
## 10 Alumenda          June  18    2014
## # ℹ 102 more rows
```

Much more straightforward!

This function requires specifying:

- The column to be split.
- `into` - Names of the new columns.
- `sep` - The separator character.

To retain the original column, use `remove = FALSE`:

```
irs_dates_1 %>%
  separate(start_date_long, into = c("month", "day", "year"), sep = " ",
remove = FALSE)
```

```
## # A tibble: 112 × 5
##    village           start_date_long month day   year
##    <chr>             <chr>           <chr> <chr> <chr>
##  1 Mess              April 07 2014   April 07    2014
##  2 Nkombedzi         April 22 2014   April 22    2014
##  3 B Compound        May 13 2014     May   13    2014
##  4 D Compound        May 13 2014     May   13    2014
##  5 Post Office       May 13 2014     May   13    2014
##  6 Mangulenje        May 15 2014     May   15    2014
##  7 Mangulenje Senior May 27 2014     May   27    2014
##  8 Old School        May 27 2014     May   27    2014
##  9 Mwanza            May 28 2014     May   28    2014
## 10 Alumenda          June 18 2014    June  18    2014
## # ℹ 102 more rows
```

Alternatively, the `lubridate` package offers functions to extract date components:

```
irs_dates_1 %>%
  mutate(start_date_long = mdy(start_date_long)) %>%
  mutate(day = day(start_date_long),
```

```
          month = month(start_date_long, label = TRUE),
year = year(start_date_long))
```

SIDE NOTE

```
## # A tibble: 112 × 5
##    village           start_date_long   day month  year
##    <chr>             <date>          <int> <ord> <dbl>
##  1 Mess              2014-04-07          7 Apr    2014
##  2 Nkombedzi         2014-04-22         22 Apr    2014
##  3 B Compound        2014-05-13         13 May    2014
##  4 D Compound        2014-05-13         13 May    2014
##  5 Post Office       2014-05-13         13 May    2014
##  6 Mangulenje        2014-05-15         15 May    2014
##  7 Mangulenje Senior 2014-05-27         27 May    2014
##  8 Old School        2014-05-27         27 May    2014
##  9 Mwanza            2014-05-28         28 May    2014
## 10 Alumenda          2014-06-18         18 Jun    2014
## # ℹ 102 more rows
```

When some rows lack all the necessary parts, `separate()` will issue a warning. Let's demonstrate this by artificially removing all instances of the word "April" from our dates:

```
irs_dates_with_problem <-
  irs_dates_1 %>%
  mutate(start_date_missing = str_replace(start_date_long, "April ", ""))
irs_dates_with_problem
```

```
## # A tibble: 112 × 3
##    village           start_date_long start_date_missing
##    <chr>             <chr>           <chr>
##  1 Mess              April 07 2014   07 2014
##  2 Nkombedzi         April 22 2014   22 2014
##  3 B Compound        May 13 2014     May 13 2014
##  4 D Compound        May 13 2014     May 13 2014
##  5 Post Office       May 13 2014     May 13 2014
##  6 Mangulenje        May 15 2014     May 15 2014
##  7 Mangulenje Senior May 27 2014     May 27 2014
##  8 Old School        May 27 2014     May 27 2014
##  9 Mwanza            May 28 2014     May 28 2014
## 10 Alumenda          June 18 2014    June 18 2014
## # ℹ 102 more rows
```

Now, let's try to split the date parts:

```
irs_dates_with_problem %>%
  separate(start_date_missing, into = c("month", "day", "year"), sep = " ")
```

```
## # A tibble: 112 × 5
##    village           start_date_long month day   year
##    <chr>             <chr>           <chr> <chr> <chr>
##  1 Mess              April 07 2014   07    2014  <NA>
##  2 Nkombedzi         April 22 2014   22    2014  <NA>
##  3 B Compound        May 13 2014     May   13    2014
##  4 D Compound        May 13 2014     May   13    2014
##  5 Post Office       May 13 2014     May   13    2014
##  6 Mangulenje        May 15 2014     May   15    2014
##  7 Mangulenje Senior May 27 2014     May   27    2014
##  8 Old School        May 27 2014     May   27    2014
##  9 Mwanza            May 28 2014     May   28    2014
## 10 Alumenda          June 18 2014    June  18    2014
## # ℹ 102 more rows
```

As you can see, rows missing parts will produce warnings. Handle such warnings carefully, as they can lead to inaccurate data. In this case, we now have the day and month information for those rows in the wrong columns.

Q: Splitting Age Range Strings

Consider the esoph_ca dataset, from the {medicaldata} package, which involves a case-control study of esophageal cancer in France.

```
medicaldata::esoph_ca %>% as_tibble()
```


```
## # A tibble: 88 × 5
##    agegp alcgp     tobgp    ncases ncontrols
##    <ord> <ord>     <ord>     <dbl>     <dbl>
##  1 25-34 0-39g/day 0-9g/day      0        40
##  2 25-34 0-39g/day 10-19         0        10
##  3 25-34 0-39g/day 20-29         0         6
##  4 25-34 0-39g/day 30+           0         5
##  5 25-34 40-79     0-9g/day      0        27
##  6 25-34 40-79     10-19         0         7
##  7 25-34 40-79     20-29         0         4
##  8 25-34 40-79     30+           0         7
##  9 25-34 80-119    0-9g/day      0         2
## 10 25-34 80-119    10-19         0         1
## # ℹ 78 more rows
```

Split the age ranges in the `agegp` column into two separate columns: `agegp_lower` and `agegp_upper`.

After using the `separate()` function, the "75+" age group will require special handling. Use `readr::parse_number()` or another method to convert the lower age limit ("75+") to a number.

```
medicaldata::esoph_ca %>%
  separate(________) %>%
  # convert 75+ to a number
  mutate(________)
```

Separating Special Characters

To use the `separate()` function on special characters like the period (.), we need to escape them with a double backslash (`\\`).

Consider the scenario where dates are formatted with periods:

```
irs_with_period <- irs_dates_1 %>%
  mutate(start_date_long = format(lubridate::mdy(start_date_long),
"%d.%m.%Y"))
irs_with_period
```

```
## # A tibble: 112 × 2
##    village           start_date_long
##    <chr>             <chr>
##  1 Mess              07.04.2014
##  2 Nkombedzi         22.04.2014
##  3 B Compound        13.05.2014
##  4 D Compound        13.05.2014
##  5 Post Office       13.05.2014
##  6 Mangulenje        15.05.2014
##  7 Mangulenje Senior 27.05.2014
##  8 Old School        27.05.2014
##  9 Mwanza            28.05.2014
## 10 Alumenda          18.06.2014
## # ℹ 102 more rows
```

Attempting to separate this date format directly with `sep = "."` will not work:

```
irs_with_period %>%
  separate(start_date_long, into = c("day", "month", "year"), sep = ".")
```

```
## # A tibble: 112 × 4
##    village           day   month year
##    <chr>             <chr> <chr> <chr>
##  1 Mess              ""    ""    ""
##  2 Nkombedzi         ""    ""    ""
##  3 B Compound        ""    ""    ""
##  4 D Compound        ""    ""    ""
##  5 Post Office       ""    ""    ""
##  6 Mangulenje        ""    ""    ""
##  7 Mangulenje Senior ""    ""    ""
##  8 Old School        ""    ""    ""
##  9 Mwanza            ""    ""    ""
## 10 Alumenda          ""    ""    ""
## # ℹ 102 more rows
```

This doesn't work as intended because, in regex (regular expressions), the period is a special character. We'll learn more about these in due course. The correct approach is to escape the period uses a double backslash (\):

```
irs_with_period %>%
  separate(start_date_long, into = c("day", "month", "year"), sep = "\\.")
```

```
## # A tibble: 112 × 4
##    village           day   month year
##    <chr>             <chr> <chr> <chr>
##  1 Mess              07    04    2014
##  2 Nkombedzi         22    04    2014
##  3 B Compound        13    05    2014
##  4 D Compound        13    05    2014
##  5 Post Office       13    05    2014
##  6 Mangulenje        15    05    2014
##  7 Mangulenje Senior 27    05    2014
##  8 Old School        27    05    2014
##  9 Mwanza            28    05    2014
## 10 Alumenda          18    06    2014
## # ℹ 102 more rows
```

Now, the function understands to split the string at each literal period.

Similarly, when using other special characters like +, ∗, or ?, we also need to precede them with a double backslash (\) in the `sep` argument.

What is a Special Character?

In regular expressions, which help find patterns in text, special characters have specific roles. For example, a period (.) is a wildcard that can

SIDE NOTE

represent any character. So, in a search, "do.t" could match "dolt," "dost," or "doct" Similarly, the plus sign (+) is used to indicate one or more occurrences of the preceding character. For example, "ho+se" would match "hose" or "hooose" but not "hse." When we need to use these characters in their ordinary roles, we use a double backslash (\\) before them, like "`\\.`" or "`\\+.`" More on these special characters will be covered in a future lesson.

Q: Separating Special Characters

Your next task involves the `hiv_dat_clean_1` dataset. Focus on the `regimen` column, which lists drug regimens separated by a + sign. Your goal is to split this column into three new columns: `drug_1`, `drug_2`, and `drug_3` using the `separate()` function. Pay close attention to how you handle the + separator. Here's the column:

PRACTICE (in RMD)

```
hiv_dat_clean_1 %>%
  select(regimen)
```

```
## # A tibble: 1,413 × 1
##    regimen
##    <chr>
##  1 AZT+3TC+NVP
##  2 TDF+3TC+EFV
##  3 TDF+3TC+EFV
##  4 TDF+3TC+EFV
##  5 TDF+3TC+EFV
##  6 AZT+3TC+NVP
##  7 TDF+3TC+EFV
##  8 AZT+3TC+NVP
##  9 AZT+3TC+NVP
## 10 TDF+3TC+EFV
## # ℹ 1,403 more rows
```

Combining Strings with `paste()`

The `paste()` function in R concatenates or joins together character strings. This allows you to combine multiple strings into a single string.

To combine two simple strings:

```
string1 <- "Hello"
string2 <- "World"
paste(string1, string2)
```

```
## [1] "Hello World"
```

The default separator is a space, so this returns "Hello World".

Let's demonstrate how to use this on a dataset, with the IRS date data. First, we'll separate the start date into individual columns:

```
irs_dates_separated <- # store for later use
  irs_dates_1 %>%
  separate(start_date_long, into = c("month", "day", "year"), sep = " ",
remove = FALSE)
irs_dates_separated
```

```
## # A tibble: 112 × 5
##    village           start_date_long month day   year
##    <chr>             <chr>           <chr> <chr> <chr>
##  1 Mess              April 07 2014   April 07    2014
##  2 Nkombedzi         April 22 2014   April 22    2014
##  3 B Compound        May 13 2014     May   13    2014
##  4 D Compound        May 13 2014     May   13    2014
##  5 Post Office       May 13 2014     May   13    2014
##  6 Mangulenje        May 15 2014     May   15    2014
##  7 Mangulenje Senior May 27 2014     May   27    2014
##  8 Old School        May 27 2014     May   27    2014
##  9 Mwanza            May 28 2014     May   28    2014
## 10 Alumenda          June 18 2014    June  18    2014
## # ℹ 102 more rows
```

Then we can recombine day, month and year with `paste()`:

```
irs_dates_separated %>%
  select(day, month, year) %>%
  mutate(start_date_long_2 = paste(day, month, year))
```

```
## # A tibble: 112 × 4
##    day   month year  start_date_long_2
##    <chr> <chr> <chr> <chr>
##  1 07    April 2014  07 April 2014
##  2 22    April 2014  22 April 2014
##  3 13    May   2014  13 May 2014
```

```
##  4 13    May   2014  13 May 2014
##  5 13    May   2014  13 May 2014
##  6 15    May   2014  15 May 2014
##  7 27    May   2014  27 May 2014
##  8 27    May   2014  27 May 2014
##  9 28    May   2014  28 May 2014
## 10 18    June  2014  18 June 2014
## # ℹ 102 more rows
```

The `sep` argument specifies the separator between elements. For a different separator, like a hyphen, we can write:

```
irs_dates_separated %>%
  mutate(start_date_long_2 = paste(day, month, year, sep = "-"))
```

```
## # A tibble: 112 × 6
##    village           start_date_long month day   year
##    <chr>             <chr>           <chr> <chr> <chr>
##  1 Mess              April 07 2014   April 07    2014
##  2 Nkombedzi         April 22 2014   April 22    2014
##  3 B Compound        May 13 2014     May   13    2014
##  4 D Compound        May 13 2014     May   13    2014
##  5 Post Office       May 13 2014     May   13    2014
##  6 Mangulenje        May 15 2014     May   15    2014
##  7 Mangulenje Senior May 27 2014     May   27    2014
##  8 Old School        May 27 2014     May   27    2014
##  9 Mwanza            May 28 2014     May   28    2014
## 10 Alumenda          June 18 2014    June  18    2014
## # ℹ 102 more rows
## # ℹ 1 more variable: start_date_long_2 <chr>
```

To concatenate without spaces, we can set `sep = ""`:

```
irs_dates_separated %>%
  select(day, month, year) %>%
  mutate(start_date_long_2 = paste(day, month, year, sep = ""))
```

```
## # A tibble: 112 × 4
##    day   month year  start_date_long_2
##    <chr> <chr> <chr> <chr>
##  1 07    April 2014  07April2014
##  2 22    April 2014  22April2014
##  3 13    May   2014  13May2014
##  4 13    May   2014  13May2014
##  5 13    May   2014  13May2014
##  6 15    May   2014  15May2014
##  7 27    May   2014  27May2014
##  8 27    May   2014  27May2014
##  9 28    May   2014  28May2014
```

```
## 10 18    June  2014  18June2014
## # ℹ 102 more rows
```

Or we can use the `paste0()` function, which is equivalent to `paste(..., sep = "")`:

```
irs_dates_separated %>%
  select(day, month, year) %>%
  mutate(start_date_long_2 = paste0(day, month, year))
```

```
## # A tibble: 112 × 4
##    day   month year  start_date_long_2
##    <chr> <chr> <chr> <chr>
##  1 07    April 2014  07April2014
##  2 22    April 2014  22April2014
##  3 13    May   2014  13May2014
##  4 13    May   2014  13May2014
##  5 13    May   2014  13May2014
##  6 15    May   2014  15May2014
##  7 27    May   2014  27May2014
##  8 27    May   2014  27May2014
##  9 28    May   2014  28May2014
## 10 18    June  2014  18June2014
## # ℹ 102 more rows
```

Let's try to combine `paste()` with some other string functions to solve a realistic data problem. Consider the ID column in the `hiv_dat_messy_1` dataset:

```
hiv_dat_messy_1 %>%
  select(patient_id)
```

```
## # A tibble: 1,413 × 1
##    patient_id
##    <chr>
##  1 pd-10037
##  2 pd-10537
##  3 pd-5489
##  4 id-5523
##  5 pd-4942
##  6 pd-4742
##  7 pd-10879
##  8 id-2885
##  9 pd-4861
## 10 pd-5180
## # ℹ 1,403 more rows
```

Imagine we wanted to standardize these IDs to have the same number of characters. This is often a requirement for IDs (think about phone numbers, for instance).

To implement this, we can use `separate()` to split the IDs into parts, then use `paste()` to recombine them into a standardized format.

```
hiv_dat_messy_1 %>%
  select(patient_id) %>% # for visibility
  separate(patient_id, into = c("prefix", "patient_num"), sep = "-", remove
= F) %>%
  mutate(patient_num = str_pad(patient_num, width = 5, side = "left", pad =
"0")) %>%
  mutate(patient_id_padded = paste(prefix, patient_num, sep = "-"))
```

```
## # A tibble: 1,413 × 4
##    patient_id prefix patient_num patient_id_padded
##    <chr>      <chr>  <chr>       <chr>
##  1 pd-10037   pd     10037       pd-10037
##  2 pd-10537   pd     10537       pd-10537
##  3 pd-5489    pd     05489       pd-05489
##  4 id-5523    id     05523       id-05523
##  5 pd-4942    pd     04942       pd-04942
##  6 pd-4742    pd     04742       pd-04742
##  7 pd-10879   pd     10879       pd-10879
##  8 id-2885    id     02885       id-02885
##  9 pd-4861    pd     04861       pd-04861
## 10 pd-5180    pd     05180       pd-05180
## # ℹ 1,403 more rows
```

In this example, `patient_id` is split into a prefix and a number. The number is then padded with zeros to ensure consistent length, and finally, the two parts are concatenated back together using `paste()` with a hyphen as the separator. This process standardizes the format of patient IDs.

Great work!

Q: Standardizing IDs in the `lima_messy_1` Dataset

In the `lima_messy_1` dataset, the IDs are not zero-padded, making them hard to sort.

For example, the ID pe-998 is at the top of the list after sorting in descending order, which is not what we want.

```
lima_messy_1 %>%
  select(id) %>%
```

```
  arrange(desc(id)) # sort in descending order (highest IDs
should be at the top)
```

```
## # A tibble: 1,293 × 1
##    id
##    <chr>
##  1 pe-998
##  2 pe-996
##  3 pe-951
##  4 pe-900
##  5 pe-2347
##  6 pe-2337
##  7 pe-2335
##  8 pe-2333
##  9 pe-2331
## 10 pe-2329
## # ℹ 1,283 more rows
```

Try to fix this using a similar procedure to the one used for `hiv_dat_messy_1`.

Your Task:

- Separate the ID into parts.
- Pad the numeric part for standardization.
- Recombine the parts using `paste()`.
- Resort the IDs in descending order. The highest ID should end in `2347`

```
lima_messy_1 %>%
  _____________
```

Q: Creating Summary Statements

Create a column containing summary statements combining `village`, `start_date_default`, and `coverage_p` from the `irs` dataset. The statement should describe the spray coverage for each village.

Desired Output: "For village X, the spray coverage was Y% on Z date."

PRACTICE

(in RMD)

Your Task: - Select the necessary columns from the `irs` dataset. - Use `paste()` to create the summary statement.

```
irs %>%
  select(village, start_date_default, coverage_p) %>%
  ____________________
```

REMINDER

As we go through this lesson, remember that RStudio's auto-complete can help you find functions in the stringr package.

Just type `str_` and a list of stringr functions will pop up. All stringr functions start with `str_`.

So instead of trying to memorize them all, you can use auto-complete as a reference when needed.

Subsetting strings with `str_sub`

`str_sub` allows you to extract parts of a string based on character positions. The basic syntax is `str_sub(string, start, end)`.

Example: Extracting the first 2 characters from patient IDs:

```
patient_ids <- c("ID12345-abc", "ID67890-def")
str_sub(patient_ids, 1, 2) # Returns "ID", "ID"
```

```
## [1] "ID" "ID"
```

Or the first 5:

```
str_sub(patient_ids, 1, 5) # Returns "ID123", "ID678"
```

```
## [1] "ID123" "ID678"
```

For example, to get the last 4 characters of patient IDs.

```
str_sub(patient_ids, -4, -1) # Returns "-abc", "-def"
```

```
## [1] "-abc" "-def"
```

Be sure to pause and understand what happened above.

When indices are outside the string length, `str_sub` handles it gracefully without errors:

```
str_sub(patient_ids, 1, 30) # Safely returns the full string when the range
exceeds the string length
```

```
## [1] "ID12345-abc" "ID67890-def"
```

In a data frame, we can use `str_sub` within `mutate()`. For example, below we extract the year and month from the `start_date_default` column and create a new column called `year_month`:

```
irs %>%
  select(start_date_default) %>%
  mutate(year_month = str_sub(start_date_default, start = 1, end = 7))
```

```
## # A tibble: 112 × 2
##    start_date_default year_month
##    <date>             <chr>
##  1 2014-04-07         2014-04
##  2 2014-04-22         2014-04
##  3 2014-05-13         2014-05
##  4 2014-05-13         2014-05
##  5 2014-05-13         2014-05
##  6 2014-05-15         2014-05
##  7 2014-05-27         2014-05
##  8 2014-05-27         2014-05
##  9 2014-05-28         2014-05
## 10 2014-06-18         2014-06
## # ℹ 102 more rows
```

Q: Extracting ID Substrings

Use `str_sub()` to isolate just the numeric part of the `patient_id` column in the `hiv_dat_messy_1` dataset.

```
hiv_dat_messy_1 %>%
  select(patient_id) %>%
  # your code here:
  _____________
```

Wrap up

Congratulations on reaching the end of this lesson! You've learned about strings in R and various functions to manipulate them effectively.

The table below gives a quick recap of the key functions we covered. Remember, you don't need to memorize all these functions. Knowing they exist and how to look them up (like using Google) is more than enough for practical applications.

Function	Description	Example	Example Output
`str_to_upper()`	Convert characters to uppercase	`str_to_upper("hiv")`	"HIV"
`str_to_lower()`	Convert characters to lowercase	`str_to_lower("HIV")`	"hiv"
`str_to_title()`	Convert first character of each word to uppercase	`str_to_title("hiv awareness")`	"Hiv Awareness"
`str_trim()`	Remove whitespace from start & end	`str_trim(" hiv ")`	"hiv"
`str_squish()`	Remove whitespace from start & end and reduce internal spaces	`str_squish(" hiv cases ")`	"hiv cases"
`str_pad()`	Pad a string to a fixed width	`str_pad("45", width = 5)`	"00045"
`str_wrap()`	Wrap a string to a given width (for formatting output)	`str_wrap("HIV awareness", width = 5)`	"HIV"
`str_split()`	Split elements of a character vector	`str_split("Hello-World", "-")`	c("Hello", "World")
`paste()`	Concatenate vectors after		

converting to character	`paste("Hello", "World")`	"Hello World"	
`str_sub()`	Extract and replace substrings from a character vector	`str_sub("HelloWorld", 1, 4)`	"Hell"
`separate()`	Separate a character column into multiple columns	`separate(tibble(a = "Hello-World"), a, into = c("b", "c"), sep = "-")`	`|b |c |` `|Hello |World |`

Note that while these functions cover common tasks such as string standardization, splitting and joining strings, this introduction only scratches the surface of what's possible with the {stringr} package. If you work with a lot of raw text data, you may want to do further exploring on the stringr website.

Answer Key

Q: Error Spotting in String Definitions

1. **ex_a**: Correct.
2. **ex_b**: Correct.
3. **ex_c**: Error. Corrected version: `ex_c <- "They've been \"best friends\" for years."`
4. **ex_d**: Error. Corrected version: `ex_d <- 'Jane\'s diary'`
5. **ex_e**: Error. Close quote missing. Corrected version: `ex_e <- "It's a sunny day!"`

Q: Cleaning Patient Name Data

```
patient_names <- c("  john doe", "ANNA SMITH   ", "Emily Davis")

patient_names <- str_trim(patient_names) # Trim white spaces
patient_names <- str_to_title(patient_names) # Convert to title case
```

Q: Standardizing Drug Codes

```
drug_codes <- c("12345", "678", "91011")
```

```
# Pad each code with zeros on the left to a fixed width of 8
characters.drug_codes_padded <- str_pad(drug_codes, 8, pad = "0")
```

Q: Wrapping Medical Instructions

```
instructions <- "Take two tablets daily after meals. If symptoms persist for
more than three days, consult your doctor immediately. Do not take more than
the recommended dose. Keep out of reach of children."

# Wrap instructions
wrapped_instructions <- str_wrap(instructions, width = 50)

ggplot(data.frame(x = 1, y = 1), aes(x, y, label = wrapped_instructions)) +
  geom_label() +
  theme_void()
```

Q: Formatting a Tuberculosis Dataset

The steps to clean the `lima_messy` dataset would involve:

```
lima_clean <- lima_messy %>%
  mutate(
    marital_status = str_squish(str_to_title(marital_status)), # Clean and
standardize marital_status
    sex = str_squish(str_to_upper(sex)),                       # Clean and
standardize sex
    age = str_squish(str_to_lower(age))                        # Clean and
standardize age
  )


lima_clean %>%
  select(marital_status, sex, age) %>%
  tbl_summary()
```

Then, use the `tbl_summary()` function to create the summary table.

Q: Wrapping Axis Labels in a Plot

```
# Assuming lima_clean is already created and contains marital_status
ggplot(lima_clean, aes(x = str_wrap(marital_status, width = 15))) +
  geom_bar() +
  labs(x = "Marital Status")
```

Q: Splitting Age Range Strings

```
esoph_ca %>%
  select(agegp) %>% # for illustration
  separate(agegp, into = c("agegp_lower", "agegp_upper"), sep = "–") %>%
  mutate(agegp_lower = readr::parse_number(agegp_lower))
```

Q: Creating Summary Statements

```
irs %>%
  select(village, start_date_default, coverage_p) %>%
  mutate(summary_statement = paste0("For village ", village, ", the spray
coverage was ", coverage_p, "% on ", start_date_default))
```

Q: Extracting ID Substrings

```
hiv_dat_messy_1 %>%
  select(patient_id) %>%
  mutate(numeric_part = str_sub(patient_id, 4))
```

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

Factors in R

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.csv"))
```

```
## # 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
## # ℹ 21 more variables: birth_month <chr>, birth_day <dbl>,
## #   death_year <dbl>, death_month <chr>, death_day <dbl>, …
```

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.

Review of the Main Data Classes in R

RECAP

- `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).

RECAP

- `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.

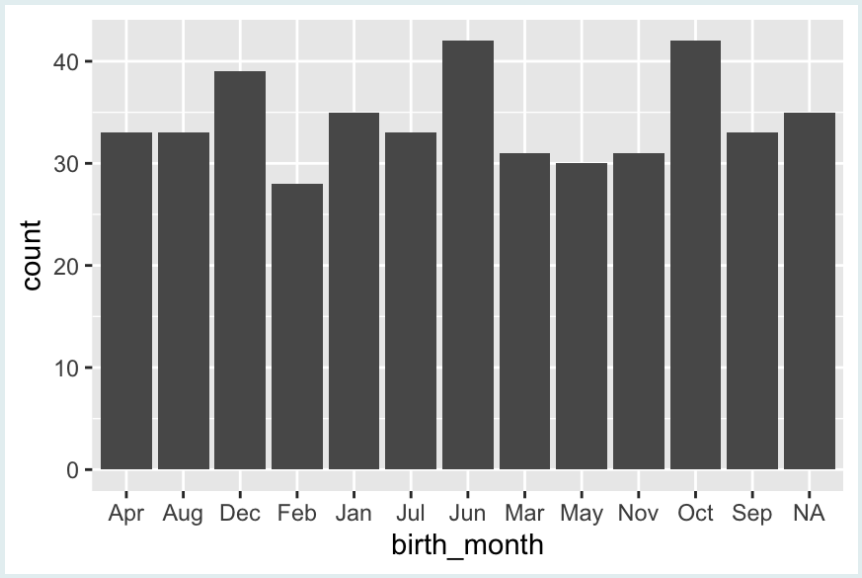
This last point, controlling the order of factor levels, will be our primary focus.

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)
```

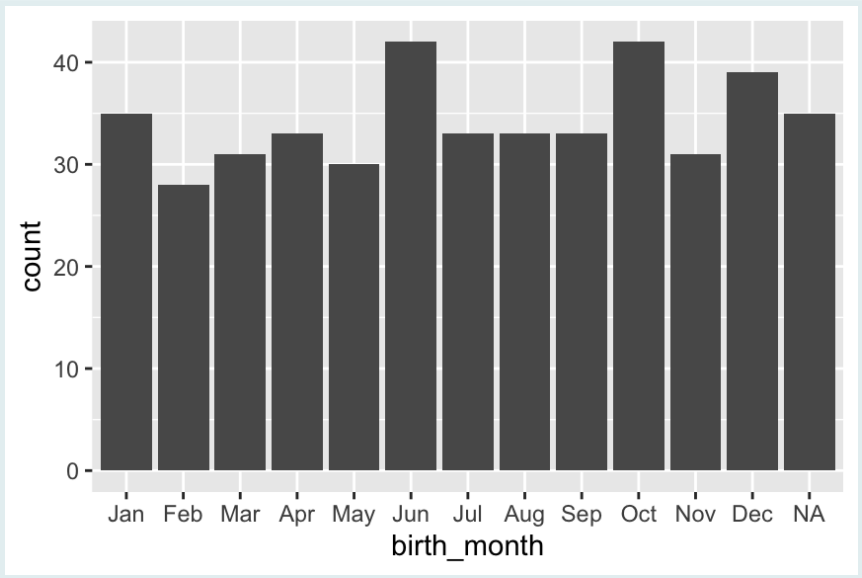
```
## [1] "factor"
```

```
# 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
```

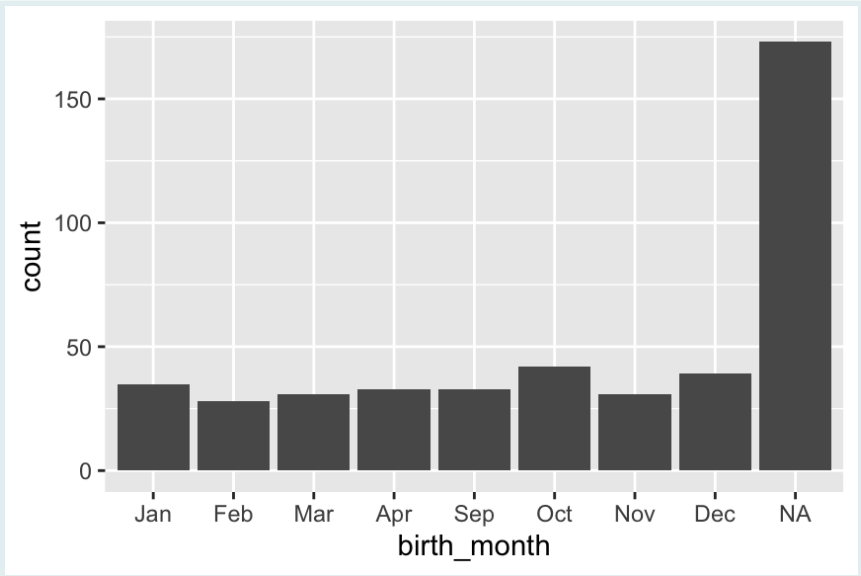
::: watch-out Be mindful when creating factor levels! Any values in the variable that are *not* included in the set of levels provided to the `levels` argument will be converted to NA.

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:

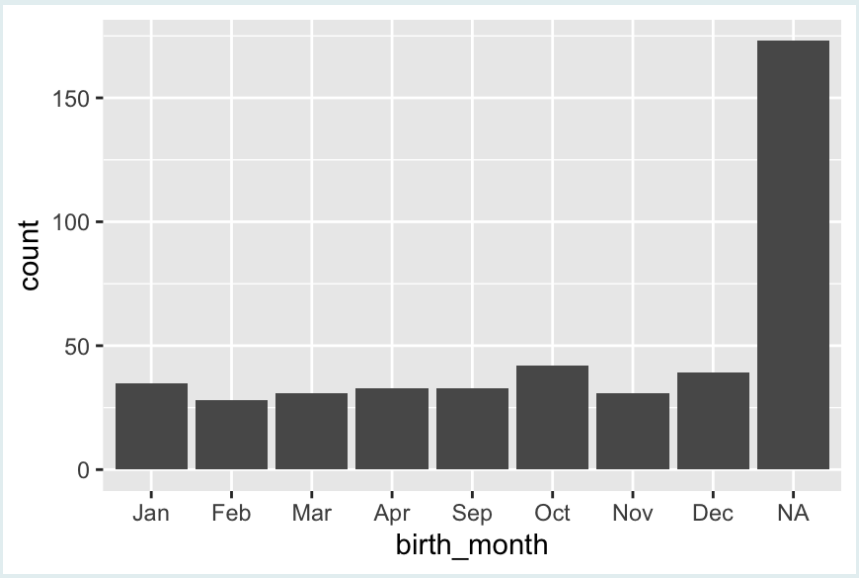
```
ggplot(hiv_mort_missing_months) +
  geom_bar(aes(x = birth_month))
```



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

```r
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")))
```

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



SIDE NOTE

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

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

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

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

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

SIDE NOTE

```
##  [1] "Apr" "Aug" "Dec" "Feb" "Jan" "Jul" "Jun" "Mar" "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`

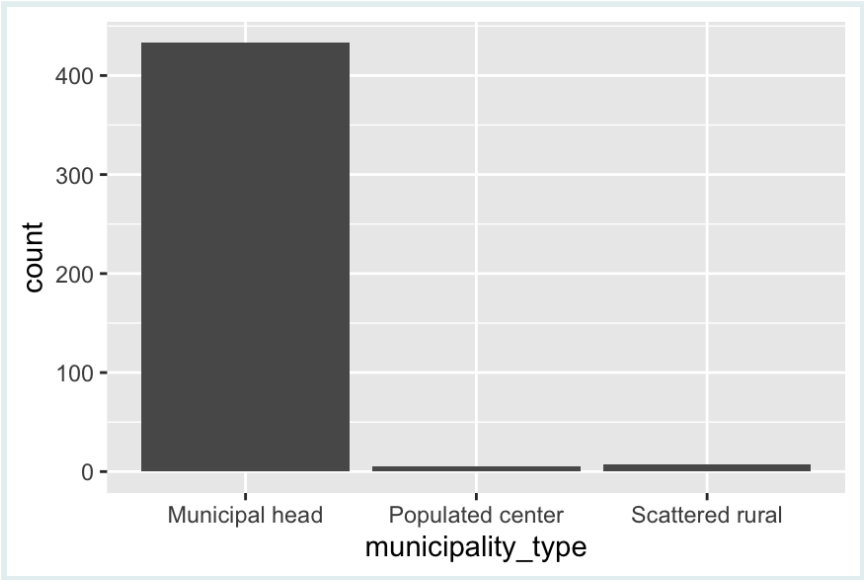
Factors are very useful, but they can sometimes be a little tedious to manipulate using base R functions alone. Thankfully, the `forcats` package, a member of the tidyverse, offers a set of functions that make factor manipulation much simpler. We'll consider four functions here, but there are many others, 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?

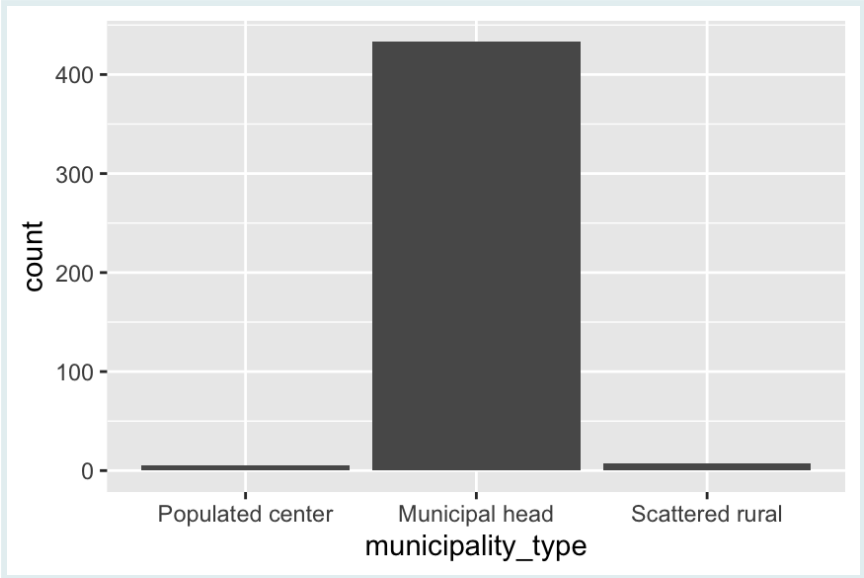
This can be achieved using `fct_relevel()`. Here's how:

```
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:

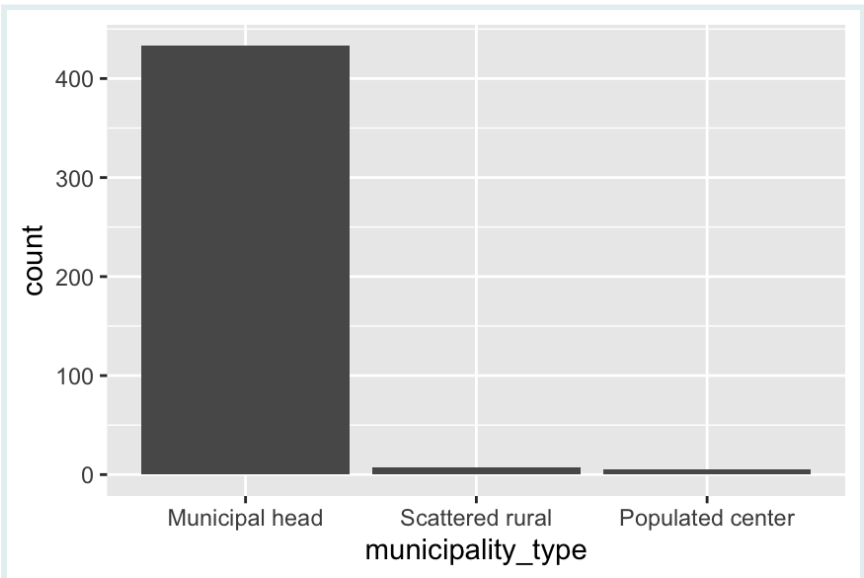
```
ggplot(hiv_mort_pop_center_first) +
  geom_bar(aes(x = municipality_type))
```



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))
```

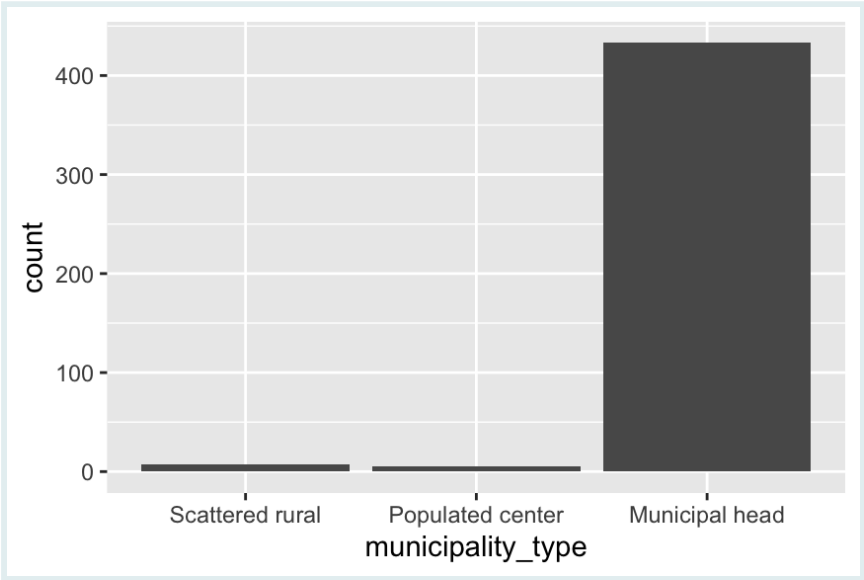


The syntax is: specify the factor, the level to move, and use the `after` argument to define what position to place it after.

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")))

ggplot() +
  geom_bar(aes(x = municipality))
```

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 med_age_death
##    <chr>                  <int>          <dbl>         <dbl>
##  1 Aguadas                    2           42            42
##  2 Anserma                   15           37.4          37.5
##  3 Aranzazu                   2           37.5          37.5
##  4 Belalcázar                 4           38.8          41
##  5 Chinchiná                 62           43.6          42.5
##  6 Filadelfia                 5           42.6          43
##  7 La Dorada                 46           41.0          41
##  8 La Merced                  3           27            28
##  9 Manizales                199           41.0          41
## 10 Manzanares                 3           38.3          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:

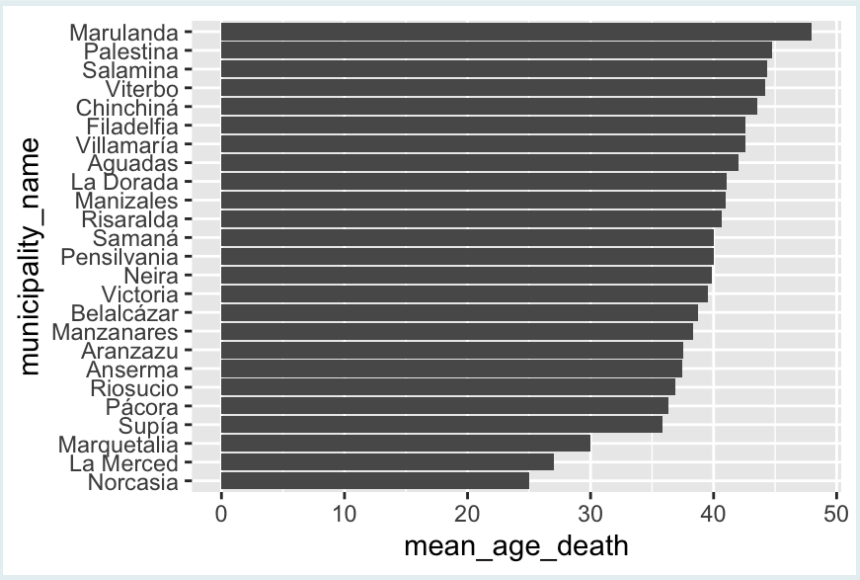
```
summary_per_muni_reordered <-
  summary_per_muni %>%
  mutate(municipality_name = fct_reorder(.f = municipality_name,
                                         .x = mean_age_death))
```

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))
```



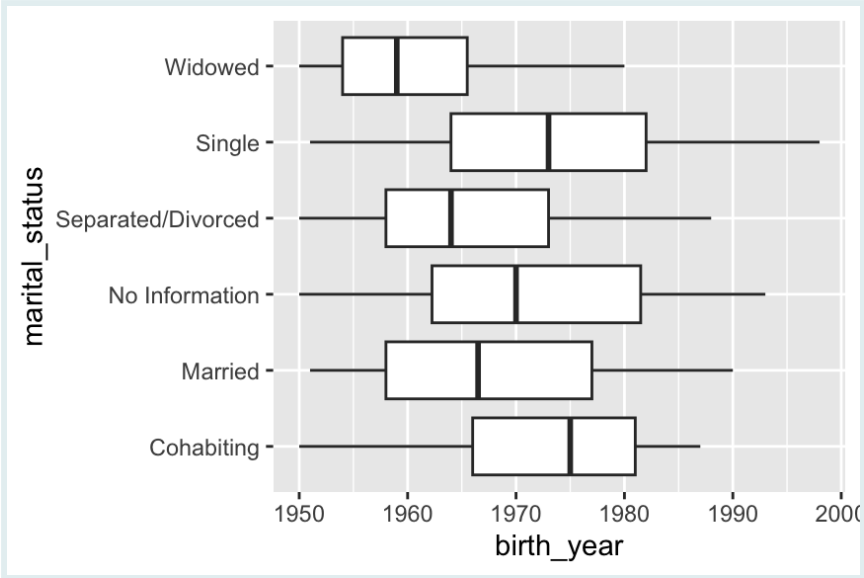
Q: Using fct_reorder

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).

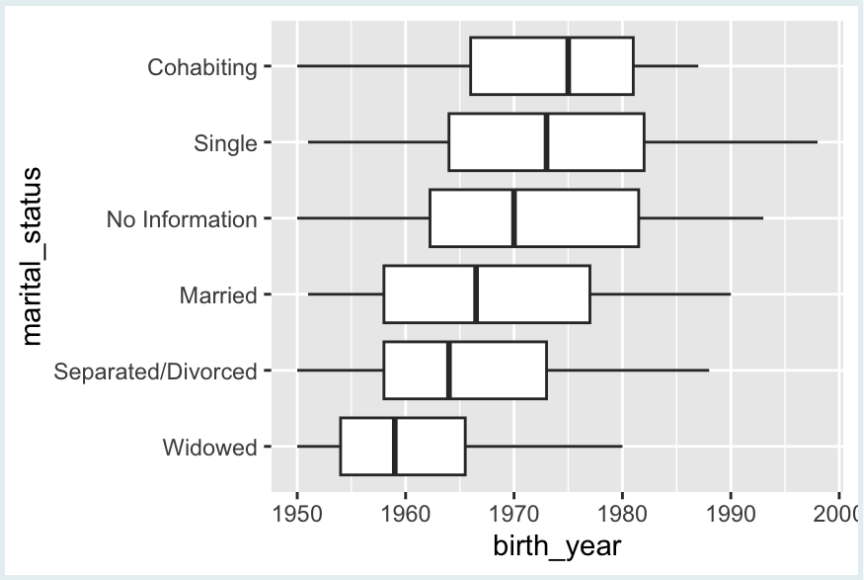
This is where the `.fun` argument of `fct_reorder()` comes in. The `.fun` argument allows us to specify a summary function that will be used to calculate the new order of the levels:

```
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" "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
```

SIDE NOTE

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)
```

```
## [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

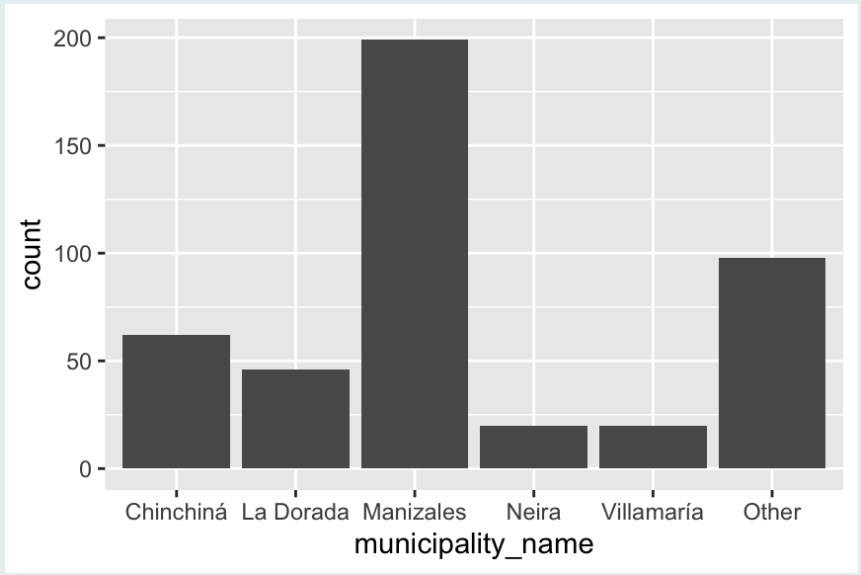
Sometimes, we have too many levels for a display table or plot, and we want to lump the least frequent levels into a single category, typically called 'Other'.

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))
```

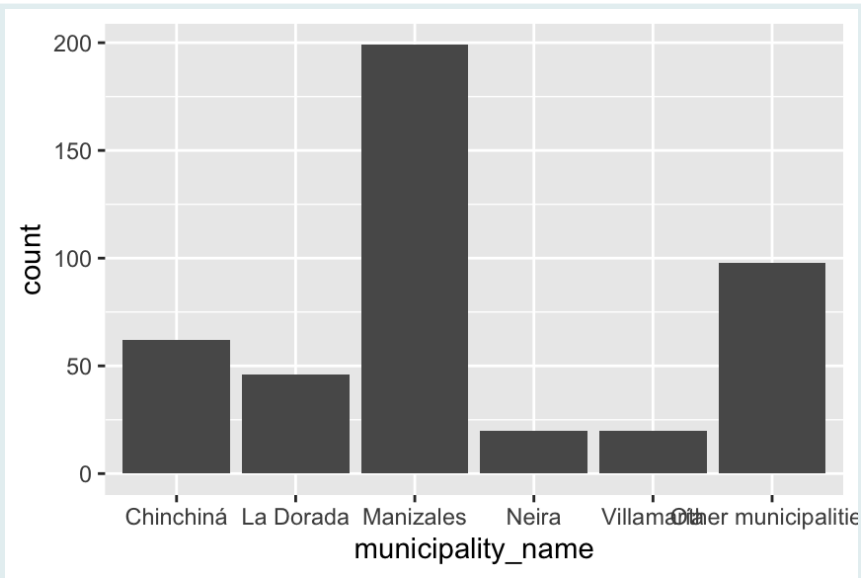
```
ggplot(hiv_mort_lump_muni, aes(x = municipality_name)) +  geom_bar()
```



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.

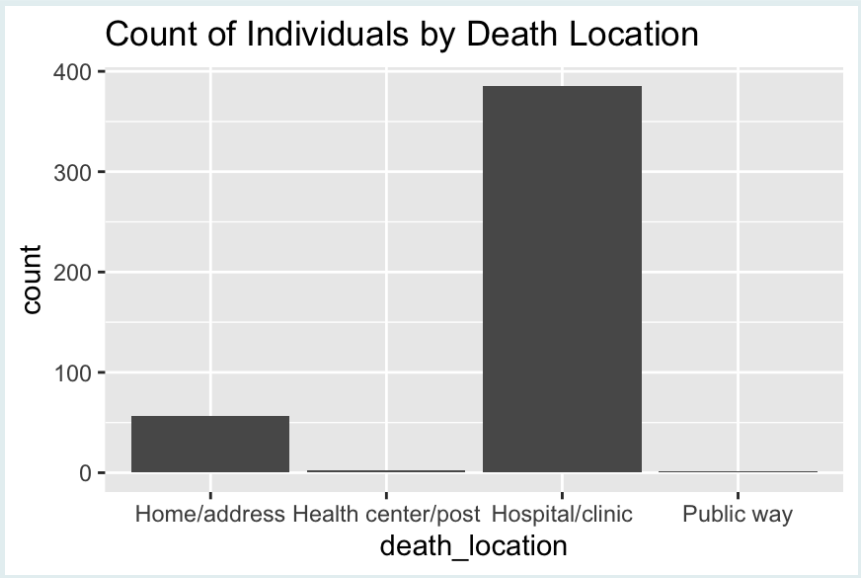
The other two statements are not true.

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".

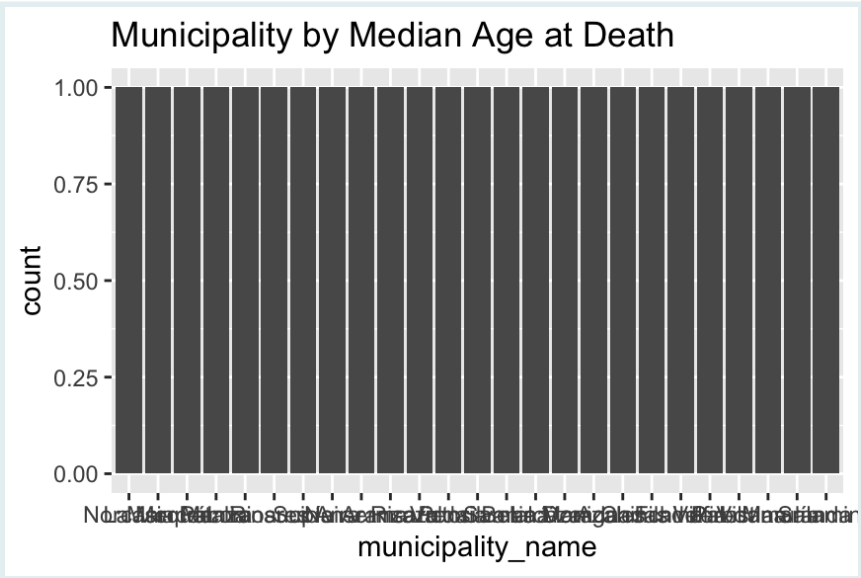
Q: Using fct_relevel

```
# Convert death_location to a factor with 'Home/address' as the first level
hiv_mort <- hiv_mort %>%
  mutate(death_location = fct_relevel(death_location, "Home/address"))

# Create a bar plot of death_location
ggplot(hiv_mort, aes(x = death_location)) +
  geom_bar() +
  labs(title = "Count of Individuals by Death Location")
```



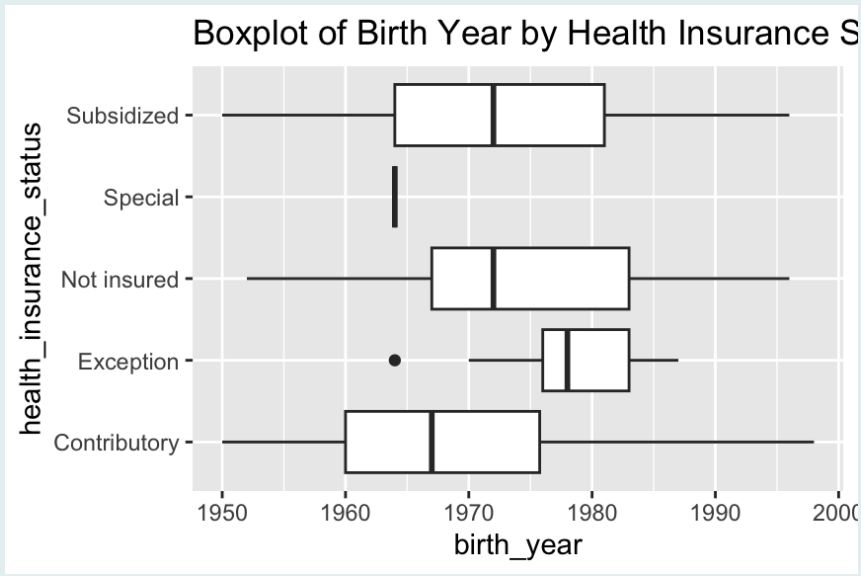
Q: Using fct_reorder

```
# Reorder municipality_name by med_age_death and plot
summary_per_muni <- summary_per_muni %>%
  mutate(municipality_name = fct_reorder(municipality_name, med_age_death))

# Create a bar plot of reordered municipality_name
ggplot(summary_per_muni, aes(x = municipality_name)) +
  geom_bar() +
  labs(title = "Municipality by Median Age at Death")
```



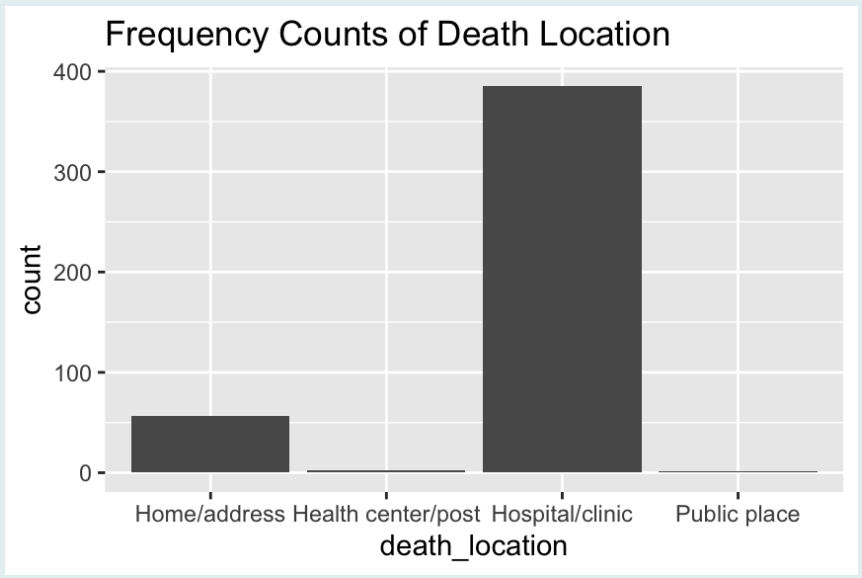
Q: Using .fun

```
# Create a boxplot of birth_year by health_insurance_status
ggplot(hiv_mort, aes(x = health_insurance_status, y = birth_year)) +
  geom_boxplot() +
  labs(title = "Boxplot of Birth Year by Health Insurance Status") +
  coord_flip()  # To display categories on the y-axis
```



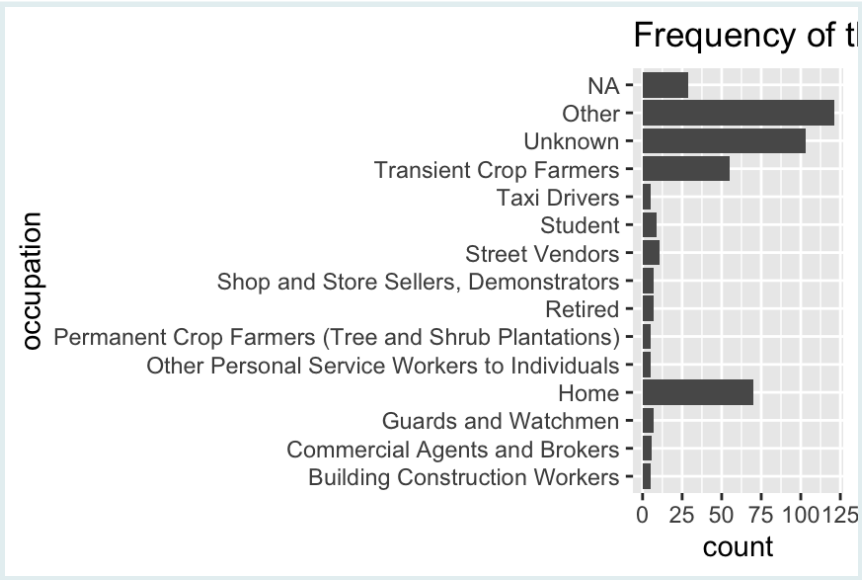
Q: Using fct_recode

```
# Convert death_location to a factor and rename 'Public way' to 'Public
place'
hiv_mort <- hiv_mort %>%
  mutate(death_location = fct_recode(death_location, "Public place" =
"Public way"))

# Plot frequency counts of the updated variable
ggplot(hiv_mort, aes(x = death_location)) +
  geom_bar() +
  labs(title = "Frequency Counts of Death Location")
```



Q: Using fct_lump

```r
# Create a bar chart with the frequency of the 10 most common occupations
hiv_mort <- hiv_mort %>%
  mutate(occupation = fct_lump(occupation, n = 10, other_level = "Other"))

# Create the bar plot with occupation on the y-axis
ggplot(hiv_mort, aes(x = occupation)) +
  geom_bar() +
  labs(title = "Frequency of the 10 Most Common Occupations") +
  coord_flip()  # To display categories on the y-axis
```



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_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

Dates 1: Recognizing and Formatting Dates

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
## # ℹ 5 more variables: start_date_default <date>,
## #   end_date_default <date>, start_date_typical <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
## # ℹ 3 more variables: adm_date_abbrev <chr>,
## #   adm_date_messy <chr>, disch_date_default <date>
```

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"
```

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.

SIDE NOTE

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 thinking too much about 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
## # ℹ 102 more rows
## # ℹ 2 more variables: start_date_long <chr>, …
```

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.

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
## # ℹ 5 more variables: start_date_default <date>,
## #   end_date_default <date>, start_date_typical <date>, …
```

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", "dmy", "ymd")))
```

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")))
```

Format date vector

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

Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

Dates 2: Intervals, Components and Rounding

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 progression of diseases over time and the fluctuation in population health metrics across different periods.

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,
               here,
               lubridate)
```

Dataset

The two datasets we will be working with contain information related to indoor residual spraying (IRS) for malaria control efforts in Illovo, Malawi.

The first dataset details the start and end dates of mosquito spraying campaigns in different villages. More information about the dataset can be found in the previous lesson on dates.

```
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
## # ℹ 5 more variables: start_date_default <date>,
## #   end_date_default <date>, start_date_typical <chr>, …
```

The second dataset contains monthly data from 2015-2019 comparing the average incidence of malaria per 1000 people in villages that received indoor residual spraying (IRS) versus villages that did not.

```
incidence_temp <- read_csv(here("data/Illovo_ir_weather.csv"))
incidence_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 `ir_case` column shows malaria incidence in IRS villages and `ir_control` shows incidence in non-IRS villages. The `date` column contains the month and random day. The dataset also includes average monthly minimum and maximum temperatures (`avg_min` and `avg_max`).

The final dataset contains 1,460 rows of daily weather data for the Illovo region from 2015-2019.

```
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
```

Each row represents a single day and includes measurements of minimum temperature (`min_temp`) in degrees Celsius, maximum temperature (`max_temp`) in degrees Celsius, and rainfall (`rain`) in millimeters.

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. This is sometimes called the interval operator. 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
```

This means there were six 5-day-periods in that interval.

Lubridate weeks

Use the `weeks()` function in place of `days()` in the lubridate method to calculate the time difference in weeks between the two dates below:

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

Comparison

So which of the methods is best? Lubridate provides more flexibility and accuracy when working with dates in R. Let's look at a simple example to see why.

First let's set two dates that are 6 years apart:

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

If we want to calculate how many years have passed between these dates, how would we proceed in base R? We might first subtract the two dates, `date_2 - date_1`, then divide by an average day count, like 365.25 (accounting for leap years):

```
(date_2 - date_1)/365.25
```

```
## Time difference of 6.001369 days
```

The result is close to 6 years but not precise due to the averaging of leap years!

The result is still given in "days". You can convert the time difference to a numeric value easily:

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

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

Dividing by 365 or 366 will also give imprecise results:

```
(date_2 - date_1)/365
```

```
## Time difference of 6.005479 days
```

```
(date_2 - date_1)/366
```

```
## Time difference of 5.989071 days
```

What you need to do is take into account that there are just two leap years (two extra days) between those dates and subtract those two days out first:

```
(date_2 - date_1 - 2)/365
```

```
## Time difference of 6 days
```

But this will be a painful thing to do in practice when working with real data. With lubridate intervals, the process is more straightforward, as leap years are handled for you:

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

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

The difference is slight, but lubridate is clearly a winner in this situation.

Although we don't cover them in this course, {lubridate} is also great for dealing with time irregularities like time zones and daylight savings shifts.

Lubridate intervals

Can you apply lubridate's interval function 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`.

SIDE NOTE

Lubridate has a technical distinction between "intervals", "periods" and "durations". You can find out more here

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
```

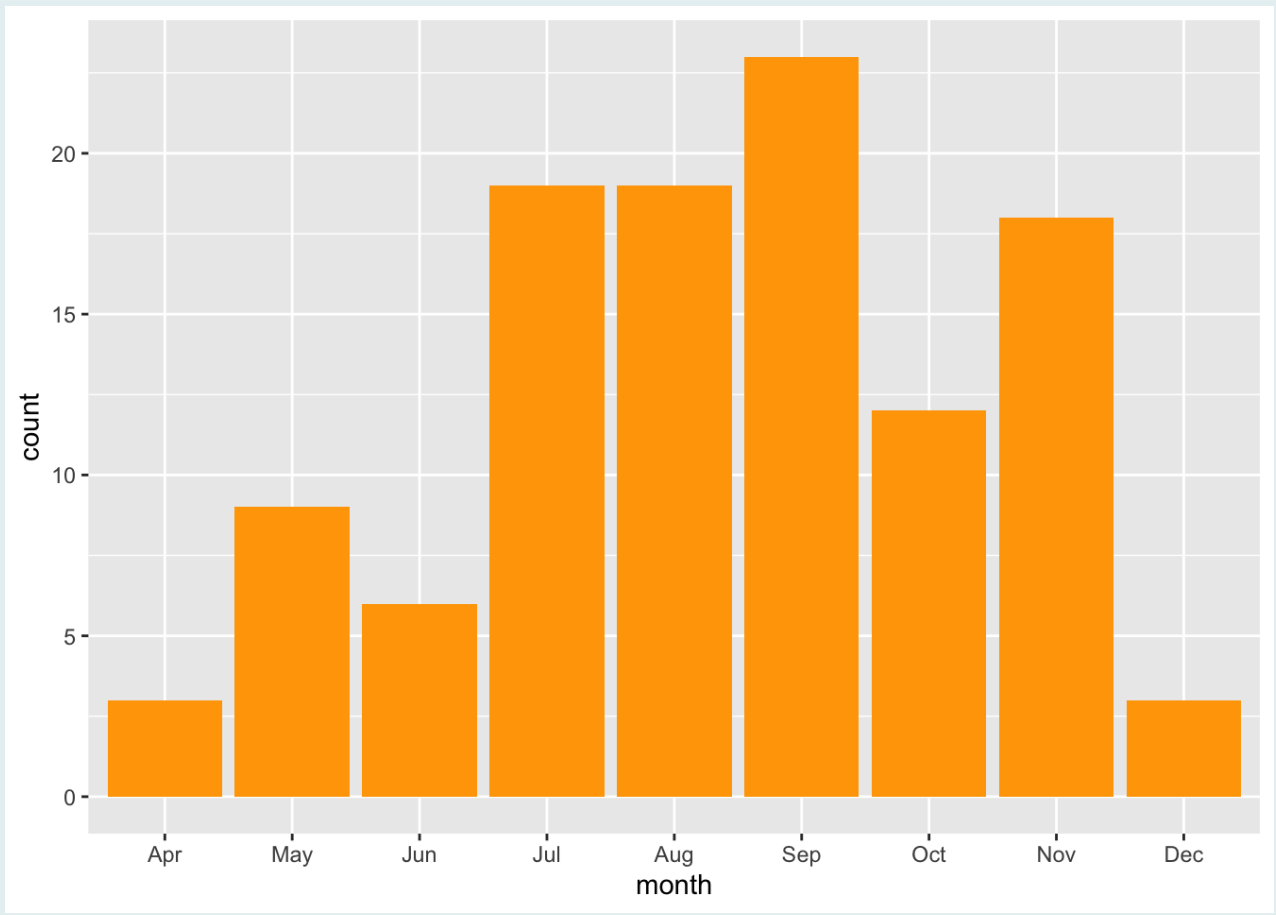
Extracting weekdays

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.

For example, let's say we wanted to visualize 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 fits the pattern observed.

Visualizing spray end months

Using the `irs` dataset, create a new graph showing the months when the spraying campaign ended and compare it to the graph of when they started. Do they have a similar pattern?

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"` argument.

```
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.

Rounding dates practice

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:

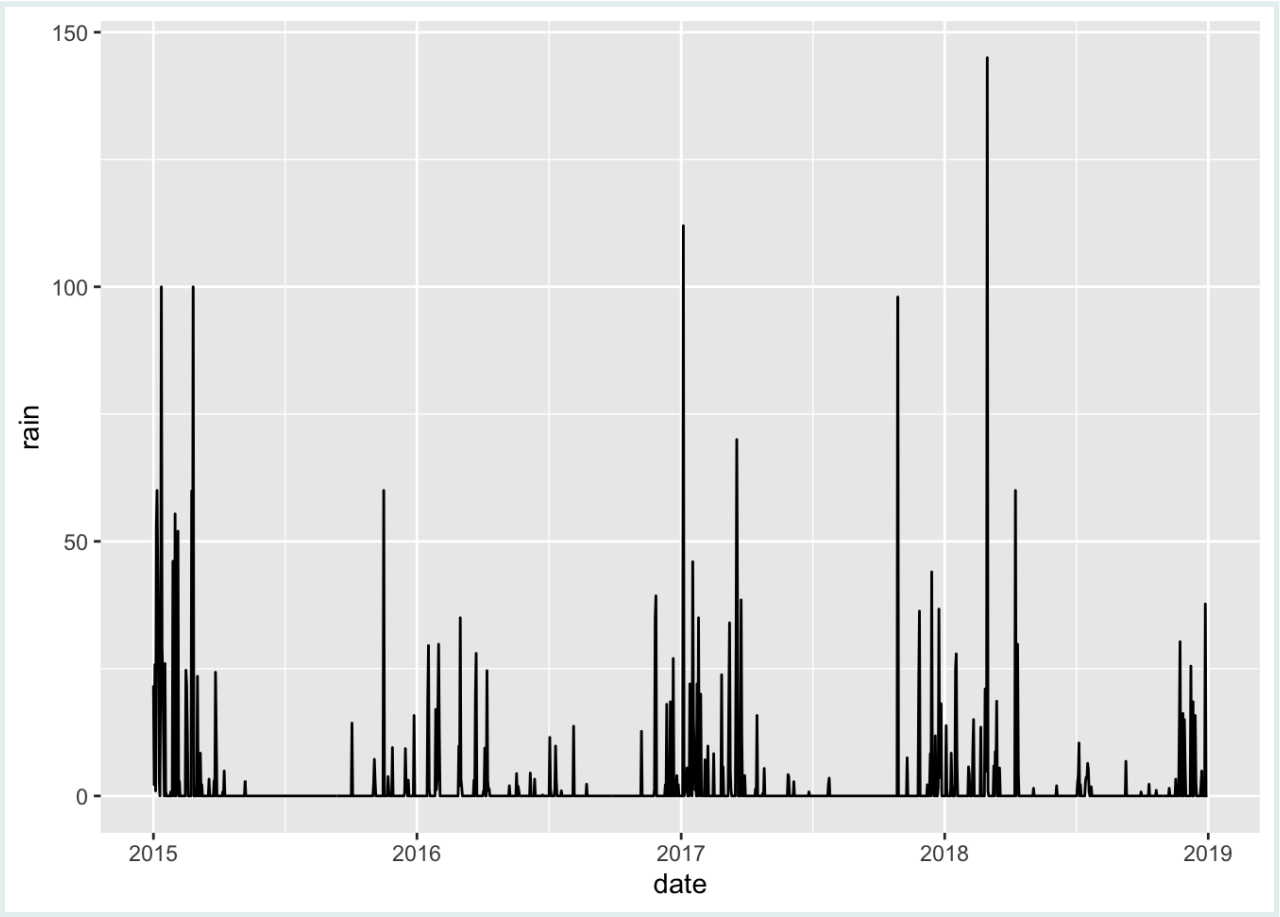
```
date_round <- as.Date("2001-11-29")
floor_date(date_round, unit="year")
```

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:

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



Aside from being visually messy, it is a little bit difficult to see seasonal patterns. 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 %>%
  mutate(month_year=str_sub(date, 1, 7))
```

Next, we group by `month_year` to calculate the average rainfall for each month:

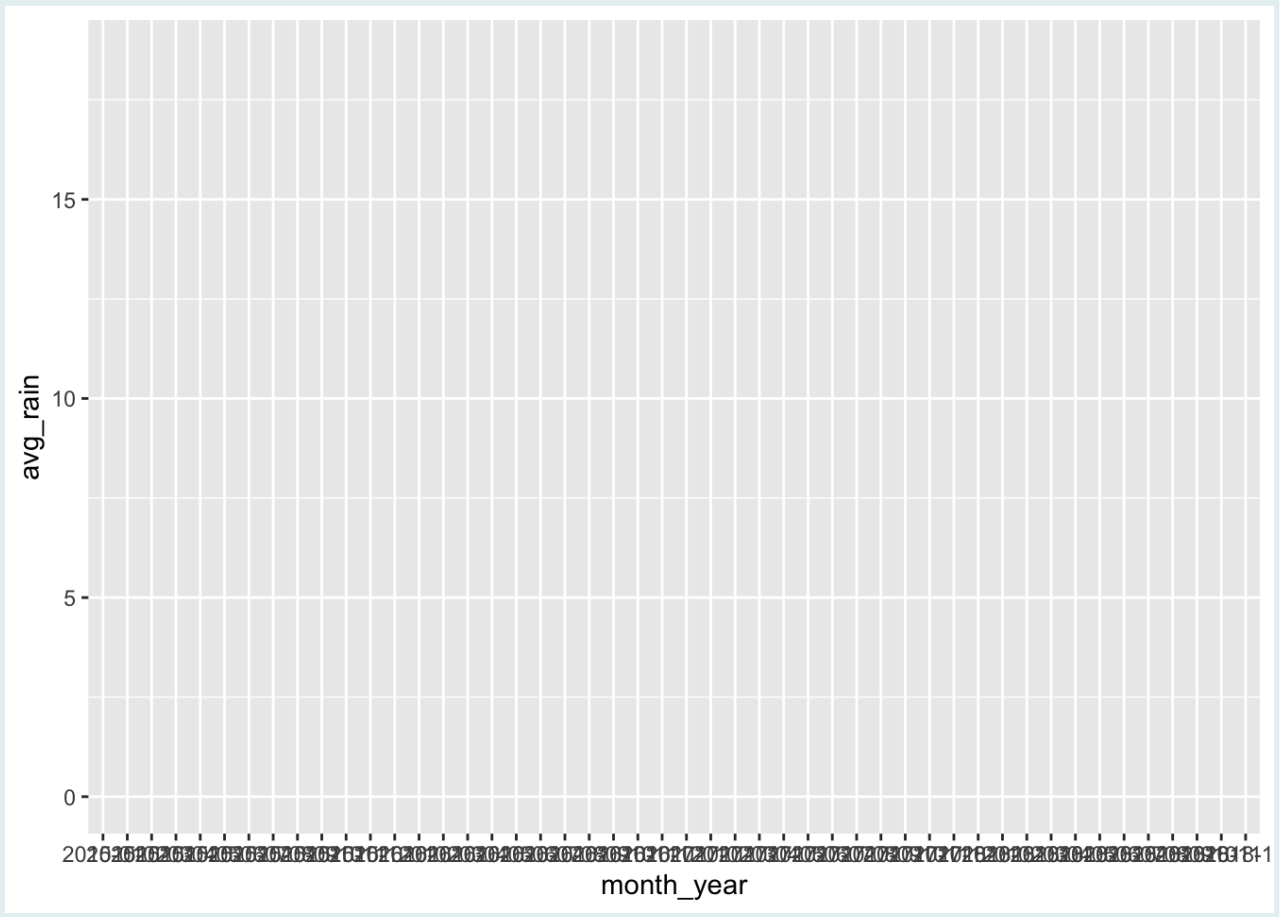
```
weather_summary_1 <-
  weather %>%
  mutate(month_year=str_sub(date, 1, 7)) %>%
  # group and summarise
  group_by(month_year) %>%
  summarise(avg_rain=mean(rain))
weather_summary_1
```

```
## # A tibble: 48 × 2
##    month_year avg_rain
##    <chr>         <dbl>
##  1 2015-01     18.6
##  2 2015-02     10.4
##  3 2015-03      2.69
##  4 2015-04      0.19
##  5 2015-05      0.0968
##  6 2015-06      0
##  7 2015-07      0
##  8 2015-08      0
##  9 2015-09      0
## 10 2015-10      0.461
## # ℹ 38 more rows
```

However, we encounter a problem when trying to plot this data. Our `month_year` variable is now a character, not a date. This means it's not continuous. Plotting a line graph with a non-continuous variable does not work:

```
weather_summary_1 %>%
  ggplot() +
  geom_line(aes(month_year, avg_rain))
```

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

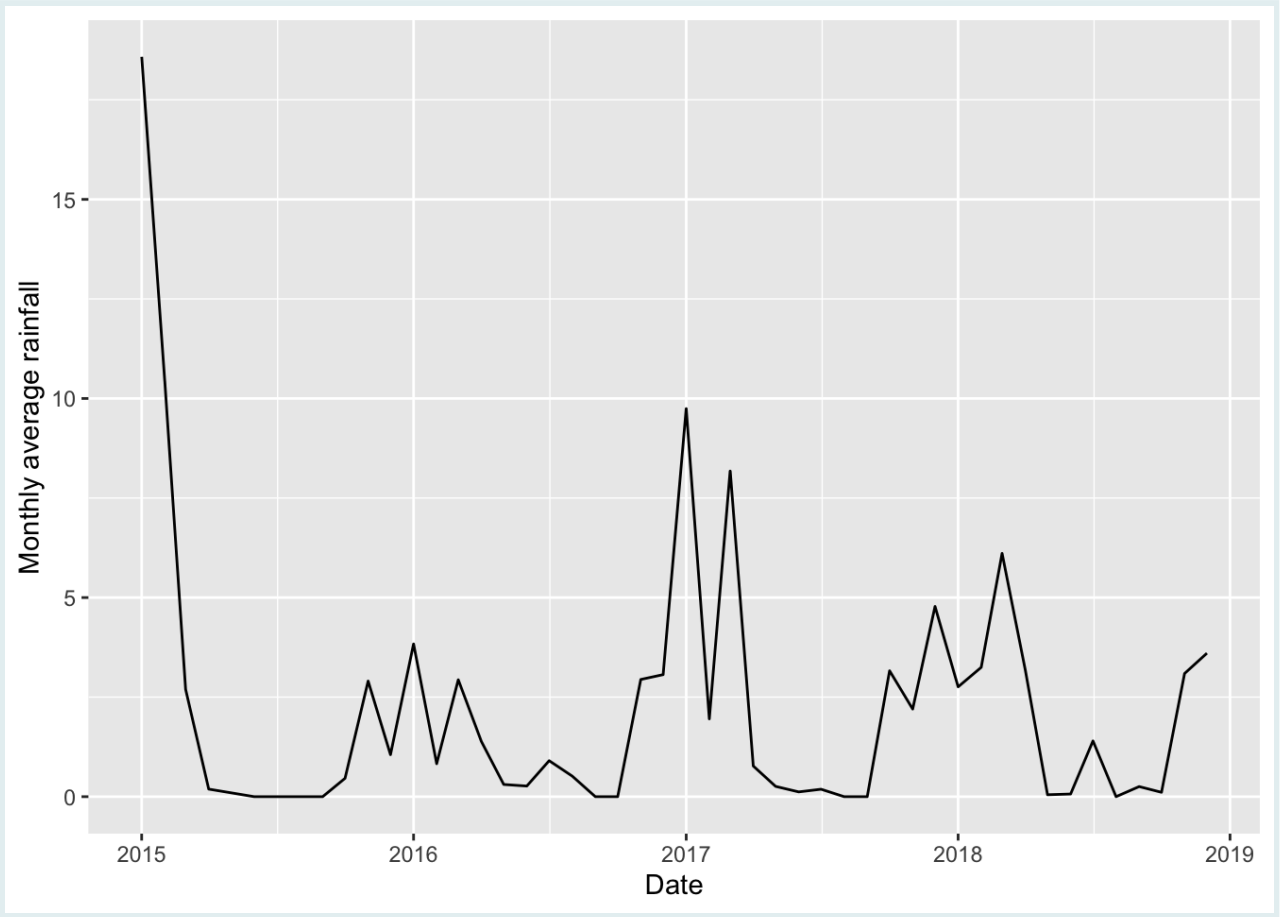


The best way to do this is to first round our dates down to the month using `floor_date()` function, then group our data by our new `month_year` variable, and then calculate the monthly average. Let's try it out now.

```
weather_summary_2 <- weather %>%
  mutate(month_year=floor_date(date, unit="month")) %>%
  group_by(month_year) %>%
  summarise(avg_rain=mean(rain))
weather_summary_2
```

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

```
weather_summary_2 %>%
  ggplot() +
  geom_line(aes(month_year, avg_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.

Plot avg monthly min and max temperatures

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

Wrap Up!

This lesson covered fundamental skills for working with dates in R - calculating intervals, extracting components, rounding, and creating time series visualizations. With these key building blocks now mastered, you can can now start to wrangle date data to uncover and analyze patterns over time.

Answer Key

Lubridate weeks

```
oct_31 <- as.Date("2023-10-31")
jul_20 <- as.Date("2023-07-20")
time_difference <- oct_31 %--% jul_20
time_difference/weeks(1)
```

```
## [1] -14.71429
```

Lubridate intervals

```
irs %>%
  mutate(spraying_time = interval(start_date_default,
end_date_default)/days(1)) %>%
  select(spraying_time)
```

```
## # A tibble: 112 × 1
##    spraying_time
##            <dbl>
##  1            10
##  2             5
##  3             0
##  4             0
##  5             0
##  6            11
##  7             0
##  8             0
##  9            19
## 10             9
## # ℹ 102 more rows
```

Extracting weekdays

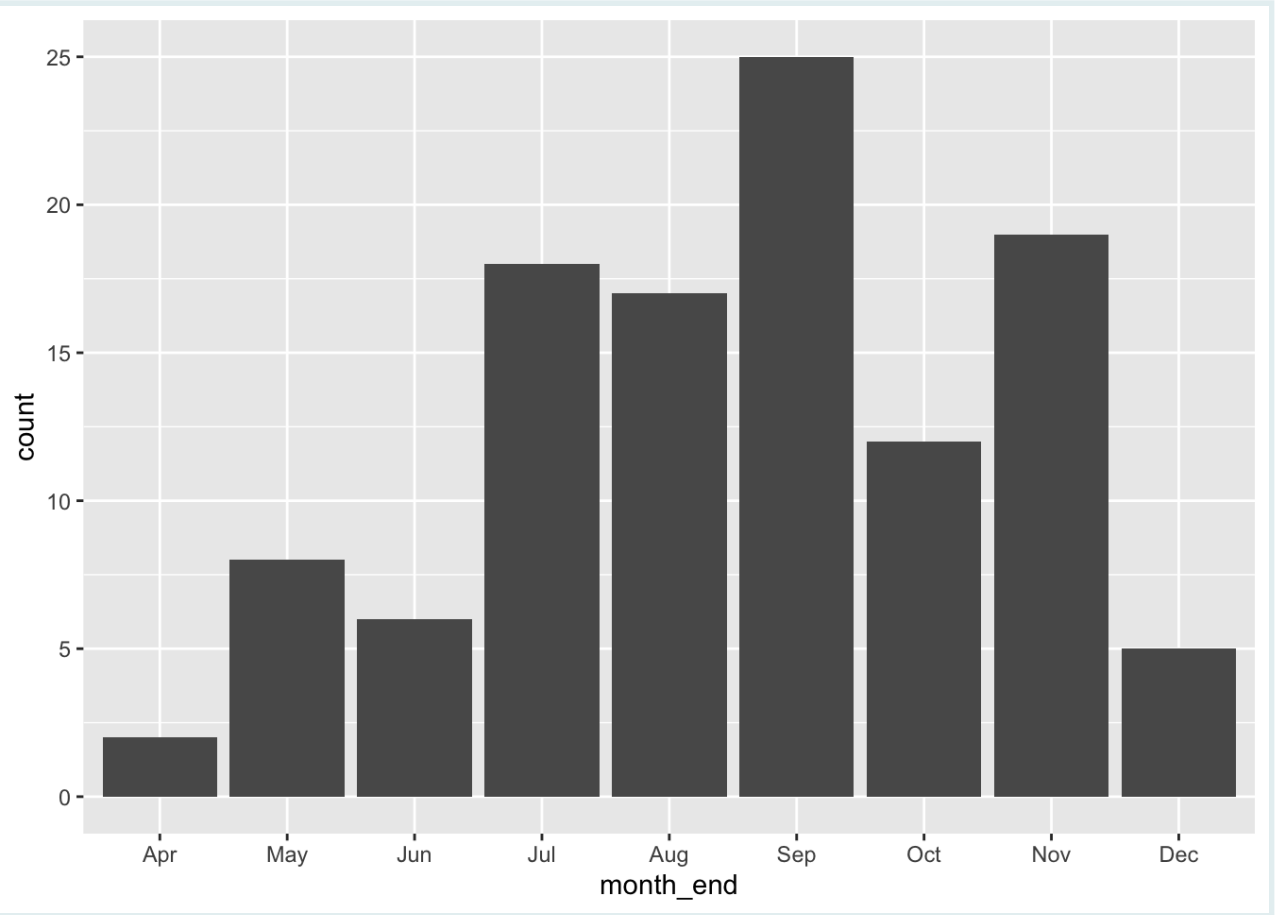
```
irs %>%
  mutate(wday_start = wday(start_date_default, label = TRUE)) %>%
  select(wday_start)
```

```
## # A tibble: 112 × 1
##    wday_start
##    <ord>
##  1 Mon
##  2 Tue
##  3 Tue
##  4 Tue
##  5 Tue
```

```
##  6 Thu
##  7 Tue
##  8 Tue
##  9 Wed
## 10 Wed
## # ℹ 102 more rows
```

Visualizing spray end months

```
irs %>%
  mutate(month_end = month(end_date_default, label = TRUE)) %>%
  ggplot(aes(x = month_end)) +
  geom_bar()
```



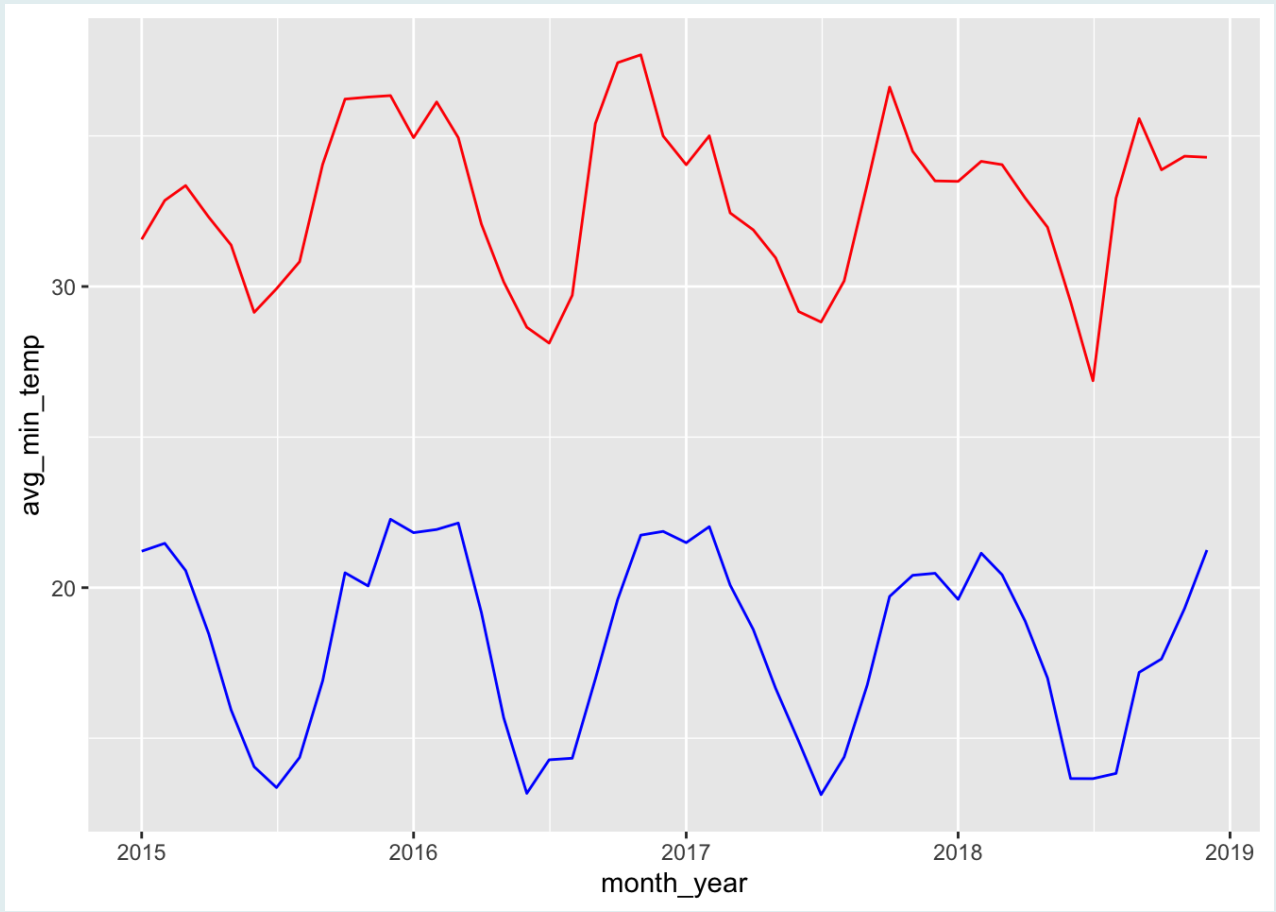
Rounding dates practice

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

```
## [1] "2001-01-01"
```

Plot avg monthly min and max temperatures

```r
weather %>%
  mutate(month_year = floor_date(date, unit="month")) %>%
  group_by(month_year) %>%
  summarise(avg_min_temp = mean(min_temp),
            avg_max_temp = mean(max_temp)) %>%
  ggplot() +
  geom_line(aes(x = month_year, y = avg_min_temp), color = "blue") +
  geom_line(aes(x = month_year, y = avg_max_temp), color = "red")
```



Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network

Intro to Joining Datasets

Prelude

Joining datasets is a crucial skill when working with health-related data as it allows you to combine information about the same entities 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: left, right, inner and full.
- 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
```

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!

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()`.

Here's how to do this:

```
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."

Left Join Patients and Checkups

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()`

Left Join with by Argument

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

```r
# 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:

```r
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.

VOCAB

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

Left Join Diagnoses and Demographics

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",
  4,           "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"))
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, for countries from all continents:

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

Left Join TB Cases and Continents

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.

Inner Join Pathogens

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 TB incidence 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
```

Works great!

Notice that there are now only 44 rows in the output, since the three african countries without corresponding expenditure information in the `health_exp_2019` were dropped.

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.

Inner Join One Row

The code chunk below filters the `health_exp_2019` dataset to the 70 countries with the highest spending:

```
highest_exp <-
  health_exp_2019 %>%
  arrange(-expend_usd) %>%
  head(70)
```

Use an `inner_join()` to join this `highest_exp` dataset with the African TB incidence dataset, `tb_2019_africa`.

If you do this correctly, there will be just one row returned. Why?

`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.

Full Join Malaria Data

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)
```

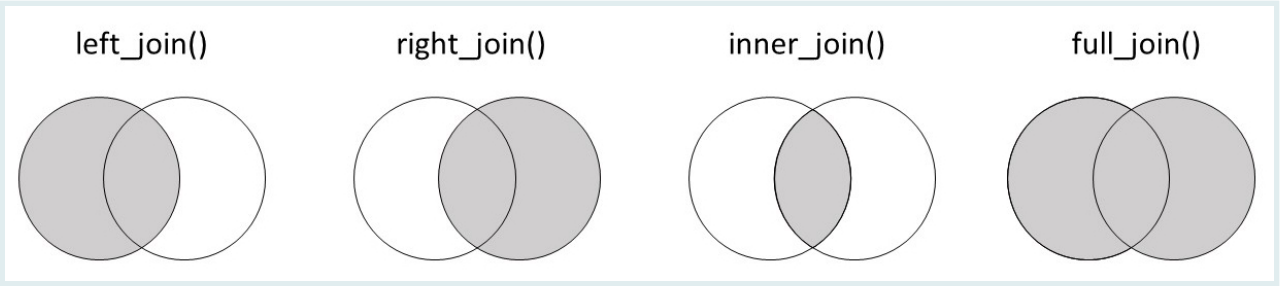
```
## Joining with `by = join_by(country)`
```

```
full_tb_health
```

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

Wrap Up!

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!



Answer Key

Q: Left Join Patients and Checkups

```
left_join(x=patients, y=checkups)
```

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

```
## # A tibble: 3 × 4
##   patient_id name    age checkup_date
##        <dbl> <chr> <dbl> <chr>
## 1          1 John     32 2023-01-20
## 2          2 Joy      28 2023-02-20
## 3          3 Khan     40 2023-05-15
```

Q: Left Join with by Argument

```
left_join(x=patient_details, y=vaccination_records,
by=c("id_number"="patient_code"))
```

```
## # A tibble: 3 × 5
##   id_number full_name address      vaccine_type
##   <chr>     <chr>     <chr>        <chr>
## 1 A001      Alice     123 Elm St   COVID-19
## 2 B002      Bob       456 Maple Dr Flu
## 3 C003      Charlie   789 Oak Blvd Hepatitis B
## # ℹ 1 more variable: vaccination_date <chr>
```

Q: Left Join Diagnoses and Demographics

```
left_join(x=patient_demographics, y=disease_dx)
```

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

```
## # A tibble: 5 × 6
##   patient_id name        age gender disease
##        <dbl> <chr>     <dbl> <chr>  <chr>
## 1          1 Fred         28 Female Influenza
## 2          2 Genevieve    45 Female <NA>
## 3          3 Henry        32 Male   <NA>
## 4          5 Irene        55 Female <NA>
## 5          8 Jules        40 Male   Influenza
## # ℹ 1 more variable: date_of_diagnosis <chr>
```

Q: Left Join TB Cases and Continents

```
left_join(x=tb_cases_children, y=country_continents,
by=c(country="country.name.en"))
```

```
## # A tibble: 5 × 4
##   country        tb_cases_smear_…¹ continent region
##   <chr>                      <dbl> <chr>     <chr>
## 1 Afghanistan                  588 Asia      South Asia
## 2 Albania                        0 Europe    Europe & Centr…
## 3 Algeria                       89 Africa    Middle East & …
## 4 American Samoa                NA Oceania   East Asia & Pa…
## 5 Andorra                        0 Europe    Europe & Centr…
## # ℹ abbreviated name: ¹tb_cases_smear_0_14
```

Q: Inner Join Pathogens

```
inner_join(total_inf, outcomes)
```

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

```
## # A tibble: 2 × 4
##   pathogen      total_infections n_hosp n_deaths
##   <chr>                    <dbl>  <dbl>    <dbl>
## 1 Campylobacter             9751   1938       42
## 2 Listeria                   136    128       30
```

Q: Inner Join One Row

```
inner_join(highest_exp, tb_2019_africa)
```

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

```
## # A tibble: 1 × 4
##   country    expend_usd cases conf_int_95
##   <chr>           <dbl> <dbl> <chr>
## 1 Seychelles        572    15 [13 – 18]
```

There is only one country in common between the two datasets.

Q: Full Join Malaria Data

```
full_join(malaria_inc, malaria_deaths)
```

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

```
## # A tibble: 5 × 3
##    year inc_100k deaths_100k
##   <dbl>    <dbl>       <dbl>
## 1  2010     69.5        NA
## 2  2011     66.5        12.9
## 3  2014     59.8        NA
## 4  2016     58.7         9.4
## 5  2017     59.2        NA
```

Contributors

The following team members contributed to this lesson:

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

Joining 2: Mismatched Values, One-to-Many & Multi-Key Joins

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, countrycode)
```

Pre-join data cleaning: addressing data inconsistencies

A toy example

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 inconsistencies 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
```

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

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

Now let's try a `left_join()` and `inner_join()` on our two datasets.

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

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

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

```
## # A tibble: 1 × 4
##   name    age test_date  result
##   <chr> <dbl> <chr>      <chr>
## 1 Bob      32 2023-08-10 Positive
```

As we can see, R didn't recognize `Alice` and `alice` as the same person, and it also could not match `Charlie` and `charlie`. So in the `left_join()`, `Alice` and `Charlie` are left with NAs, and in the `inner_join()`, they are dropped.

How can we fix this? We need to ensure that the names in both datasets are in title case, with a capitalized first letter. For this we can use `str_to_title()`. Let's try it:

```
test_info_title <- test_info %>%
  mutate(name = str_to_title(name)) # convert to title case
test_info_title
```

```
## # 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 2023-05-02 Negative
```

```
left_join(demographic, test_info_title, 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 2023-05-02 Negative
```

```
inner_join(demographic, test_info_title, 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 2023-05-02 Negative
```

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

Inner Join countries

The following two datasets contain data for India, Indonesia, and the Philippines. However an `inner_join()` of these datasets produces no output. 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
)

inner_join(df1, df2)
```

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

```
## # A tibble: 0 × 4
## # ℹ 4 variables: Country <chr>, Capital <chr>,
## #   Population <dbl>, Life_Expectancy <dbl>
```

Real Data Example 1: Key Typos

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 TB notifications (TB cases or relapses) in 2022 for all 36 Indian states and Union Territories, taken from the Government of India Tubrculosis Report.

```
tb_notifs <- read_csv(here("data/notif_TB_india_modified.csv"))

tb_notifs_public <- tb_notifs %>%
  filter(hc_type == "public") %>% # we want only public systems for now
  select(-hc_type)

tb_notifs_public
```

```
## # A tibble: 5 × 2
##   state                     tb_notif_count
##   <chr>                              <dbl>
## 1 Andaman & Nicobar Islands            510
## 2 Andhra Pradesh                     62075
## 3 Arunachal Pradesh                   2722
## 4 Assam                              36801
## 5 Bihar                              79008
```

The variables are the state/Union Territory name and the number of TB notifications from 2022.

Our second dataset is on COVID screening among TB cases for 36 Indian states taken from the same TB Report:

```
covid_screening <- read_csv(here("data/COVID_india_modified.csv"))

covid_screening_public <- covid_screening %>%
  filter(hc_type == "public") %>% # we want only public systems for now
  select(-hc_type)

covid_screening_public
```

```
## # A tibble: 5 × 2
##   state                     tb_covid_pos
##   <chr>                            <dbl>
## 1 Andaman & Nicobar Islands            0
## 2 Andhra Pradesh                      97
## 3 ArunachalPradesh                     0
```

```
## 4 Assam                                31
## 5 Bihar                                78
```

It contains the state/Union Territory name and the number of TB patients who tested positive for COVID-19, `tb_covid_pos`. Note that there are some missing values in this dataset.

Now, we'd like to join these two datasets, to allow us to calculate the percentage of TB patients in each state who tested positive for COVID-19.

Let's give it a go using `inner_join()`:

```
tb_notifs_and_covid_screening <-
  inner_join(tb_notifs_public, covid_screening_public)
```

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

```
tb_notifs_and_covid_screening
```

```
## # A tibble: 5 × 3
##   state                     tb_notif_count tb_covid_pos
##   <chr>                              <dbl>        <dbl>
## 1 Andaman & Nicobar Islands            510            0
## 2 Andhra Pradesh                     62075           97
## 3 Assam                              36801           31
## 4 Bihar                              79008           78
## 5 Chandigarh                          5664            8
```

We can now perform the percentage calculation:

```
tb_notifs_and_covid_screening %>%
  mutate(pct_covid_pos = 100 * tb_covid_pos/tb_notif_count)
```

```
## # A tibble: 5 × 4
##   state            tb_notif_count tb_covid_pos pct_covid_pos
##   <chr>                     <dbl>        <dbl>         <dbl>
## 1 Andaman & Nicob…            510            0         0
## 2 Andhra Pradesh            62075           97         0.156
## 3 Assam                     36801           31         0.0842
## 4 Bihar                     79008           78         0.0987
## 5 Chandigarh                 5664            8         0.141
```

Seems alright!

However, notice that we now only have 32 rows in the output dataset, even though both initial datasets had 36 rows. Whenever you lose data in this way, it is worth investigating.

In this case, it turns out that there are several regions spelled differently in the two datasets. Because of these "key typos" could not be joined and were therefore dropped. How can we identify these to avoid information loss?

Recall that a "key" refers to the column(s) used to match rows across datasets in a join. The join matches rows that have identical values in the key columns.

Key typos are spelling or formatting inconsistencies in the values of key columns across datasets. For example, one dataset may list "New Delhi" while the other lists "Delhi". Because of these inconsistencies, rows that should match can't be joined properly and are dropped.

Identifying unmatched values with `setdiff()`

To identify these key typos, we can compare which values are in one data frame but not the other using the `setdiff()` function.

Let's start by comparing the `state` values from `tb_notifs_public` dataframe to the `state` values from the `covid_screening_public` dataframe.

```
setdiff(tb_notifs_public$state, covid_screening_public$state)
```

```
## [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? By putting the `tb_notifs_public` dataset first, we are asking R "which values are in `tb_notifs_public` but *not* in `covid_screening_public`?"

We can (and should!) also switch the order of the datasets to check the reverse, asking "which values are in `covid_screening_public` but not in `tb_notifs_public`?" Let's do this and compare the two lists.

```
setdiff(covid_screening_public$state, tb_notifs_public$state)
```

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

As we can see, there are four values in `covid_screening_public` that have spelling errors or are written differently than in `tb_notifs_public`. In this case, the easiest thing would be to clean our `covid_screening_public` data using the `case_when()` function to have our two dataframes match. Let's clean this up and then compare our datasets again.

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

setdiff(tb_notifs_public$state, covid_screening_public_clean$state)
```

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

```
setdiff(covid_screening_public_clean$state, tb_notifs_public$state)
```

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

Great! Now we have no more differences in the region's names.

We can now perform our join without unnecessary data loss:

```
inner_join(tb_notifs_public, covid_screening_public_clean)
```

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

```
## # A tibble: 5 × 3
##   state                     tb_notif_count tb_covid_pos
##   <chr>                              <dbl>        <dbl>
## 1 Andaman & Nicobar Islands            510            0
## 2 Andhra Pradesh                     62075           97
## 3 Arunachal Pradesh                   2722            0
## 4 Assam                              36801           31
## 5 Bihar                              79008           78
```

All 36 rows are retained!

Identifying unmatched values with `antijoin()`

The `anti_join()` function in {dplyr} is a handy alternative to `setdiff()` for identifying discrepancies in key columns before joining two dataframes. It returns all rows from the first dataframe where the key values don't match the second dataframe.

For example, to find unmatched `state` values in `tb_notifs_public` when compared to `covid_screening_public`, you can use:

```
anti_join(tb_notifs_public, covid_screening_public)
```

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

```
## # A tibble: 4 × 2
##   state                                      tb_notif_count
##   <chr>                                               <dbl>
## 1 Arunachal Pradesh                                    2722
## 2 Dadra and Nagar Haveli and Daman and Diu              1294
## 3 Tamil Nadu                                           71896
## 4 Tripura                                               2865
```

And vice versa, to find the `state` values that are in `covid_screening_public` but not in `tb_notifs_public`, you can run:

```
anti_join(covid_screening_public, tb_notifs_public)
```

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

```
## # A tibble: 4 × 2
##   state                                  tb_covid_pos
##   <chr>                                         <dbl>
## 1 ArunachalPradesh                                  0
## 2 Dadra & Nagar Haveli and Daman & Diu              1
## 3 tamil nadu                                      178
## 4 Tri pura                                          2
```

This method returns the entire rows where discrepancies occur, providing more context and potentially making it easier to diagnose and fix the issues.

After identifying these, we can again fix the errors with `mutate()` and proceed with the join.

PRACTICE

Check and fix typos before join

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_modified.csv"))

child_public <- child %>%
  filter(hc_type == "public") %>%
  select(-hc_type)

child_public
```


```
## # A tibble: 5 × 2
##   state                     tb_child_notifs
##   <chr>                               <dbl>
## 1 Andaman & Nicobar Islands              18
## 2 Andhra Pradesh                       1347
## 3 ArunachalPradesh                      256
## 4 Assam                                 992
## 5 Bihar                                4434
```

1. Using `set_diff()` or `anti_join()` compare the key values from the `child_public` dataframe with those from the `tb_notifs_public` dataframe, which was defined previously
2. Make any necessary changes to the `child_public` dataframe to ensure that the values match.
3. Join the two datasets.
4. Identify which two regions have the highest proportion of TB cases in children.

Real Data Example 2: Key Typos and Data Gaps

In a previous example, we saw how key typos—spelling and formatting inconsistencies—can prevent a successful join between datasets. Now, let's delve into a slightly more complex scenario.

We start with the `covid_screening_public` dataset that has 36 entries:

```
covid_screening_public
```

```
## # A tibble: 5 × 2
##   state                     tb_covid_pos
##   <chr>                            <dbl>
## 1 Andaman & Nicobar Islands            0
## 2 Andhra Pradesh                      97
## 3 ArunachalPradesh                     0
## 4 Assam                               31
## 5 Bihar                               78
```

We aim to enhance this dataset with zoning information available in another dataset, `regions`, which contains 32 entries:

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

```
## # A tibble: 5 × 3
##   zonal_council          subdivision_categ…¹ state_UT
##   <chr>                  <chr>               <chr>
## 1 No Zonal Council       Union Territory     Andaman & Nico…
## 2 North Eastern Council  State               Arunachal Prad…
## 3 North Eastern Council  State               Assam
## 4 Eastern Zonal Council  State               Bihar
## 5 Northern Zonal Council Union Territory     Chandigarh
## # ℹ abbreviated name: ¹subdivision_category
```

The `regions` dataset columns include `zonal_council`, `subdivision_category`, and `state_UT`, which correspond to the zonal council designations, category of subdivision, and the names of states or Union Territories, respectively.

To merge this zoning information without losing rows from our original `covid_screening_public` data, we opt for a left join:

```
covid_regions <- left_join(covid_screening_public,
                           regions,
                           by = c("state" = "state_UT"))
covid_regions
```

```
## # A tibble: 5 × 4
##   state                     tb_covid_pos zonal_council
##   <chr>                            <dbl> <chr>
## 1 Andaman & Nicobar Islands            0 No Zonal Council
## 2 Andhra Pradesh                      97 <NA>
## 3 ArunachalPradesh                     0 <NA>
## 4 Assam                               31 North Eastern Coun…
## 5 Bihar                               78 Eastern Zonal Coun…
## # ℹ 1 more variable: subdivision_category <chr>
```

After performing the left join, we can observe that 7 entries lack zoning information:

```
covid_regions %>%
  filter(is.na(zonal_council))
```

```
## # A tibble: 5 × 4
##   state                              tb_covid_pos zonal_council
##   <chr>                                     <dbl> <chr>
## 1 Andhra Pradesh                               97 <NA>
## 2 ArunachalPradesh                              0 <NA>
## 3 Chhattisgarh                                 57 <NA>
## 4 Dadra & Nagar Haveli and Daman…               1 <NA>
## 5 Ladakh                                        0 <NA>
## # ℹ 1 more variable: subdivision_category <chr>
```

To understand why, we investigate the discrepancies using the `anti_join()` function.

First, we check which states are present in the `regions` dataset but not in `covid_screening_public`:

```
anti_join(regions, covid_screening_public, by = c("state_UT" = "state"))
```

```
## # A tibble: 3 × 3
##   zonal_council         subdivision_category state_UT
##   <chr>                 <chr>                <chr>
## 1 North Eastern Council State                Arunachal Prad…
## 2 Western Zonal Council Union Territory      Dadra and Naga…
## 3 North Eastern Council State                Tripura
```

This operation reveals 3 states present in `regions` but not in `covid_screening_public`:

1. Arunachal Pradesh
2. Dadra and Nagar Haveli and Daman and Diu
3. Tripura

Then, we check which states are present in the `covid_screening_public` but not in `regions`:

```
anti_join(covid_screening_public, regions, by = c("state" = "state_UT"))
```

```
## # A tibble: 5 × 2
##   state                                tb_covid_pos
##   <chr>                                       <dbl>
## 1 Andhra Pradesh                                 97
## 2 ArunachalPradesh                                0
## 3 Chhattisgarh                                   57
```

```
## 4 Dadra & Nagar Haveli and Daman & Diu            1
## 5 Ladakh                                          0
```

There are 7 states in `covid_screening_public` that are not matched in the `regions` dataset. Upon closer inspection, we can see that only three of these mismatches are due to key typos:...

The remaining four states—Andhra Pradesh, Chhattisgarh, Ladakh, and Tamil Nadu—are simply absent from the `regions` dataset.

To address the typos, we apply corrections similar to those in the previous example:

```
covid_screening_public_fixed <- covid_screening_public %>%
  mutate(state =
           case_when(state == "ArunachalPradesh" ~ "Arunachal Pradesh",
                     state == "Tri pura" ~ "Tripura",
                     state == "Dadra & Nagar Haveli and Daman & Diu" ~ "Dadra
and Nagar Haveli and Daman and Diu",
                     TRUE ~ state))
```

After applying the fixes, we perform another left join:

```
covid_regions_joined_fixed <- left_join(covid_screening_public_fixed,
                                        regions,
                                        by = c("state" = "state_UT"))
covid_regions_joined_fixed
```

```
## # A tibble: 5 × 4
##   state                      tb_covid_pos zonal_council
##   <chr>                             <dbl> <chr>
## 1 Andaman & Nicobar Islands             0 No Zonal Council
## 2 Andhra Pradesh                       97 <NA>
## 3 Arunachal Pradesh                     0 North Eastern Coun…
## 4 Assam                                31 North Eastern Coun…
## 5 Bihar                                78 Eastern Zonal Coun…
## # ℹ 1 more variable: subdivision_category <chr>
```

A subsequent check confirms that only four entries remain without zoning information:

```
covid_regions_joined_fixed %>%
  filter(is.na(zonal_council))
```

```
## # A tibble: 4 × 4
##   state          tb_covid_pos zonal_council
##   <chr>                 <dbl> <chr>
## 1 Andhra Pradesh           97 <NA>
## 2 Chhattisgarh             57 <NA>
## 3 Ladakh                    0 <NA>
```

```
## 4 tamil nadu               178 <NA>
## # ℹ 1 more variable: subdivision_category <chr>
```

These four regions were not included in the `regions` dataset, so there is no further action we can take at this point.

Through this example, we see the challenge of ensuring that no data is lost during joins, which becomes increasingly complex with larger datasets. To handle such issues, we may employ strategies such as manual data inspection and correction, or fuzzy matching for imperfect string comparisons, using tools like the `{fuzzyjoin}` package in R.

Identifying and correcting typographical errors in large datasets to be joined is a non-trivial task. There is no fully automated method for cleaning such discrepancies, and often, fuzzy matching—joining datasets based on non-exact string matches—may be the practical solution. You can look into the {fuzzyjoin} R package for information on this.

Merging TB Cases with Geographic Data

Run the code bellow to define two datasets.

The first, `top_tb_cases_kids` records the top 20 countries with the highest incidence of tuberculosis (TB) in children for the year 2012:

```
top_tb_cases_kids <- tidyr::who %>%
  filter(year == 2012) %>%
  transmute(country, iso3, tb_cases_smear_0_14 = new_sp_m014 +
new_sp_f014) %>%
  arrange(desc(tb_cases_smear_0_14)) %>%
  head(20)

top_tb_cases_kids
```

```
## # A tibble: 5 × 3
##   country                           iso3
tb_cases_smear_0_14
##   <chr>                             <chr>
<dbl>
## 1 India                             IND
12957
## 2 Pakistan                          PAK
3947
## 3 Democratic Republic of the Congo COD
```

```
3138
## 4 South Africa                    ZAF
2677
## 5 Indonesia                       IDN
1703
```

And `country_regions` lists countries along with their respective regions and continents:

```
country_regions <- countrycode::codelist %>%
  select(country_name = iso.name.en, iso3c, region) %>%
  filter(complete.cases(country_name, region))

country_regions
```

```
## # A tibble: 5 × 3
##   country_name   iso3c region
##   <chr>          <chr> <chr>
## 1 Afghanistan    AFG   South Asia
## 2 Albania        ALB   Europe & Central Asia
## 3 Algeria        DZA   Middle East & North Africa
## 4 American Samoa ASM   East Asia & Pacific
## 5 Andorra        AND   Europe & Central Asia
```

Your task is to augment the TB cases data with the region and continent information without losing any relevant data.

1. Perform a `left_join` of `top_tb_cases_kids` with `country_regions` with the country names as the key. Identify which five countries fail to match correctly.

```
left_join(top_tb_cases_kids, ____________________, by =
__________)
```

2. Using the code below, amend the country names in `top_tb_cases_kids` using `case_when` to rectify mismatches:

```
top_tb_cases_kids_fixed <- top_tb_cases_kids %>%
  mutate(country = case_when(
    country == "Democratic Republic of the Congo" ~ "Congo (the
Democratic Republic of the)",
    country == "Philippines" ~ "Philippines (the)",
    country == "Democratic People's Republic of Korea" ~ "Korea
(the Democratic People's Republic of)",
    country == "United Republic of Tanzania" ~ "Tanzania, the
United Republic of",
    country == "Cote d'Ivoire" ~ "Côte d'Ivoire",
    TRUE ~ country
  ))

top_tb_cases_kids_fixed
```


```
## # A tibble: 20 × 3
##    country                              iso3  tb_cases_smear_…
1
##    <chr>                                <chr>
<dbl>
##  1 India                                IND
12957
##  2 Pakistan                             PAK
3947
##  3 Congo (the Democratic Republic o…    COD
3138
##  4 South Africa                         ZAF
2677
##  5 Indonesia                            IDN
1703
##  6 Nigeria                              NGA
1187
##  7 China                                CHN
1091
##  8 Philippines (the)                    PHL
1049
##  9 Kenya                                KEN
996
## 10 Angola                               AGO
982
## 11 Bangladesh                           BGD
966
## 12 Uganda                               UGA
636
## 13 Afghanistan                          AFG
588
## 14 Brazil                               BRA
580
## 15 Korea (the Democratic People's R…    PRK
520
## 16 Tanzania, the United Republic of     TZA
493
```

```
## 18 Madagascar                          MDG
419
## 19 Côte d'Ivoire                       CIV
367
## 20 Myanmar                             MMR
338
## # ℹ abbreviated name: ¹tb_cases_smear_0_14
```

Now attempt the join again using the revised dataset.


```
left_join(top_tb_cases_kids_fixed, ____________________, by
= __________)
```

3. Try another `left_join`, but this time use the three-letter ISO code as the key. Do those initial five countries now align properly?

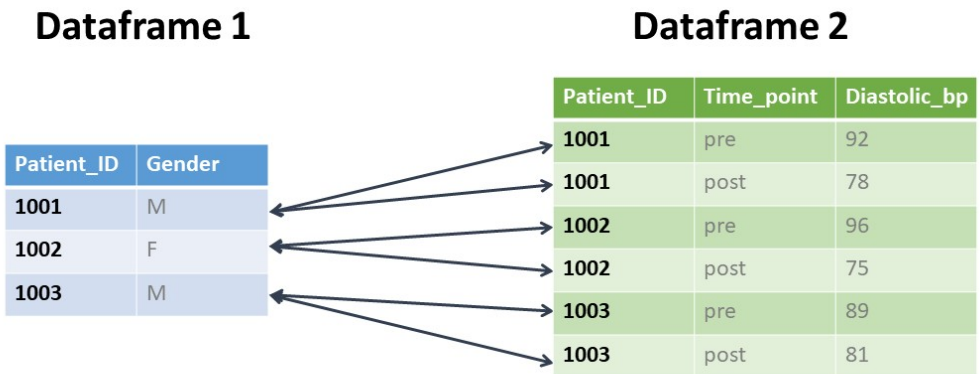
```
left_join(top_tb_cases_kids, ____________________, by =
__________)
```

4. What is the advantage of utilizing ISO codes when recording and storing country information?

One-to-many relationships

So far, we have primarily looked at one-to-one joins, where an observation in one dataframe corresponded to only 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:



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()` with `demographic` as the dataset to the left of the call:

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

```
## 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 2023-06-10 Positive
## 3 Bob        32 2023-08-10 Positive
## 4 Charlie    45 <NA>       <NA>
```

What´s happened above? 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:

PRACTICE

(in RMD)

Merging One-to-Many Patient Records

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 `tb_notifs` dataset. Here is what it looks like:

```
tb_notifs
```

```
## # A tibble: 5 × 3
##   state                     hc_type tb_notif_count
##   <chr>                     <chr>            <dbl>
## 1 Andaman & Nicobar Islands public             510
## 2 Andaman & Nicobar Islands private             24
## 3 Andhra Pradesh            public            62075
## 4 Andhra Pradesh            private           30112
## 5 Arunachal Pradesh         public             2722
```

Note that this dataset is a long dataset, with two rows per state, one for notifications from public health facilities in the state and one for private health facilities.

The second dataset is an Indian regions dataset:

```
full_regions <- read_csv(here("data/region_data_india_full.csv"))
```

```
## Rows: 36 Columns: 3
## ── Column specification
────────────────────────────────────────────────────────
## Delimiter: ","
## chr (3): zonal_council, subdivision_category, state_UT
##
## 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.
```

```
full_regions
```

```
## # A tibble: 5 × 3
##   zonal_council          subdivision_categ…¹ state_UT
##   <chr>                  <chr>               <chr>
## 1 No Zonal Council       Union Territory     Andaman & Nico…
## 2 North Eastern Council  State               Arunachal Prad…
## 3 North Eastern Council  State               Assam
## 4 Eastern Zonal Council  State               Bihar
```

```
## 5 Northern Zonal Council Union Territory     Chandigarh
## # ℹ abbreviated name: ¹subdivision_category
```

Let's try joining the datasets:

```
notif_regions <- tb_notifs %>%
  left_join(regions, by = c("state" = "state_UT"))
notif_regions
```

```
## # A tibble: 5 × 5
##   state                hc_type tb_notif_count zonal_council
##   <chr>                <chr>            <dbl> <chr>
## 1 Andaman & Nicobar … public               510 No Zonal Counc…
## 2 Andaman & Nicobar … private               24 No Zonal Counc…
## 3 Andhra Pradesh       public            62075 <NA>
## 4 Andhra Pradesh       private           30112 <NA>
## 5 Arunachal Pradesh    public             2722 North Eastern …
## # ℹ 1 more variable: subdivision_category <chr>
```

As expected, the data from the `regions` dataframe was duplicated for every matching value of the `tb_notifs` dataframe.

Joining `child` with `regions`

Join `child` with `region`

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

```
joined_dat <- left_join(_____________, _________________, by =
_____________)
```

Then work out which Zonal council has the highest number of states/Union territories

```
joined_dat %>%
  count(_________________)
```

Multiple key columns

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

```
blood_pressure <- tribble(
  ~name,    ~time_point,  ~systolic,
  "Alice",   "pre",         139,
  "Alice",   "post",        121,
  "Bob",     "pre",         137,
  "Bob",     "post",        128,
  "Charlie", "pre",         137,
  "Charlie", "post",        130 )
```

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. High creatinine levels may be a side effect of the drug being tested.

```
kidney <- tribble(
  ~name,    ~time_point,  ~creatinine,
  "Alice",   "pre",         0.9,
  "Alice",   "post",        1.3,
  "Bob",     "pre",         0.7,
  "Bob",     "post",        0.8,
  "Charlie", "pre",         0.6,
  "Charlie", "post",        1.4
)
```

We want to join the two datasets so that each patient has two rows, one row for their levels before the drug and one row for their levels after. 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")
bp_kidney_dups
```

```
## # A tibble: 5 × 5
##   name  time_point.x systolic time_point.y creatinine
##   <chr> <chr>           <dbl> <chr>             <dbl>
## 1 Alice pre               139 pre                 0.9
## 2 Alice pre               139 post                1.3
## 3 Alice post              121 pre                 0.9
## 4 Alice post              121 post                1.3
## 5 Bob   pre               137 pre                 0.7
```

As we can see, this isn't what we wanted at all! We end up with duplicated rows. Now we have FOUR rows for each person. And 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. 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
```

```
## # A tibble: 5 × 4
##   name    time_point systolic creatinine
##   <chr>   <chr>         <dbl>      <dbl>
## 1 Alice   pre             139        0.9
## 2 Alice   post            121        1.3
## 3 Bob     pre             137        0.7
## 4 Bob     post            128        0.8
## 5 Charlie pre             137        0.6
```

Note that specifying `by = c("name", "time_point")` is different from using a named vector in the form of `by = c("keya" = "keyb")`. The former matches the columns by name across both datasets, while the latter is used to match columns with different names between two datasets.

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

```
tb_notifs
```

```
## # A tibble: 5 × 3
##   state                     hc_type tb_notif_count
##   <chr>                     <chr>            <dbl>
## 1 Andaman & Nicobar Islands public             510
## 2 Andaman & Nicobar Islands private             24
## 3 Andhra Pradesh            public           62075
## 4 Andhra Pradesh            private          30112
## 5 Arunachal Pradesh         public            2722
```

```
covid_screening
```

```
## # A tibble: 5 × 3
##   state                     hc_type tb_covid_pos
##   <chr>                     <chr>          <dbl>
## 1 Andaman & Nicobar Islands public             0
## 2 Andaman & Nicobar Islands private            0
## 3 Andhra Pradesh            public            97
## 4 Andhra Pradesh            private           17
## 5 ArunachalPradesh          public             0
```

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` 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 <- tb_notifs %>%
  left_join(covid_screening, by=c("state", "hc_type"))
notif_covid
```

```
## # A tibble: 5 × 4
##   state                  hc_type tb_notif_count tb_covid_pos
##   <chr>                  <chr>            <dbl>        <dbl>
## 1 Andaman & Nicobar Isl… public             510            0
## 2 Andaman & Nicobar Isl… private             24            0
## 3 Andhra Pradesh         public            62075           97
## 4 Andhra Pradesh         private           30112           17
## 5 Arunachal Pradesh      public             2722           NA
```

Great, that's exactly what we wanted!

Joining three datasets, including one-to-many

Join the following three datasets: `notif_covid`, `child` and `regions`, ensuring that no data is lost from any of the datasets.

Wrap Up!

In this lesson, we delved into the intricacies of data cleaning before a join, focusing on how to detect and correct mismatches or inconsistencies in key columns. We also highlighted the impact of one-to-many relationships in joining dataframes, showing how data from the "one" side is duplicated for each matching row of the "many" side. Finally, we demonstrated how to join dataframes using multiple key columns.

As we conclude this lesson, we hope that you have gained a deeper understanding of the importance and utility of joining dataframes in R.

Answer Key

Q: Inner Join countries

```
df2_fixed <- df2 %>%
  mutate(Country =
           case_match(Country,
                              "India " ~ "India", # Remove blank space at end
                              "indonesia" ~ "Indonesia", # Capitalize
                              "Philipines" ~ "Philippines", # Fix spelling
                              .default=Country))

inner_join(df1, df2_fixed)
```

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

```
## # A tibble: 3 × 4
##   Country     Capital   Population Life_Expectancy
##   <chr>       <chr>          <dbl>           <dbl>
## 1 India       New Delhi 1393000000            69.7
## 2 Indonesia   Jakarta    273500000            71.7
## 3 Philippines Manila     113000000            72.7
```

Q: Check and fix typos before join

1.

```
# setdiff()
setdiff(child_public$state, tb_notifs_public$state)
```

```
## [1] "ArunachalPradesh"  "Jammu and Kashmir" "kerala"
## [4] "Pondicherry"
```

```
setdiff(tb_notifs_public$state, child_public$state)
```

```
## [1] "Arunachal Pradesh" "Jammu & Kashmir"   "Kerala"
## [4] "Puducherry"
```

```
# antijoin
anti_join(child_public, tb_notifs_public)
```

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

```
## # A tibble: 4 × 2
##   state             tb_child_notifs
##   <chr>                       <dbl>
## 1 ArunachalPradesh              256
## 2 Jammu and Kashmir             511
## 3 kerala                        480
## 4 Pondicherry                   101
```

```
anti_join(tb_notifs_public, child_public)
```

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

```
## # A tibble: 4 × 2
##   state             tb_notif_count
##   <chr>                      <dbl>
## 1 Arunachal Pradesh           2722
## 2 Jammu & Kashmir            10022
## 3 Kerala                     16766
## 4 Puducherry                  3732
```

2.

```
child_public_fixed <- child_public %>%
  mutate(state =
           case_when(state == "ArunachalPradesh" ~ "Arunachal Pradesh",
                     state == "Jammu and Kashmir" ~ "Jammu & Kashmir",
                     state == "kerala" ~ "Kerala",
                     state == "Pondicherry" ~ "Puducherry",
                     TRUE ~ state))
```

3.

```
child_tb_public <- child_public_fixed %>%
  inner_join(tb_notifs_public)
```

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

4.

```
child_tb_public %>%
  mutate(tb_child_prop = tb_child_notifs/tb_notif_count) %>%
  arrange(-tb_child_prop)
```

```
## # A tibble: 5 × 4
##   state             tb_child_notifs tb_notif_count
##   <chr>                       <dbl>          <dbl>
## 1 Delhi                        7867          76966
## 2 Arunachal Pradesh             256           2722
## 3 Lakshadweep                     1             11
## 4 Chandigarh                    496           5664
## 5 Mizoram                       123           1697
## # ℹ 1 more variable: tb_child_prop <dbl>
```

Q: Merging TB Cases with Geographic Data

1.

```
left_join(top_tb_cases_kids, country_regions, by =
c("country"="country_name"))
```

```
## # A tibble: 5 × 5
##   country                     iso3  tb_cases_smear_…¹ iso3c
##   <chr>                       <chr>             <dbl> <chr>
## 1 India                       IND               12957 IND
## 2 Pakistan                    PAK                3947 PAK
## 3 Democratic Republic of the … COD               3138 <NA>
## 4 South Africa                ZAF                2677 ZAF
## 5 Indonesia                   IDN                1703 IDN
```

```
## # ℹ abbreviated name: ¹tb_cases_smear_0_14
## # ℹ 1 more variable: region <chr>
```

```
setdiff(top_tb_cases_kids$country, country_regions$country_name)
```

```
## [1] "Democratic Republic of the Congo"
## [2] "Philippines"
## [3] "Democratic People's Republic of Korea"
## [4] "United Republic of Tanzania"
## [5] "Cote d'Ivoire"
```

2.

```
left_join(top_tb_cases_kids_fixed, country_regions, by =
c("country"="country_name"))
```

```
## # A tibble: 5 × 5
##   country                     iso3  tb_cases_smear_…¹ iso3c
##   <chr>                       <chr>             <dbl> <chr>
## 1 India                       IND               12957 IND
## 2 Pakistan                    PAK                3947 PAK
## 3 Congo (the Democratic Repub… COD               3138 COD
## 4 South Africa                ZAF                2677 ZAF
## 5 Indonesia                   IDN                1703 IDN
## # ℹ abbreviated name: ¹tb_cases_smear_0_14
## # ℹ 1 more variable: region <chr>
```

3.

```
left_join(top_tb_cases_kids, country_regions, by = c("iso3" = "iso3c"))
```

```
## # A tibble: 5 × 5
##   country            iso3  tb_cases_smear_…¹ country_name
##   <chr>              <chr>             <dbl> <chr>
## 1 India              IND               12957 India
## 2 Pakistan           PAK                3947 Pakistan
## 3 Democratic Republ… COD                3138 Congo (the Dem…
## 4 South Africa       ZAF                2677 South Africa
## 5 Indonesia          IDN                1703 Indonesia
## # ℹ abbreviated name: ¹tb_cases_smear_0_14
## # ℹ 1 more variable: region <chr>
```

4.

ISO codes are standardized - there in only one way of writing them. This makes it useful for joining.

Q: Merging One-to-Many Patient Records

```
# 5 rows in the final dataframe
patient_info %>%
  left_join(conditions)
```

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

```
## # A tibble: 5 × 4
##   patient_id name    age disease
##        <dbl> <chr> <dbl> <chr>
## 1          1 Liam     32 Diabetes
## 2          1 Liam     32 Hypertension
## 3          2 Manny    28 Asthma
## 4          3 Nico     40 High Cholesterol
## 5          3 Nico     40 Arthritis
```

Q: Joining `child` with `regions`

```
child_fixed <- child %>%
  mutate(state =
           case_when(state == "ArunachalPradesh" ~ "Arunachal Pradesh",
                     state == "Jammu and Kashmir" ~ "Jammu & Kashmir",
                     state == "kerala" ~ "Kerala",
                     state == "Pondicherry" ~ "Puducherry",
                     TRUE ~ state))

joined_dat <- left_join(child_fixed, regions, by = c("state"="state_UT"))
joined_dat
```

```
## # A tibble: 5 × 5
##   state              hc_type tb_child_notifs zonal_council
##   <chr>              <chr>             <dbl> <chr>
## 1 Andaman & Nicobar… public               18 No Zonal Counc…
## 2 Andaman & Nicobar… private               1 No Zonal Counc…
## 3 Andhra Pradesh     public             1347 <NA>
## 4 Andhra Pradesh     private            1333 <NA>
## 5 Arunachal Pradesh  public              256 North Eastern …
## # ℹ 1 more variable: subdivision_category <chr>
```

```
joined_dat %>%
  count(zonal_council)
```

```
## # A tibble: 5 × 2
##   zonal_council              n
```

```
##   <chr>                  <int>
## 1 Central Zonal Council      6
## 2 Eastern Zonal Council      8
## 3 No Zonal Council           4
## 4 North Eastern Council     16
## 5 Northern Zonal Council    14
```

Q: Joining three datasets, including one-to-many

```
join_1 <- notif_covid %>%
  full_join(child_fixed)
```

```
## Joining with `by = join_by(state, hc_type)`
```

```
final_join <- join_1 %>%
  full_join(regions, by=c("state" ="state_UT"))

final_join
```

```
## # A tibble: 5 × 7
##   state                  hc_type tb_notif_count tb_covid_pos
##   <chr>                  <chr>            <dbl>        <dbl>
## 1 Andaman & Nicobar Isl… public             510            0
## 2 Andaman & Nicobar Isl… private             24            0
## 3 Andhra Pradesh         public           62075           97
## 4 Andhra Pradesh         private          30112           17
## 5 Arunachal Pradesh      public            2722           NA
## # ℹ 3 more variables: tb_child_notifs <dbl>,
## #   zonal_council <chr>, subdivision_category <chr>
```

Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

CAMILLE BEATRICE VALERA

Project Manager and Scientific Collaborator, The GRAPH Network

Lesson Notes: Intro to Functions and Conditionals

Intro

The two main components of the R language are objects and functions. Objects are the data structures that we use to store information, and functions are the tools that we use to manipulate these objects. Quoting John Chambers, who played a key role in the development of the R language, everything that "exists" in a R environment is an object, and everything that "happens" is a function.

So far you have mostly used functions written by others. In this lesson, you will learn how to write your own functions in R.

Writing functions allows you to automate repetitive tasks, improve efficiency and reduce errors in your code.

In this lesson, we will learn the fundamentals of functions with simple examples. Then in a future lesson, we will write more complex functions that can automate large parts of your data analysis workflow.

Learning objectives

By the end of this lesson, you will be able to:

1. Create and use your own functions in R.
2. Design function arguments and set default values.
3. Use conditional logic like `if`, `else if`, and `else` within functions.
4. Check and validate function arguments to prevent errors.
5. Manage function scope and understand local vs. global variables.
6. Handle vectorized data in functions.

7. Organize and store your custom functions for easy reuse.

Packages

Run the following code to install and load the packages needed for this lesson:

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

Basics of a Function

Let's start by creating a very simple function. Consider the following function that converts pounds (a unit of weight) to kilograms (another unit of weight):

```
pounds_to_kg <- function(pounds) {
  return(pounds * 0.4536)
}
```

If you execute this code, you will create a function named `pounds_to_kg`, which can be used directly in a script or in the console:

```
pounds_to_kg(150)
```

```
## [1] 68.04
```

Let's break down the structure of this first function step by step.

First, a function is created using the statement `function`, followed by a pair of parentheses and a pair of braces.

```
function() {

}
```

Inside the parentheses, we indicate the **arguments** of the function. Our function only takes one argument, which we have decided to name `pounds`. This is the value that we want to convert from pounds to kilograms.

```
function(pounds) {

}
```

Of course, we could have named this argument anything we wanted.

The next element, inside the braces, is the **body** of the function. This is where we write the code that we want to execute when the function is called.

```
function(pounds) {
  pounds * 0.4536
}
```

Now we want that our function return what is calculated inside its body. This is achieved via the instruction `return`.

```
function(pounds) {
  return(pounds * 0.4536)
}
```

Sometimes you can skip the `return` statement, and just write the expression to return at the end of the function, as R will automatically return the last expression evaluated in the function:

```
function(pounds) {
  pounds * 0.4536 # R will automatically return this expression
}
```

However, it is good practice to always include the `return` statement, as it makes the code more readable.

We may also want to first assign the result to an object and then return it:

```
function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

This is a bit more wordy, but it makes the function clearer.

Finally, in order for our function to be called and used, we need to give it a name. This is the same as storing a value in an object. Here we store it in an object named `pounds_to_kg`.

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

With our function, we have therefore created a new object in our environment called `pounds_to_kg`, of class `function`.

```
class(pounds_to_kg)
```

```
## [1] "function"
```

We can now use it like this with a named argument:

```
pounds_to_kg(pounds = 150)
```

```
## [1] 68.04
```

Or without a named argument:

```
pounds_to_kg(150)
```

```
## [1] 68.04
```

The function can also be used with a vector of values:

```
my_vec <- c(150, 200, 250)
pounds_to_kg(my_vec)
```

```
## [1]  68.04  90.72 113.40
```

And that's it! You have just created your first function in R.

You can view the source code of any function by typing its name without parentheses:

```
pounds_to_kg
```

```
## function(pounds) {
##   kg <- pounds * 0.4536
##   return(kg)
## }
## <bytecode: 0x14bc556c0>
```

To view this as an R script, you can use the `View` function:

```
View(pounds_to_kg)
```

This will open a new tab in RStudio with the source code of the function.

This method works for any function, not just the ones you create. For an example of what a *real* function in the wild looks like, try viewing the source code of the `reactable` function from the `reactable` package:

```
View(reactable)
```

Age Months Function

Create a simple function called `years_to_months` that transforms age in years to age in months.

Try it out with `years_to_months(12)`

```
# Your code here
years_to_months <- ...
```

Let's now write a slightly more complex function, to get some extra practice. The function we will write will convert a temperature in Fahrenheit (used in the US) to a temperature in Celsius. The formula for this conversion is:

$$C = \frac{5}{9} \times (F - 32)$$

And here is the function:

```
fahrenheit_to_celsius <- function(fahrenheit) {
  celsius <- (5 / 9) * (fahrenheit - 32)
  return(celsius)
}

fahrenheit_to_celsius(32) # freezing point of water. Should be 0
```

```
## [1] 0
```

Let's test out the function on a column of the `airquality` dataset, one of the built-in datasets in R:

```
airquality %>%
  select(Temp) %>%
  mutate(Temp = fahrenheit_to_celsius(Temp)) %>%
  head()
```

```
##       Temp
## 1 19.44444
## 2 22.22222
## 3 23.33333
## 4 16.66667
```

```
## 5 13.33333
## 6 18.88889
```

Great!

When to Write a Function in R

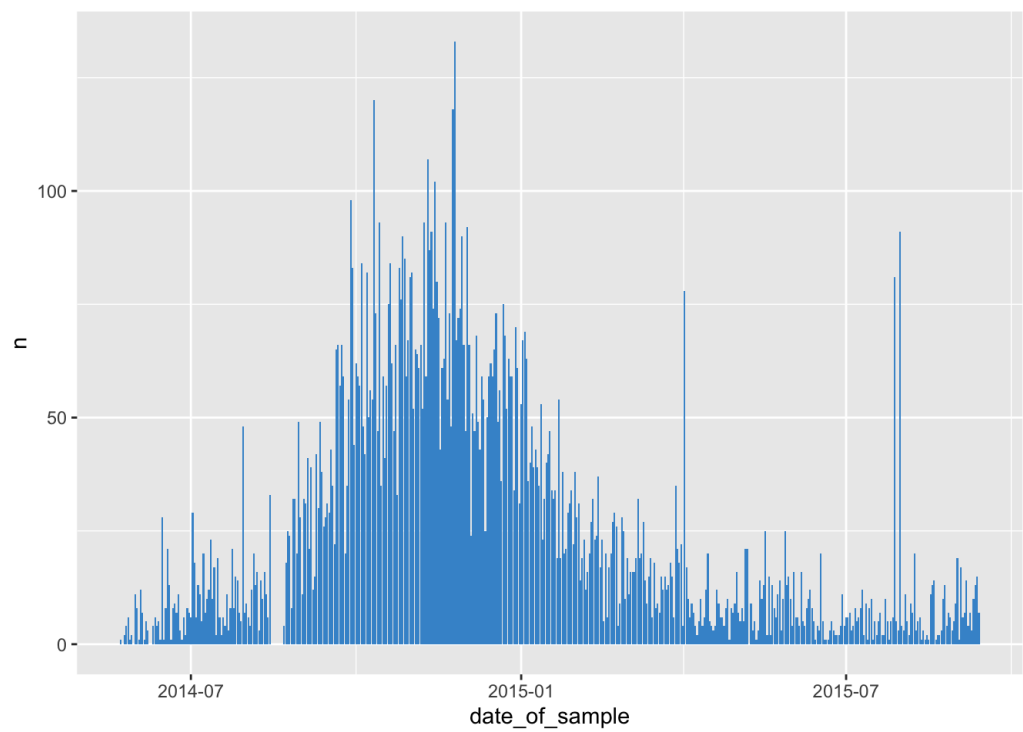
In R, many operations can be completed using existing functions or by combining a few. However, there are occasions when crafting your own function is advantageous:

- **Reusability**: If you find yourself repeatedly writing the same code, it may be beneficial to encapsulate it in a function. For instance, if you frequently convert temperatures from Fahrenheit to Celsius, creating a `fahrenheit_to_celsius` function would streamline your code and improve efficiency.
- **Readability**: Functions can improve code readability, particularly when they have descriptive names. With straightforward functions like `fahrenheit_to_celsius`, the benefits are not so obvious. However, as functions become more complex, descriptive names become increasingly important.
- **Sharing**: Functions make it easier to share code. They can be distributed either as part of a package or as standalone scripts. Although creating a package is complex and beyond the scope of this course, sharing simpler functions is quite straightforward. We'll talk about options for this later in the lesson.

Data Frame and Plot functions

The most useful functions you'll likely write will involve data frame and plot manipulation. Here's a function that takes a linelist of cases and returns a plot of the epidemic curve:

```r
# Function to plot an epidemic curve
plot_epidemic_curve <- function(data, date_column) {
  data %>%
    count({{ date_column }}) %>%
    complete({{ date_column }} := seq(min({{ date_column }}),
                                      max({{ date_column }}),
by="day")) %>%
    ggplot(aes(x = {{ date_column }}, y = n)) +
    geom_col(fill = "#4395D1")
}

# Example usage
plot_epidemic_curve(outbreaks::ebola_sierraleone_2014,
date_of_sample)
```



SIDE NOTE

This lesson will delve into more complex functions later. For now, we'll focus on the basics of function writing, using simple vector manipulation functions as examples.

Celsius to Fahrenheit Function

Create a function named `celsius_to_fahrenheit` that converts temperature from Celsius to Fahrenheit. Here is the formula for this conversion:

$$Fahrenheit = Celsius * 1.8 + 32$$

```
# Your code here
celsius_to_fahrenheit <- ...
```

Then test your function on the `temp` column of the built-in `beaver1` dataset in R:

```
beaver1 %>%
  select(temp) %>%
  head()
```

Functions with Multiple Arguments

Most functions take multiple arguments rather than just one. Let's look at an example of a function that takes three arguments:

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}

calculate_calories(carb_grams = 50, protein_grams = 25, fat_grams = 10)
```

```
## [1] 390
```

The `calculate_calories` function computes the total calories based on the grams of carbohydrates, protein, and fat. Carbohydrates and proteins are estimated to be 4 calories per gram, while fat is estimated to be 9 calories per gram.

If you attempt to use the function without supplying all the arguments, it will yield an error.

```
calculate_calories(carb_grams = 50, protein_grams = 25)
```

```
Error in calculate_calories(50, 25) : argument "fat_grams" is missing, with
no default
```

You can define **default values** for your function's arguments. If an argument is **called** without a **value assigned to it**, then this argument assumes its default value.

Here's an example where `fat_grams` is given a default value of `0`.

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams = 0) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}

calculate_calories(50, 25)
```

```
## [1] 300
```

In this revised version, `carb_grams` and `protein_grams` are mandatory arguments, but we could make all arguments optional by giving them all default values:

```
calculate_calories <- function(carb_grams = 0, protein_grams = 0, fat_grams =
0) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Now, we can call the function with no arguments:

```
calculate_calories()
```

```
## [1] 0
```

We can also call it with some arguments:

```
calculate_calories(carb_grams= 50, protein_grams = 25)
```

```
## [1] 300
```

And it works as expected.

PRACTICE

BMI Function

Create a function named `calc_bmi` that calculates the Body Mass Index (BMI) for one or more individuals. Keep in mind that BMI is calculated as weight in kg divided by the square of the height in meters. Therefore, this function requires two mandatory arguments: weight and height.


```
# Your code here
calc_bmi <- ...
```

Then, apply your function to the `medicaldata::smartpill` dataset to calculate the BMI for each person:

```
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(BMI = calc_bmi(Weight, Height))
```

Passing Arguments to Internal Functions

When writing functions in R, you might need to use existing functions within your custom function. For instance, consider our familiar function that converts pounds to kilograms:

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

It could be useful to have the capability to round the output to specified decimal places without calling a separate function.

For this, we can incorporate the `round` function directly into our custom function. The `round` function has two arguments: `x`, the number to round, and `digits`, the number of decimal places to round to:

```
round(x = 1.2345, digits = 2)
```

```
## [1] 1.23
```

Now, we can add an argument to our function called `round_to` that will be passed to the `digits` argument of the `round` function.

```r
pounds_to_kg <- function(pounds, round_to = 2) {
  kg <- pounds * 0.4536
  kg_rounded <- round(x = kg, digits = round_to)
  return(kg_rounded)
}
```

In the function above, we've added an argument called `round_to` with a default value of 3.

Now when you pass a value to the `round_to` argument, it will be used by the `round` function.

```r
pounds_to_kg(10) # with no argument passed to round_to, the default value of 2 is used
```

```
## [1] 4.54
```

```r
pounds_to_kg(10, round_to = 1)
```

```
## [1] 4.5
```

```r
pounds_to_kg(10, round_to = 3)
```

```
## [1] 4.536
```

The Ellipsis Argument, `...`

Sometimes, there are many arguments to pass to an internal function. For instance, consider the `format()` function in R, which has many arguments:

```r
format(x = 12364.2345,
       big.mark = " ", # thousands separator
       decimal.mark = ",", # decimal point. French style!
       nsmall = 2, # number of digits to the right of the decimal point
       scientific = FALSE # use scientific notation?
       )
```

```
## [1] "12 364,23"
```

You can see all the arguments by typing `?format` in the console.

If we want our function to have the ability to pass all these arguments to the `format` function, we'll use the ellipsis argument, `...`. Here is an example:

```r
pounds_to_kg <- function(pounds, ...) {
  kg <- pounds * 0.4536
  kg_formatted <- format(x = kg, ...)
  return(kg_formatted)
}
```

Now, when we pass any arguments to the `pounds_to_kg` function, they'll get passed to the `format` function, even though we didn't explicitly define them in our function.

```r
pounds_to_kg(10000.234)
```

```
## [1] "4536.106"
```

```r
pounds_to_kg(10000.234, big.mark = " ", decimal.mark = ",")
```

```
## [1] "4 536,106"
```

Great!

Practice with the ... Argument

Consider our `calculate_calories()` function.

```r
calculate_calories <- function(carb_grams, protein_grams,
fat_grams) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams
* 9)
  return(result)
}
```

Enhance this function to accept formatting arguments using the `...` mechanism.

Understanding Scope in R

Scope refers to the visibility of variables and objects within different parts of your R code. It is important to understand scope when writing functions.

Objects created within a function have **local scope** within that function (as opposed to **global scope**) and are not accessible outside of the function. Let's illustrate this with the `pounds_to_kg` function:

Imagine you want to convert a weight in pounds to kilograms and you write a function to do this:

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
}
```

You may be tempted to try to access the `kg` variable outside of the function, but you'll get an error:

```
pounds_to_kg(50)
kg
```

```
Error: object 'kg' not found
```

This is because `kg` is a local variable within the `pounds_to_kg` function and is not accessible in the global environment.

To use a value generated within a function, we must ensure it's returned by the function:

```
pounds_to_kg <- function(pounds) {
  kg <- pounds * 0.4536
  return(kg)
}
```

And then we can store the function's result in a global variable:

```
kg <- pounds_to_kg(50)
```

Now, we can access the `kg` object:

```
kg
```

```
## [1] 22.68
```

The Superassignment Operator, `<<-`

While we said that objects created within a function have local scope, it actually **is** possible to create global variables from within a function using the special <<- operator.

Consider the following example:


```
test <- function() {
  new_obj <<- 15
}
```

Now, if we run the function, `new_obj` will be created in the global environment, with a value of 15:

```
test()
new_obj
```

```
## [1] 15
```

While this is technically possible, it's generally not recommended (especially for non-experts) due to potential side effects and maintenance challenges.

Intro to Conditionals: `if`, `else if` and `else`

Conditionals, which are used to control the flow of code execution, are an essential part of programming, especially when writing functions. In R, conditionals are implemented using `if`, `else`, and `else if` statements.

When we employ `if`, we're specifying that we want certain code to run only if a specific condition is true.

Below is the structure of an `if` statement:

```
if (condition) {
  # code to run if condition is true
}
```

Notice that it looks similar to the structure of a function.

Now, let's see an `if` statement in action, for converting a temperature from Celsius to Fahrenheit:

```
celsius <- 20
convert_to <- "fahrenheit"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
}
```

```
## [1] 68
```

In this snippet, if the variable `convert_to` is equal to `"fahrenheit"`, the conversion is carried out and the result is printed.

Now, let's see what happens when `convert_to` is set to a value other than `"fahrenheit"`:

```
convert_to <- "kelvin"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
}
```

In this case, nothing is printed because the condition is not satisfied.

To address situations when `convert_to` is not `"fahrenheit"`, we can add an `else` clause:

```
convert_to <- "kelvin"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else {
  print(celsius)
}
```

```
## [1] 20
```

Here, if the `convert_to` variable doesn't match `"fahrenheit"`, the code in the `else` block runs, printing the original Celsius value.

```r
convert_to <- "kelvin"

if (convert_to == "fahrenheit") {
  fahrenheit <- (celsius * 9/5) + 32
  print(fahrenheit)
} else if (convert_to == "kelvin") {
  kelvin_temp <- celsius + 273.15
  print(kelvin_temp)
} else {
  print(celsius)
}
```

```
## [1] 293.15
```

Here, the code handles three possibilities:

- Convert to Fahrenheit if convert_to is "fahrenheit".
- if convert_to is NOT "fahrenheit", check if it's "kelvin". If so, convert to Kelvin.
- If convert_to is neither "fahrenheit" nor "kelvin", print the original Celsius value.

Note that you can have as many `else if` statements as you need, while you can only have one `else` statement attached to an `if` statement.

Finally, we can encapsulate this logic into a function. We will assign the output within each conditional to a variable called `out` and return `out` at the end of the function:

```r
celsius_convert <- function(celsius, convert_to) {
  if (convert_to == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_to == "kelvin") {
    out <- celsius + 273.15
  } else {
    out <- celsius
  }
  return(out)
}
```

Let's test the function:

```r
celsius_convert(20, "fahrenheit")
```

```
## [1] 68
```

```r
celsius_convert(20, "kelvin")
```

```
## [1] 293.15
```

One problem with the current function is that if there is an invalid value of `convert_to`, we do not get any informative statements about this:

```
celsius_convert(20, "celsius")
```

```
## [1] 20
```

```
celsius_convert(20, "foo")
```

```
## [1] 20
```

This is a common problem with functions that use conditionals. We will discuss how to handle this in the next section.

Debugging a Function with Conditional Logic

A function named `check_negatives` is designed to analyze a vector of numbers in R and print a message indicating whether the vector contains any negative numbers. However, the function currently has syntax errors.


```
check_negatives <- function(numbers) {
  x <- numbers

  if any(x < 0) {
    print("x contains negative numbers")
  else {
    print("x does not contain negative numbers")
}
```

Identify and correct the syntax errors in the `check_negatives` function. After correcting the function, test it with the following vectors to ensure it works correctly: 1. `c(8, 3, -2, 5)` 2. `c(10, 20, 30, 40)`

Argument Checking with Conditionals

When writing functions in R, it is often useful to ensure that the inputs provided are sensible and within the expected domain. Without proper checks, a function might return

incorrect results or fail silently, which can be confusing and time-consuming to debug. This is where argument checking comes in.

Consider the following scenario with our temperature conversion function `celsius_convert()`, which converts a temperature from Celsius to

```
celsius_convert(30, "centigrade")
```

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

In this case, the user is trying to convert a temperature from Kelvin to "centigrade". But our function fails silently, returning the original Celsius value instead of an error message. This is because the `convert_to` argument is not checked for validity.

To enhance our function, we can introduce argument checking to validate `convert_from` and `convert_to`. The `stop()` function in R allows us to terminate the execution of a function and print an error message. Here is how we can use `stop()` to check for valid values of `convert_from` and `convert_to`:

```
# Testing out stop() outside of a function, to later integrate into
conv_temp()
convert_to <- "bad scale"

if (!(convert_to %in% c("fahrenheit", "kelvin"))) {
  stop("convert_to must be fahrenheit or kelvin")
}
```

```
Error: convert_to must be celsius, fahrenheit, or kelvin
```

Now let's integrate this into our `celsius_convert()` function:

```
celsius_convert <- function(celsius, convert_to) {
  if (!(convert_to %in% c("fahrenheit", "kelvin"))) {
    stop("convert_to must be fahrenheit, or kelvin")
  }

  if (convert_to == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_to == "kelvin") {
    out <- celsius + 273.15
  } else {
    out <- celsius
  }
  return(out)
}
```

Note that n this updated function, there is no longer a need for the `else` statement, since the `stop()` function will terminate the function if `convert_to` is not one of the three

valid values. So we can simplify the function as follows:

```
celsius_convert <- function(celsius, convert_to) {
  if (!(convert_to %in% c("fahrenheit", "kelvin"))) {
    stop("convert_to must be fahrenheit or kelvin")
  }

  if (convert_to == "fahrenheit") {
    out <- (celsius * 9/5) + 32
  } else if (convert_to == "kelvin") {
    out <- celsius + 273.15
  }
  return(out)
}
```

Now, if we run the original problematic command:

```
celsius_convert(30, "centigrade")
```

The function will immediately stop and provide a clear error message, indicating that "centigrade" is not a recognized temperature scale.

While argument checking enhances function reliability, overuse can bog down performance and complicate the code. Over time, by examining other people's code and through experience, you'll develop a good sense of how much checking is just right—balancing thoroughness with efficiency and clarity. For now, note that it is usually good to err on the side of more checking.

Argument Checking Practice

Consider the `calculate_calories` function we wrote earlier:

```
calculate_calories <- function(carb_grams = 0, protein_grams =
0, fat_grams = 0) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams
* 9)
  return(result)
}
```

Write a function called `calculate_calories2()` that is the same as `calculate_calories()` except that it checks if the `carb_grams`, `protein_grams`, and `fat_grams` arguments are numeric. If any of them

are not numeric, the function should print an error message using the `stop()` function.


```
calculate_calories2 <- function(carb_grams = 0, protein_grams =
0, fat_grams = 0) {

  # your code here

  return(result)
}
```

Vectorized Conditionals

In previous examples, we conditioned on a single value, such as the `convert_to` argument. Now, let's explore how to construct conditionals based on a vector of values. Such cases require special handling because the `if` statement is not vectorized.

For instance, if we wanted to write a function to classify temperature readings into hypothermia, normal, or fever, you might think of using an if-else construct like this:

```
classify_temp <- function(temp) {
  if (temp < 35) {
    print("hypothermia")
  } else if (temp >= 35 & temp <= 37) {
    print("normal")
  } else if (temp > 37) {
    print("fever")
  }
}

# Side note: These fever temperatures are not exactly correct, but we keep it
simple for the sake of the example.
```

This works on a single value:

```
classify_temp(36)
```

```
## [1] "normal"
```

But it will not work as intended on a vector, because the `if` statement is not vectorized and only evaluates the first element of the vector. For example:

```
temp_vec <- c(36, 37, 38)
```

```
classify_temp(temp_vec)  # This won't work correctly
```

```
Error in if (temp < 35) { : the condition has length > 1
```

To address the entire vector, you should use vectorized functions such as `ifelse` or `case_when` from the `dplyr` package. Here's how you can employ `ifelse`:

```
classify_temp <- function(temp) {
  out <- ifelse(temp < 35, "hypothermia",
                ifelse(temp >= 35 & temp <= 37, "normal",
                       ifelse(temp > 37, "fever", NA)))
  return(out)
}

classify_temp(temp_vec)  # This works as expected
```

```
## [1] "normal" "normal" "fever"
```

For a cleaner and more readable alternative, `dplyr::case_when` can also be used:

```
classify_temp <- function(temp) {
 case_when(
    temp < 35 ~ "hypothermia",
    temp >= 35 & temp <= 37 ~ "normal",
    temp > 37 ~ "fever",
    TRUE ~ NA_character_
  )
}

classify_temp(temp_vec)  # This also works as expected
```

```
## [1] "normal" "normal" "fever"
```

This function works seamlessly with data frames too:

```
NHSRdatasets::synthetic_news_data %>%
  select(temp) %>%
  mutate(temp_class = classify_temp(temp))
```

```
## # A tibble: 1,000 × 2
##     temp temp_class
##    <dbl> <chr>
##  1  36.8 normal
##  2  35   normal
##  3  36.2 normal
##  4  36.9 normal
##  5  36.4 normal
##  6  35.3 normal
##  7  35.6 normal
```

```
##  8  37.2 fever
##  9  35.5 normal
## 10  35.3 normal
## # ℹ 990 more rows
```

Practice Classifying Dosage of Isoniazid

Let's apply this knowledge to a practical case. Consider the following attempt at writing a function that calculates dosages of the drug isoniazid for adults weighing more than 30kg:


```r
calculate_isoniazid_dosage <- function(weight) {
  if (weight < 30) {
    stop("Weight must be at least 30 kg.")
  } else if (weight <= 35) {
    return(150)
  } else if (weight <= 45) {
    return(200)
  } else if (weight <= 55) {
    return(300)
  } else if (weight <= 70) {
    return(300)
  } else {
    return(300)
  }
}
```

This function fails with a vector of weights. Your task is to write a new function `calculate_isoniazid_dosage2()` that can handle vector inputs. To ensure all weights are above 30kg, you'll use the `any()` function within your error checking.

Here's a scaffold to get you started:

```r
calculate_isoniazid_dosage2 <- function(weight) {
  if (any(weight < 30)) stop("Weights must all be at least 30
kg.")

  # Your code here

  return(out)
}

calculate_isoniazid_dosage2(c(30, 40, 50, 100))
```


```
## [1] "Fever"
```

Where to Keep Your Functions

When you're writing scripts in R, deciding where to store your functions is an important consideration for maintaining clean code and efficient workflows. Here are some key strategies:

1) Top of the Script

Placing functions at the top of your script is a simple and commonly used practice.

2) Separate Script That is Sourced

As your project grows, you might have several functions. In such cases, storing them in a separate R script can keep your main analysis script tidy. You can then 'source' this script to load the functions.

```
# Sourcing a separate script
source("path_to_your_functions_script.R")
```

3) GitHub Gist

For functions that you frequently reuse or want to share with the community, storing them in a GitHub Gist is a good option. Create an account on github, then create a public gist at https://gist.github.com/. Then, you can copy and paste your function into the gist. Finally, you can obtain the gist URL and source it in your R script using the `source_gist()` function from the `devtools` package:

```
# Sourcing from a GitHub Gist
pacman::p_load(devtools)
devtools::source_gist("https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5b
```

When you run the code above, it will define a new function called `hello_from_gist()` that we created for this lesson.

```
hello_from_gist("Student")
```

```
## [1] "Hello Student! This is a function from a gist!"
```

You can see the code by going directly to the URL: https://gist.github.com/kendavidn/a5e1ce486910e6b2dc77a5b6bddf87d0.

The code in the gist can be updated at any time, and the changes will be reflected in your script when you source it again.

4) Package

As previously mentioned, functions can also be stored in packages. This is a more advanced option that requires knowledge of R package development. For more information, see the Writing R Extensions manual.

Wrap Up!

Congratulations on getting through the lesson!

You now have the key building blocks to create custom functions that automate repetitive tasks in your R workflows. Of course, there's much more to learn about functions, but you now have the foundation to build on.

Answer Key

Age Months Function

```
years_to_months <- function(years) {
  months <- years * 12
  return(months)
}

# Test
years_to_months(12)
```

```
## [1] 144
```

Celsius to Fahrenheit Function

```
celsius_to_fahrenheit <- function(celsius) {
  fahrenheit <- celsius * 1.8 + 32
  return(fahrenheit)
}

# Test
beaver1 %>%
  select(temp) %>%
  mutate(Fahrenheit = celsius_to_fahrenheit(temp))
```

```
##      temp Fahrenheit
## 1   36.33     97.394
## 2   36.34     97.412
## 3   36.35     97.430
## 4   36.42     97.556
## 5   36.55     97.790
## 6   36.69     98.042
## 7   36.71     98.078
## 8   36.75     98.150
## 9   36.81     98.258
## 10  36.88     98.384
## 11  36.89     98.402
## 12  36.91     98.438
## 13  36.85     98.330
## 14  36.89     98.402
## 15  36.89     98.402
## 16  36.67     98.006
## 17  36.50     97.700
## 18  36.74     98.132
## 19  36.77     98.186
## 20  36.76     98.168
## 21  36.78     98.204
## 22  36.82     98.276
## 23  36.89     98.402
## 24  36.99     98.582
## 25  36.92     98.456
## 26  36.99     98.582
## 27  36.89     98.402
## 28  36.94     98.492
## 29  36.92     98.456
## 30  36.97     98.546
## 31  36.91     98.438
## 32  36.79     98.222
## 33  36.77     98.186
## 34  36.69     98.042
## 35  36.62     97.916
## 36  36.54     97.772
## 37  36.55     97.790
## 38  36.67     98.006
## 39  36.69     98.042
```

```
## 40  36.62     97.916
## 41  36.64     97.952
## 42  36.59     97.862
## 43  36.65     97.970
## 44  36.75     98.150
## 45  36.80     98.240
## 46  36.81     98.258
## 47  36.87     98.366
## 48  36.87     98.366
## 49  36.89     98.402
## 50  36.94     98.492
## 51  36.98     98.564
## 52  36.95     98.510
## 53  37.00     98.600
## 54  37.07     98.726
## 55  37.05     98.690
## 56  37.00     98.600
## 57  36.95     98.510
## 58  37.00     98.600
## 59  36.94     98.492
## 60  36.88     98.384
## 61  36.93     98.474
## 62  36.98     98.564
## 63  36.97     98.546
## 64  36.85     98.330
## 65  36.92     98.456
## 66  36.99     98.582
## 67  37.01     98.618
## 68  37.10     98.780
## 69  37.09     98.762
## 70  37.02     98.636
## 71  36.96     98.528
## 72  36.84     98.312
## 73  36.87     98.366
## 74  36.85     98.330
## 75  36.85     98.330
## 76  36.87     98.366
## 77  36.89     98.402
## 78  36.86     98.348
## 79  36.91     98.438
## 80  37.53     99.554
## 81  37.23     99.014
## 82  37.20     98.960
## 83  37.25     99.050
## 84  37.20     98.960
## 85  37.21     98.978
## 86  37.24     99.032
## 87  37.10     98.780
## 88  37.20     98.960
## 89  37.18     98.924
## 90  36.93     98.474
## 91  36.83     98.294
## 92  36.93     98.474
## 93  36.83     98.294
## 94  36.80     98.240
## 95  36.75     98.150
```

```
## 96  36.71     98.078
## 97  36.73     98.114
## 98  36.75     98.150
## 99  36.72     98.096
## 100 36.76     98.168
## 101 36.70     98.060
## 102 36.82     98.276
## 103 36.88     98.384
## 104 36.94     98.492
## 105 36.79     98.222
## 106 36.78     98.204
## 107 36.80     98.240
## 108 36.82     98.276
## 109 36.84     98.312
## 110 36.86     98.348
## 111 36.88     98.384
## 112 36.93     98.474
## 113 36.97     98.546
## 114 37.15     98.870
```

BMI Function

```
calc_bmi <- function(weight, height) {
  bmi <- weight / (height^2)
  return(bmi)
}

# Test
library(medicaldata)
medicaldata::smartpill %>%
  as_tibble() %>%
  select(Weight, Height) %>%
  mutate(BMI = calc_bmi(Weight, Height))
```

```
## # A tibble: 95 × 3
##    Weight Height     BMI
##     <dbl>  <dbl>   <dbl>
##  1  102.    183. 0.00305
##  2  102.    180. 0.00314
##  3   68.0   180. 0.00209
##  4   69.9   175. 0.00227
##  5   44.9   152. 0.00193
##  6   94.8   185. 0.00276
##  7   86.2   188. 0.00244
##  8   76.2   165. 0.00280
##  9   74.4   173. 0.00249
## 10   64.9   170. 0.00224
## # ℹ 85 more rows
```

Practice with the ... Argument

```
calculate_calories <- function(carb_grams, protein_grams, fat_grams, ...) {
  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  result_formatted <- format(result, ...)
  return(result)
}
```

Debugging a Function with Conditional Logic

```
check_negatives <- function(numbers) {
  if (any(numbers < 0)) {
    print("x contains negative numbers")
  } else {
    print("x does not contain negative numbers")
  }
}

# Test
check_negatives(c(8, 3, -2, 5))
```

```
## [1] "x contains negative numbers"
```

```
check_negatives(c(10, 20, 30, 40))
```

```
## [1] "x does not contain negative numbers"
```

Argument Checking Practice

```
calculate_calories2 <- function(carb_grams = 0, protein_grams = 0, fat_grams =
0) {

  if (!is.numeric(carb_grams)) {
    stop("carb_grams must be numeric")
  }

  if (!is.numeric(protein_grams)) {
    stop("protein_grams must be numeric")
  }

  if (!is.numeric(fat_grams)) {
    stop("fat_grams must be numeric")
  }


  result <- (carb_grams * 4) + (protein_grams * 4) + (fat_grams * 9)
  return(result)
}
```

Practice Classifying Dosage of Isoniazid

```
calculate_isoniazid_dosage2 <- function(weight) {
  if (any(weight < 30)) stop("Weights must all be at least 30 kg.")

  dosage <- case_when(
    weight <= 35 ~ 150,
    weight <= 45 ~ 200,
    weight <= 55 ~ 300,
    weight <= 70 ~ 300,
    TRUE ~ 300
  )
    return(dosage)
}

calculate_isoniazid_dosage2(c(30, 40, 50, 100))
```

```
## [1] 150 200 300 300
```

Contributors

The following team members contributed to this lesson:

DANIEL CAMARA

Data Scientist at the GRAPH Network and fellowship as Public Health

researcher at Fiocruz, Brazil
Passionate about lots of things, especially when it involves people leading lives with more equality and freedom

EDUARDO ARAUJO

Student at Universidade Tecnologica Federal do Parana
Passionate about reproducible science and education

LAURE VANCAUWENBERGHE

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

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

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

- Barnier, Julien. "Introduction à R et au tidyverse." Accessed May 23, 2022. https://juba.github.io/tidyverse
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science." Accessed May 25, 2022. https://r4ds.had.co.nz/

Loops in R

Introduction

At the heart of programming is the concept of repeating a task multiple times. A `for` loop is one fundamental way to do that. Loops enable efficient repetition, saving time and effort.

Mastering this concept is essential for writing intelligent and efficient R code.

Let's dive in and enhance your coding skills!

Learning Objectives

By the end of this lesson, you will be able to:

- Explain the syntax and structure of a basic `for` loop in R
- Use index variables to iterate through multiple vectors simultaneously in a loop
- Integrate `if/else` conditional statements within a loop
- Store loop results in vectors and lists
- Apply loops to tasks like analyzing multiple datasets and generating multiple plots
- Debug loops by isolating and testing single iterations

Packages

This lesson will require the following packages to be installed and loaded:

```r
# Load packages
if(!require(pacman)) install.packages("pacman")
pacman::p_load(tidyverse, here, openxlsx, tools, outbreaks, medicaldata)
```

Intro to `for` Loops

Let's start with a simple example. Suppose we have a vector of children's ages in years, and we want to convert these to months:

```r
ages <- c(7, 8, 9) # Vector of ages in years
```

We can do this easily with the * operation in R:

```r
ages * 12
```

```
## [1]  84  96 108
```

But let's walk through how we could accomplish this using a `for` loop instead, since that is (conceptually) what R is doing under the hood.

```r
for (age in ages) print(age * 12)
```

```
## [1] 84
## [1] 96
## [1] 108
```

In this loop, `age` is a temporary variable that takes the value of each element in `ages` during each iteration. First, `age` is 7, then 8, then 9.

You can choose any name for this variable:

```r
for (random_name in ages) print(random_name * 12)
```

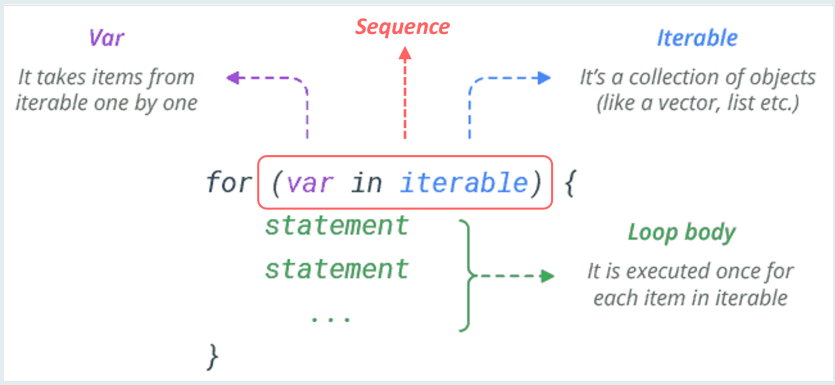
```
## [1] 84
## [1] 96
## [1] 108
```

If the content of the loop is more than one line, you need to use curly brackets `{}` to indicate the body of the loop.

```
for (age in ages) {
  month_age = age * 12
  print(month_age)
}
```

```
## [1] 84
## [1] 96
## [1] 108
```

The general structure of any `for` loop is illustrated in the diagram below:



Hours to Minutes Basic Loop

Try converting hours to minutes using a `for` loop. Start with this vector of hours:

```
hours <- c(3, 4, 5) # Vector of hours
# Your code here

for ___
_____ # convert hours to minutes and print
```

Loops can be nested within each other. For instance:

```
for (i in 1:2) {
  for (j in 1:2) {
    print(i * j)
  }
}
```

```
## [1] 1
## [1] 2
## [1] 2
## [1] 4
```

SIDE NOTE

This creates a combination of `i` and `j` values as shown in this table:

i	j	i*j
1	1	1
1	2	2
2	1	2
2	2	4

Nested loops are less common though, and often have more efficient alternatives.

Are `for` Loops Useful in R?

While `for` loops are foundational in many programming languages, their usage in R is somewhat less frequent. This is because R inherently handles *vectorized* operations, automatically applying a function to each element of a vector.

For example, our initial age conversion could be achieved without a loop:

```
ages * 12
```

```
## [1]  84  96 108
```

Moreover, R typically deals with data frames rather than raw vectors. For data frames, we often use functions from the `tidyverse` package to apply operations across columns:

```
ages_df <- tibble(age = ages)
ages_df %>%
  mutate(age_months = age * 12)
```

```
## # A tibble: 3 × 2
##     age age_months
##   <dbl>      <dbl>
## 1     7         84
```

```
## 2     8         96
## 3     9        108
```

However, there are scenarios where loops are useful, especially when working with multiple data frames or non-dataframe (sometimes called *non-rectangular*) objects.

We will explore these later in the lesson, but first we'll spend some more time getting comfortable with loops using toy examples.

Loops vs function mapping

It's important to note that loops can often be replaced by custom functions which are then mapped across a vector or data frame.

We're teaching loops nonetheless because they are quite easy to learn, reason about and debug, even for beginners.

Looping with an Index

It is often useful to loop through a vector using an index (plural: indices), which is a counter that keeps track of the current iteration.

Let's look at our `ages` vector again, which we want to convert to months:

```
ages <- c(7, 8, 9) # Vector of ages in years
```

To use indices in a loop, we first create a sequence that represents each position in the vector:

```
1:length(ages) # Create a sequence of indices that is the same length as ages
```

```
## [1] 1 2 3
```

```
indices <- 1:length(ages)
```

Now, `indices` has values 1, 2, 3, corresponding to the positions in `ages`. We use this in a `for` loop as follows:

```
for (i in indices) {
  print(ages[i] * 12)
```

```
## [1] 84
## [1] 96
## [1] 108
```

In this code, `ages[i]` refers to the `i`th element in our `ages` list.

The name of the variable `i` is arbitrary. We could have used `j` or `index` or `position` or anything else.

```
for (position in indices) {
  print(ages[position] * 12)
}
```

```
## [1] 84
## [1] 96
## [1] 108
```

Often we do not need to create a separate variable for the indices. We can just use the `:` operator to create a sequence directly in the `for` loop:

```
for (i in 1:length(ages)) {
  print(ages[i] * 12)
}
```

```
## [1] 84
## [1] 96
## [1] 108
```

Such index-based loops are useful for working with multiple vectors at the same time. We will see this in the next section.

Hours to Minutes Indexed Loop

Rewrite your loop from last question using indices:

```
hours <- c(3, 4, 5) # Vector of hours

# Your code here

for ___ {
  ___
}
```

SIDE NOTE

The function `seq_along()` is a shortcut for creating a sequence of indices. It is equivalent to `1:length()`:

```
# These two are equivalent:
seq_along(ages)
```

```
## [1] 1 2 3
```

```
1:length(ages)
```

```
## [1] 1 2 3
```

Looping on Multiple Vectors

Looping with indices allows us to work with multiple vectors simultaneously. Suppose we have vectors for ages and heights:

```
ages <- c(7, 8, 9) # ages in years
heights <- c(120, 130, 140) # heights in cm
```

We can loop through both using the index method:

```
for(i in 1:length(ages)) {
  age <- ages[i]
  height <- heights[i]

  print(paste("Age:", age, "Height:", height))
}
```

```
## [1] "Age: 7 Height: 120"
## [1] "Age: 8 Height: 130"
## [1] "Age: 9 Height: 140"
```

In each iteration: - `i` is the index. - We extract the ith element from each vector

and print it.

Alternatively, we can skip the variable assignment and use the indices in the `print()` statement directly:

```
for(i in 1:length(ages)) {
  print(paste("Age:", ages[i], "Height:", heights[i]))
}
```

```
## [1] "Age: 7 Height: 120"
## [1] "Age: 8 Height: 130"
## [1] "Age: 9 Height: 140"
```

BMI Calculation Loop

Using a for loop, calculate the Body Mass Index (BMI) of the three individuals shown below. The formul for BMI is `BMI = weight / (height ^ 2)`.


```
weights <- c(30, 32, 35) # Weights in kg
heights <- c(1.2, 1.3, 1.4) # Heights in meters

for(i in ____________________) {

  ____________________________

  print(paste("Weight:", ____,
              "Height:", ____,
              "BMI:", ____,
              ))

}
```

Storing Loop Results

In most cases, you'll want to store the results of a loop rather than just printing them as we have been doing above. Let's look at how to do this.

Consider our age-to-months example:

```r
ages <- c(7, 8, 9)

for (age in ages) {
  print(paste(age * 12, "months"))
}
```

```
## [1] "84 months"
## [1] "96 months"
## [1] "108 months"
```

To store these converted ages, we first create an empty vector:

```r
ages_months <- vector(mode = "numeric", length = length(ages))
# This can also be written as:
ages_months <- vector("numeric", length(ages))

ages_months # Shows the empty vector
```

```
## [1] 0 0 0
```

This creates a numeric vector of the same length as `ages`, initially filled with zeros. To store a value in the vector, we do the following:

```r
ages_months[1] <- 99 # Store 99 in the first element of ages_months
ages_months[2] <- 100 # Store 100 in the second element of ages_months
ages_months
```

```
## [1]  99 100   0
```

Now, let's execute the loop, storing the results in `ages_months`:

```r
ages_months <- vector("numeric", length(ages))

for (i in 1:length(ages)) {
  ages_months[i] <- ages[i] * 12
}
ages_months
```

```
## [1]  84  96 108
```

In this loop:

- On the first iteration, `i` is 1. We multiply the first element of `ages` by 12 and store it in the first element of `ages_months`.

- Then `i` is 2, then 3. In each iteration, we multiply the corresponding element of `ages` by 12 and store it in the corresponding element of `ages_months`.

PRACTICE

(in RMD)

Height cm to m

Use a for loop to convert height measurements from cm to m. Store the results in a vector called `height_meters`.

```
height_cm <- c(180, 170, 190, 160, 150) # Heights in cm

height_m <- vector(_______________-) # numeric vector of same
length as heigh_cm

for ___ {
  height_m[i] <- ____________________
}
```

WATCH OUT

In order to save the results from your iteration, you must create your empty object **outside** the loop. Otherwise, you will only save the result of the last iteration.

This is a common mistake. Consider the below as an example:

```
ages <- c(7, 8, 9)

for (i in 1:length(ages)) {
  ages_months <- vector("numeric", length(ages))
  ages_months[i] <- ages[i] * 12
}
ages_months
```

```
## [1]   0   0 108
```

Do you see the problem?

SIDE NOTE

If you are in a rush, you can skip using the `vector()` function and initialize your vector with `c()` instead, then progressively fill it with values by index:

```r
ages_months <- c()

for (i in 1:length(ages)) {
  ages_months[i] <- ages[i] * 12
}
ages_months
```

```
## [1]  84  96 108
```

SIDE NOTE

And you can also skip the index and use `c()` to append values to the end of the vector:

```r
ages_months <- c()

for (age in ages) {
  ages_months <- c(ages_months, age * 12)
}
ages_months
```

```
## [1]  84  96 108
```

However, in both of these cases, R does not know the final length of the vector as it's going through the iterations, so it has to reallocate memory at each iteration. This can cause slow performance if you are working with large vectors.

If Statements in Loops

Just as `if` statements can be used in functions, they can be integrated into loops.

Consider this example:

```r
age_vec <- c(2, 12, 17, 24, 60) # Vector of ages

for (age in age_vec) {
  if (age < 18) print(paste("Child, Age", age ))
}
```

```
## [1] "Child, Age 2"
## [1] "Child, Age 12"
```

```
## [1] "Child, Age 17"
```

It is often clearer to use curly braces to indicate the `if` statement's body. It also allows us to add more lines of code to the body of the `if` statement:

```
for (age in age_vec) {
  if (age < 18) {
    print("Processing:")
    print(paste("Child, Age", age ))
  }
}
```

```
## [1] "Processing:"
## [1] "Child, Age 2"
## [1] "Processing:"
## [1] "Child, Age 12"
## [1] "Processing:"
## [1] "Child, Age 17"
```

Let's add another condition to classify as 'Child' or 'Teen':

```
for (age in age_vec) {
  if (age < 13) {
    print(paste("Child, Age", age))
  } else if (age >= 13 && age < 18) {
    print(paste("Teen, Age", age))
  }
}
```

```
## [1] "Child, Age 2"
## [1] "Child, Age 12"
## [1] "Teen, Age 17"
```

We can include a single `else` statement at the end to catch all other ages:

```
for (age in age_vec) {
  if (age < 13) {
    print(paste("Child, Age", age))
  } else if (age >= 13 && age < 18) {
    print(paste("Teen, Age", age))
  } else {
    print(paste("Adult, Age", age))
  }
}
```

```
## [1] "Child, Age 2"
## [1] "Child, Age 12"
## [1] "Teen, Age 17"
```

```
## [1] "Adult, Age 24"
## [1] "Adult, Age 60"
```

To store these classifications, we can create an empty vector, and use an index-based loop to store the results:

```r
age_class <- vector("character", length(age_vec)) # Create empty vector
for (i in 1:length(age_vec)) {
  if (age_vec[i] < 13) {
    age_class[i] <- "Child"
  } else if (age_vec[i] >= 13 && age_vec[i] < 18) {
    age_class[i] <- "Teen"
  } else {
    age_class[i] <- "Adult"
  }
}
age_class
```

```
## [1] "Child" "Child" "Teen"  "Adult" "Adult"
```

Temperature Classification

You have a vector of body temperatures in Celsius. Classify each temperature as 'Hypothermia', 'Normal', or 'Fever' using a `for` loop combined with `if` and `else` statements.

Use these rules:

- Below 36.0°C: 'Hypothermia'
- Between 36.0°C and 37.5°C: 'Normal'
- Above 37.5°C: 'Fever'


```r
body_temps <- c(35, 36.5, 37, 38, 39.5) # Body temperatures in
Celsius
classif_vec <- vector(_____________________) # character vec,
length of body_temps
for (i in 1:length(________)) {
    # Add your if-else logic here
  if (body_temps[i] < 36.0) {
    out <- "Hypothermia"
  } ## add other conditions


    # Final print statement
    classif_vec[i] <- paste(body_temps[i], "°C is", out)
}
classif_vec
```

An expected output is below

```
35°C is Hypothermia
36.5°C is Normal
37°C is Normal
38°C is Fever
39.5°C is Fever
```

Quick Techniques for Debugging `for` Loops

Efficient editing and debugging are crucial when working with `for` loops in R. There are many approaches for this, but for now, we'll show two of the simplest ones:

- Isolate and running a single iteration of the loop
- Adding `print()` statements to the loop to print out the values of variables at each iteration

Isolating and Running a Single Iteration

Consider this loop which we saw previously:

```r
age_vec <- c(2, 12, 17, 24, 60) # Vector of ages
age_class <- vector("character", length(age_vec))

for (i in 1:length(age_vec)) {
  if (age_vec[i] < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }
}
age_class
```

```
## [1] "Child" "Child" "Child" "Adult" "Adult"
```

Let's see an example of an error we might run into when using the loop:

```r
# Age vector from the fluH7N9_china_2013 dataset
flu_dat <- outbreaks::fluH7N9_china_2013
head(flu_dat)

flu_dat_age <- flu_dat$age
age_class <- vector("character", length(flu_dat_age))
for (i in 1:length(flu_dat_age)) {
  if (flu_dat_age[i] < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }
}
```

We get this error:

```
Error in if (flu_dat_age[i] < 18) { :
  missing value where TRUE/FALSE needed
In addition: Warning message:
In Ops.factor(flu_dat_age[i], 18) : ‘<’ not meaningful for factors
```

You may already know what this error means, but let's say you didn't.

We can step into the loop and manually step through the first iteration to see what's going on:

```
for (i in 1:length(flu_dat_age)) {

  # ▶ Run from this line
  i <- 1 # Manually set i to 1

  # Then highlight `flu_dat_age[i]` and press Ctrl + Enter to run just this
code
  # After that, highlight and run `flu_dat_age[i] < 18`

  if (flu_dat_age[i] < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }

}
```

Following the above process, we can see that `flu_dat_age` is a factor, not a numeric vector. We can manually change this, in the midst of the debugging process. It is a good idea to first convert the factor to a character vector, and then to a numeric vector. Otherwise, we may get unexpected results.

Consider:

```
flu_dat_age[75]
```

```
## Error in eval(expr, envir, enclos): object 'flu_dat_age' not found
```

```
as.numeric(flu_dat_age[75])
```

```
## Error in eval(expr, envir, enclos): object 'flu_dat_age' not found
```

```
# `?`, which stands for missing in this case is converted to 1, at it is the
first level of the factor

# We therefore need:
as.numeric(as.character(flu_dat_age[75]))
```

```
## Error in eval(expr, envir, enclos): object 'flu_dat_age' not found
```

Now let's try to fix the loop, and run just the first iteration again:

```
for (i in 1:length(flu_dat_age)) {

  # ▶ Run from this line
  i <- 1 # Manually set i to 1

  age_num <- as.numeric(as.character(flu_dat_age[i]))

  # Then highlight `age_num < 18` and press Ctrl + Enter
  if (age_num < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }

}
```

Now the first iteration works, but let's see what happens when we run the entire loop:

```
age_class <- vector("character", length(flu_dat_age))

for (i in 1:length(flu_dat_age)) {
  age_num <- as.numeric(as.character(flu_dat_age[i]))

  if (age_num < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }
}
head(age_class)
```

```
Error in if (age_num < 18) { :
  missing value where TRUE/FALSE needed
```

Again, you may already know what this error means, but let's say you didn't. We'll try our next debugging technique.

Adding Print Statements to the Loop

In the last section, we saw that the loop works fine for the first iteration, but seems to fail on a further iteration.

To catch which the iteration fails on, we can add `print()` statements to the loop:

```r
for (i in 1:length(flu_dat_age)) {

  print(i) # Print the iteration number
  age_num <- as.numeric(as.character(flu_dat_age[i]))

  print(age_num) # Print the value of age_num

  if (age_num < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }

  print(age_class[i]) # Print the value of the output
}
head(age_class)
```

Now, when we inspect the output, we can see that the loop fails on the 74th iteration:

```
[1] 73 👈 73rd iteration
[1] 43 👈 Value of age_num
[1] "Adult, Age 43" 👈 Value of output on 73rd iteration
[1] 74 👈 74th iteration
[1] NA 👈 Value of age_num
```

This happens because the 74th value of `flu_dat_age` is NA (because of our factor to numeric conversion), so R cannot evaluate whether it is less than 18.

We can fix this by adding an `if` statement to check for NA values:

```r
for (i in 1:length(flu_dat_age)) {

  age_num <- as.numeric(as.character(flu_dat_age[i]))

  if (is.na(age_num)) {
    age_class[i] <- "NA"
  } else if (age_num < 18) {
    age_class[i] <- "Child"
  } else {
    age_class[i] <- "Adult"
  }
}
```

```
## Error in eval(expr, envir, enclos): object 'flu_dat_age' not found
```

```r
# Check the 74th value of age_class
age_class[74]
```

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

Great! Now we've fixed the error.

As you can see, even with our "toy" loop, debugging can be a time-consuming process. As your mother used to say "Programming is 98% debugging and 2% writing code."

R offers several other techniques for diagnosing and managing errors:

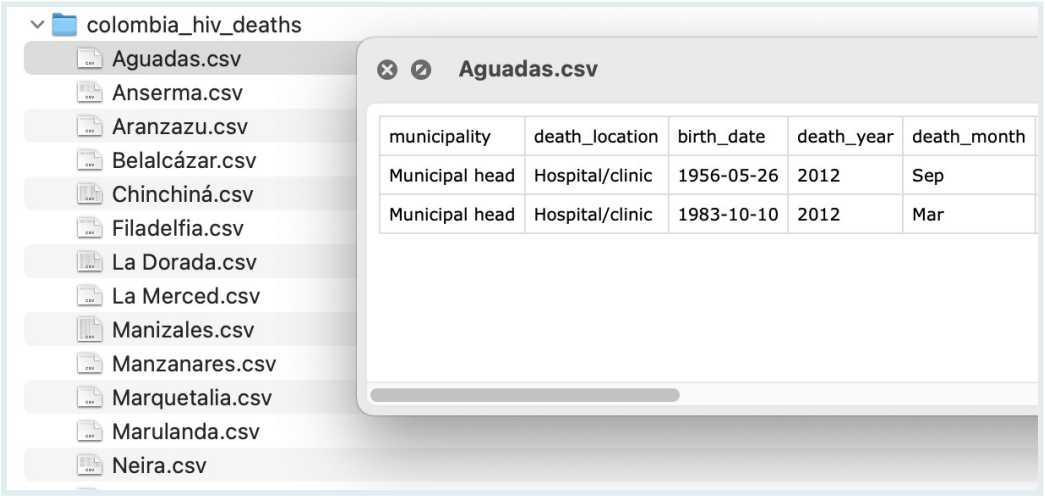
- The `try()` and `tryCatch()` functions allow error catching while continuing the loop's execution.
- The `browser()` function pauses the loop at a designated point, enabling step-by-step execution.

These are more advanced methods, and while they are not covered here, you can refer to the R documentation for further guidance when needed. Or consult Hadley Wickham's Advanced R book.

Real Loops Application 1: Analyzing Multiple Datasets

Now that you have a solid understanding of `for` loops, let's apply our knowledge to a more realistic looping task: working with multiple datasets.

We have a folder of CSV files containing HIV deaths data for municipalities in Colombia.



Imagine we were asked to compile a single table with the following information about each dataset: the number of rows (number of deaths), the number of columns, and the names of all columns.

We could do this one by one, but that would be tedious and error-prone. Instead, we can use a loop to automate the process.

First, let's list the files in the folder:

```
colom_data_paths <- list.files(here("data/colombia_hiv_deaths"),
                               full.names = TRUE)
head(colom_data_paths) # Show first 6 file paths
```

```
## [1]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/data/colombia_hi
## [2]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/data/colombia_hi
## [3]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/data/colombia_hi
## [4]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/data/colombia_hi
## [5]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/data/colombia_hi
## [6]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/data/colombia_hi
```

Now, let's import one dataset as an example to demonstrate what we want to achieve. Once we've done this, we can apply the same process to all datasets.

```
colom_data <- read_csv(colom_data_paths[1]) # Import first dataset
```

```
## Rows: 2 Columns: 15
## ── Column specification
──────────────────────────────────────────────────────────────────────────
## Delimiter: ","
## chr  (9): municipality, death_location, death_month, municipality_code,
primary_cause_death_des...
## dbl  (2): death_year, death_day
## lgl  (3): tertiary_cause_death_description,
quaternary_cause_death_description, quaternary_caus...
## date (1): birth_date
##
## 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.
```

```
colom_data
```

```
## # A tibble: 2 × 15
##   municipality   death_location  birth_date death_year
##   <chr>          <chr>           <date>          <dbl>
## 1 Municipal head Hospital/clinic 1956-05-26       2012
## 2 Municipal head Hospital/clinic 1983-10-10       2012
## # ℹ 11 more variables: death_month <chr>, death_day <dbl>,
## #   municipality_code <chr>, …
```

Then we apply a range of R functions to gather the information we want from each dataset:

```
file_path_sans_ext(basename(colom_data_paths[1]))  # Dataset/Municipality name
```

```
## [1] "Aguadas"
```

```
nrow(colom_data) # Number of rows, which is equivalent to the number of deaths
```

```
## [1] 2
```

```
ncol(colom_data) # Number of columns
```

```
## [1] 15
```

```
paste(names(colom_data), collapse = ", ") # Names of all columns
```

```
## [1] "municipality, death_location, birth_date, death_year, death_month,
death_day, municipality_code, primary_cause_death_description,
primary_cause_death_code, secondary_cause_death_description,
secondary_cause_death_code, tertiary_cause_death_description,
tertiary_cause_death_code, quaternary_cause_death_description,
quaternary_cause_death_code"
```

SIDE NOTE

`basename`: extracts the file name from a file path.

```
colom_data_paths[1]
```

```
## [1]
"/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/dat
```

```
basename(colom_data_paths[1])
```

```
## [1] "Aguadas.csv"
```

And `file_path_sans_ext` from the {tools} package removes the file extension from file names. We use it together with `basename` to get the municipality name.

```
file_path_sans_ext(basename(colom_data_paths[1]))
```

```
## [1] "Aguadas"
```

Now, we need to make a data frame with this information. We can use the `tibble` function to do this:

```
single_row <-
  tibble(dataset = basename(colom_data_paths[1]),
         n_deaths = nrow(colom_data),
         n_cols = ncol(colom_data),
         col_names = paste(names(colom_data), collapse = ", "))
single_row
```

```
## # A tibble: 1 × 4
##   dataset     n_deaths n_cols col_names
##   <chr>          <int>  <int> <chr>
## 1 Aguadas.csv        2     15 municipality, death_location,…
```

So we're going to need to repeat this process for each dataset. Within the loop, we will store each single-row data frame in a list, then combine them at the end. Recall that lists are R objects that can contain any other R objects, including data frames.

Let's initialize this empty list now:

```
data_frames_list <- vector("list", length(colom_data_paths))
head(data_frames_list) # Show first 6 elements
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
```

Let's add the first single-row data frame to the list:

```
data_frames_list[[1]] <- single_row
```

Now if we look at the list, we see the first element is the single-row data frame:

```
head(data_frames_list)
```

```
## [[1]]
## # A tibble: 1 × 4
##   dataset     n_deaths n_cols col_names
##   <chr>          <int>  <int> <chr>
## 1 Aguadas.csv        2     15 municipality, death_location,…
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
```

```
## [[6]]
## NULL
```

And we can access the data frame by subsetting the list:

```
data_frames_list[[1]]
```

```
## # A tibble: 1 × 4
##   dataset     n_deaths n_cols col_names
##   <chr>          <int>  <int> <chr>
## 1 Aguadas.csv        2     15 municipality, death_location,…
```

Note the use of double brackets for accessing elements of the list.

We now have all the pieces we need to create a loop that will process each dataset and store the results in a list. Let's go!

```
for (i in 1:length(colom_data_paths)) {
    path <- colom_data_paths[i]

    # Import
    colom_data <- read_csv(path)

    # Get info
    n_deaths <- nrow(colom_data)
    n_cols <- ncol(colom_data)
    col_names <- paste(names(colom_data), collapse = ", ")

    # Create data frame for this dataset
    hiv_dat_row <- tibble(dataset = file_path_sans_ext(basename(path)),
                          n_deaths = n_deaths,
                          n_cols = n_cols,
                          col_names = col_names)

    # Store in the list
    data_frames_list[[i]] <- hiv_dat_row
}
```

Let's check the list:

```
head(data_frames_list, 2) # Show first 2 elements
```

```
## [[1]]
## # A tibble: 1 × 4
##   dataset n_deaths n_cols col_names
##   <chr>      <int>  <int> <chr>
## 1 Aguadas        2     15 municipality, death_location, bir…
##
## [[2]]
## # A tibble: 1 × 4
##   dataset n_deaths n_cols col_names
```

```
##   <chr>      <int>  <int> <chr>
## 1 Anserma       15     16 municipality, death_location, dea…
```

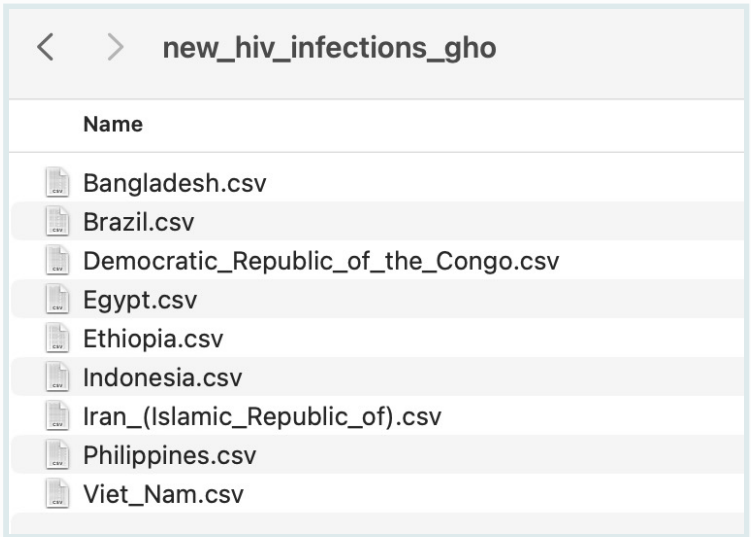
And now we can combine all the data frames in the list into one final data frame. This can be done with the `bind_rows` function from the {dplyr} package:

```
colom_data_final <- bind_rows(data_frames_list)
colom_data_final
```

```
## # A tibble: 25 × 4
##    dataset    n_deaths n_cols col_names
##    <chr>         <int>  <int> <chr>
##  1 Aguadas           2     15 municipality, death_location,…
##  2 Anserma          15     16 municipality, death_location,…
##  3 Aranzazu          2     16 municipality, death_location,…
##  4 Belalcázar        4     14 municipality, death_location,…
##  5 Chinchiná        62     17 municipality, death_location,…
##  6 Filadelfia        5     15 municipality, death_location,…
##  7 La Dorada        46     16 municipality, death_location,…
##  8 La Merced         3     17 municipality, death_location,…
##  9 Manizales       199     17 municipality, death_location,…
## 10 Manzanares        3     14 municipality, death_location,…
## # ℹ 15 more rows
```

File Properties

You have a folder containing CSV files with data on HIV cases, sourced from WHO.



Using the principles learned, you will write a loop that extracts the following information from each dataset and stores this in a single data frame:

- The name of the dataset (i.e. the country)
- The size of the dataset in bytes
- The date the dataset was last modified

You can use the `file.size()` and `file.mtime()` functions to get the latter two pieces of information. For example:

```
file.size(here("data/new_hiv_infections_gho/Bangladesh.csv"))
```

```
## [1] 6042
```

```
file.mtime(here("data/new_hiv_infections_gho/Bangladesh.csv"))
```

```
## [1] "2023-12-11 17:34:28 GMT"
```

Note that you do not need to import the CSVs to get this information.


```
# List files
csv_files <- list.files(path = "data/new_hiv_infections_gho",

_____________________________________________)

for (i in _______________) {

  path <- csv_files[i]

  # Get the country name. Hint: use file_path_sans_ext and
basename
  country_name <- ________________________________________

  # Get the file size and date modified
  size <- ________________________________________________
  date <- ________________________________________________

  # Data frame for this interation. Hint: use tibble() to
combine the objects above
  hiv_dat_row <- ____________________________________________

  # Store in the list. Hint: use double brackets and the index
i
  data_frames_list_____ <- hiv_dat_row
}

# Combine into one data frame
hiv_file_info_final <- bind_rows(data_frames_list)
```

Data Filtering Loop

You will again work with the folder of HIV datasets from the previous question. Here is an example of one of the country datasets from that folder:


```
bangla_dat <-
read_csv(here("data/new_hiv_infections_gho/Bangladesh.csv"))
bangla_dat
```

```
## # A tibble: 89 × 5
##    Continent       Country     Year Sex
##    <chr>           <chr>      <dbl> <chr>
##  1 South-East Asia Bangladesh  2022 Female
##  2 South-East Asia Bangladesh  2022 Both sexes
##  3 South-East Asia Bangladesh  2022 Male
##  4 South-East Asia Bangladesh  2021 Female
##  5 South-East Asia Bangladesh  2021 Both sexes
```

```
##  6 South-East Asia Bangladesh  2021 Male
##  7 South-East Asia Bangladesh  2020 Female
##  8 South-East Asia Bangladesh  2020 Both sexes
##  9 South-East Asia Bangladesh  2020 Male
## 10 South-East Asia Bangladesh  2019 Female
## # ℹ 79 more rows
## # ℹ 1 more variable: NewHIVCases <chr>
```

Your task is to complete the loop template below so that it: - Imports each CSV in the folder - Filters to data to just the "Female" sex - Saves each filtered dataset as a CSV in your `outputs` folder

Note that in this case you do not need to store the outputs in a list, since you are importing, modifying then directly exporting each dataset.

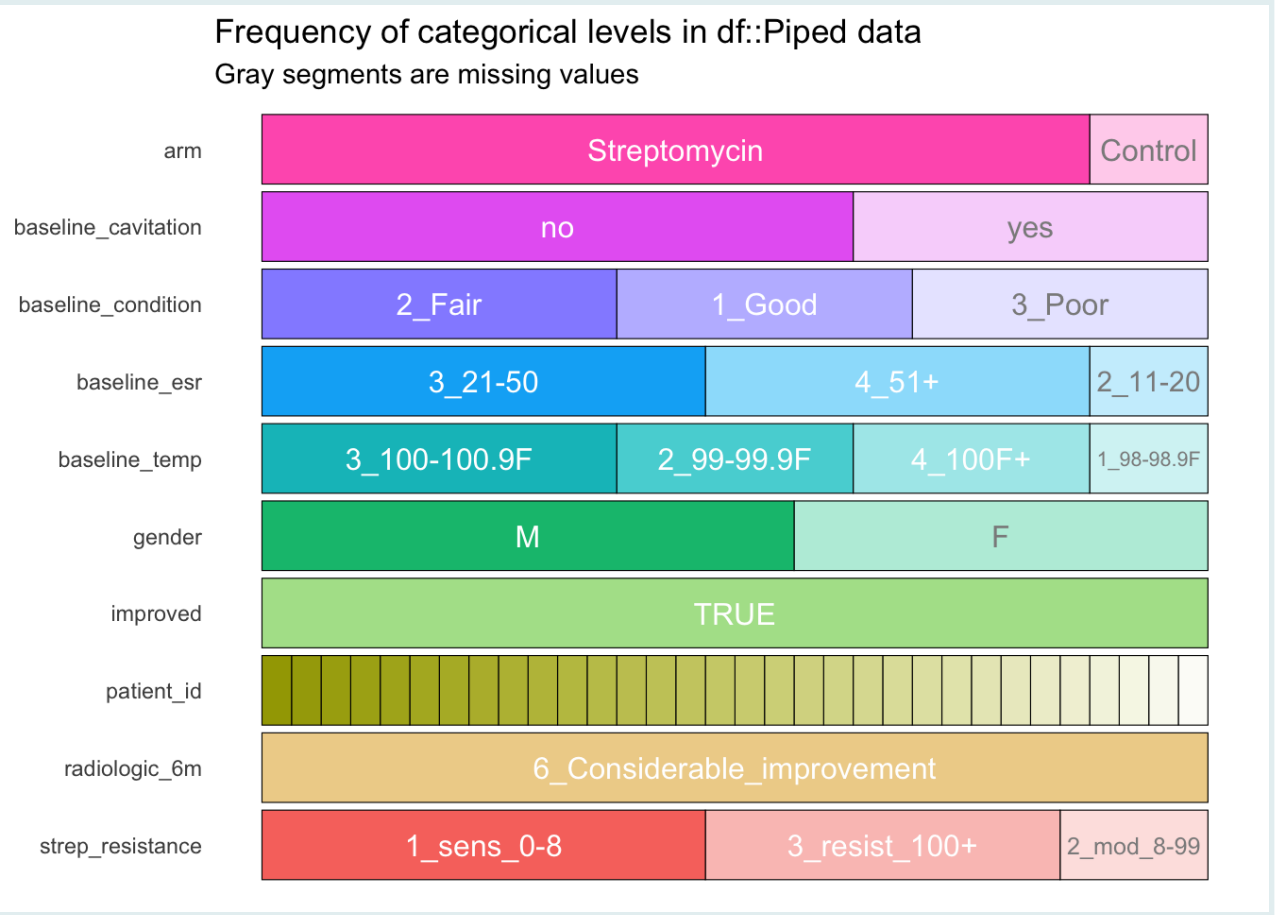

```
# List files
csv_files <- list.files(path = "data/new_hiv_infections_gho",
                        pattern = "*.csv", full.names = TRUE)

for (file in _____________) {

  # Import the data. Hint: use read_csv with the `file`
variable as the path
  hiv_dat ____________________________________________

  # Filter. Hint: use filter() and the `Sex` variable
  hiv_dat_filtered <-
_____________________________________________

  # Name output file
  # This line is done for you, but make sure you understand it
  output_file_name <-  paste0(here(), "outputs/", "Female_",
basename(file)))

  # Export.
  write_csv(hiv_dat_filtered, output_file_name)
}
```

Real Loops Application 2: Generating Multiple Plots

Another common application of loops is for generating multiple plots for different groups within a dataset. We'll use the `strep_tb` dataset from the `medicaldata` package to demonstrate this. Our aim is to create category inspection plots for each radiologic 6-month improvement group.

Let's start by creating a plot for one of the groups. We'll use `inspectdf::inspect_cat()` to generate a category inspection plot:

```
cat_plot <-
  medicaldata::strep_tb %>%
  filter(radiologic_6m == "6_Considerable_improvement") %>%
  inspectdf::inspect_cat() %>%
  inspectdf::show_plot()
cat_plot
```



This plot gives us a quick way to visualize the distribution of categories in our dataset.

Now, we want to create similar plots for each radiologic improvement group in the dataset. First, let's identify all the unique groups using the `unique` function:

```
radiologic_levels_6m <- medicaldata::strep_tb$radiologic_6m %>% unique()
radiologic_levels_6m
```

```
## [1] 6_Considerable_improvement   5_Moderate_improvement       4_No_change
## [4] 3_Moderate_deterioration     2_Considerable_deterioration 1_Death
## 6 Levels: 6_Considerable_improvement 5_Moderate_improvement ... 1_Death
```

Next, we'll initiate an empty list object where we will store the plots.

```
cat_plot_list <- vector("list", length(radiologic_levels_6m))
cat_plot_list
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
##
## [[6]]
## NULL
```

We will also set the names of the list elements to the radiologic improvement groups. This is an optional step, but it makes it easier to access specific plots later on.

```
names(cat_plot_list) <- radiologic_levels_6m
cat_plot_list
```

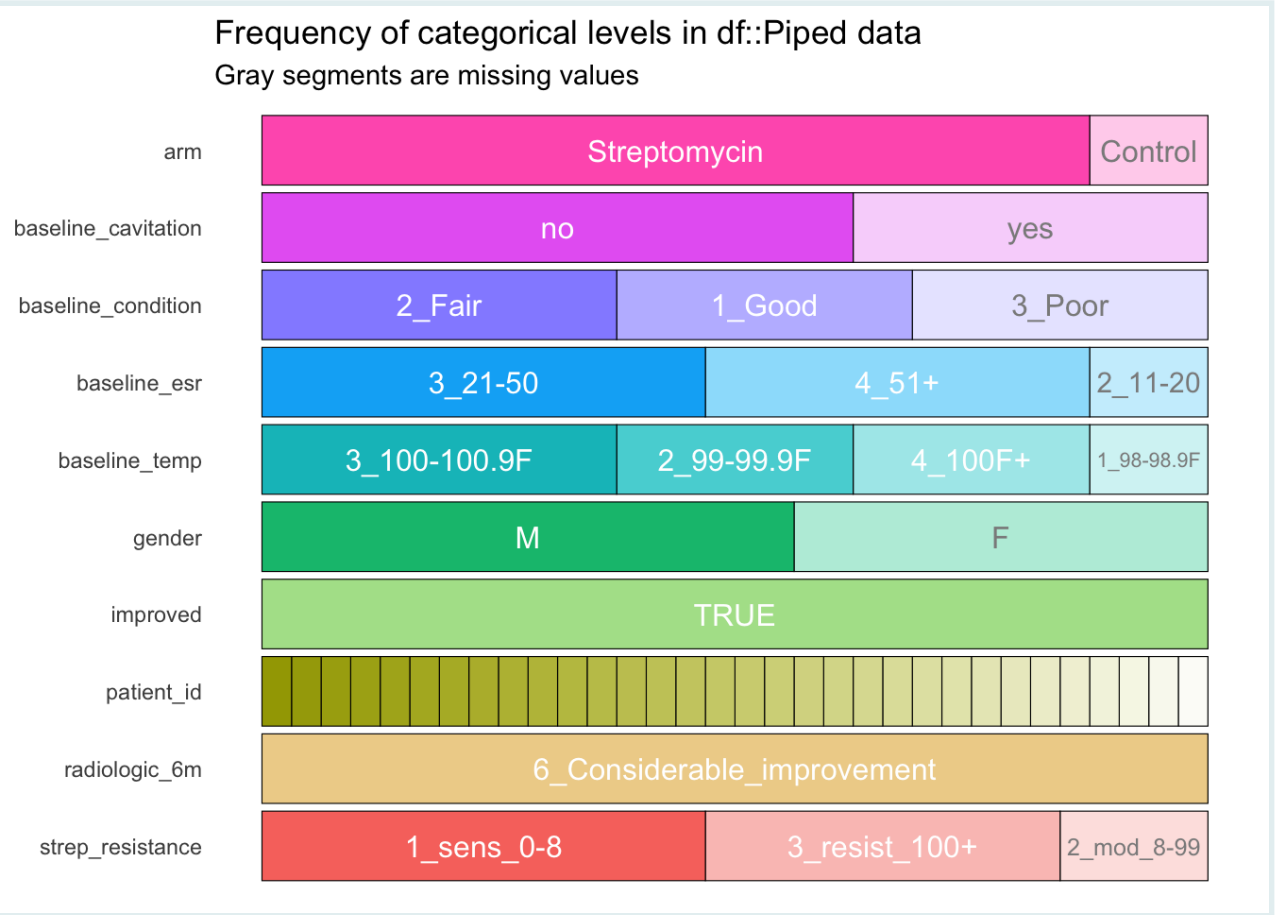
```
## $`6_Considerable_improvement`
## NULL
##
## $`5_Moderate_improvement`
## NULL
##
## $`4_No_change`
## NULL
##
## $`3_Moderate_deterioration`
## NULL
##
## $`2_Considerable_deterioration`
## NULL
##
## $`1_Death`
## NULL
```

Finally, we'll use a loop to generate a plot for each group and store it in the list:

```
for (level in radiologic_levels_6m) {

  # Generate plot for each level
  cat_plot <-
    medicaldata::strep_tb %>%
    filter(radiologic_6m == level) %>%
    inspectdf::inspect_cat() %>%
    inspectdf::show_plot()

  # Append to the list
  cat_plot_list[[level]] <- cat_plot
}
```

To access a specific plot, we can use the double bracket syntax:

```
cat_plot_list[["6_Considerable_improvement"]]
```

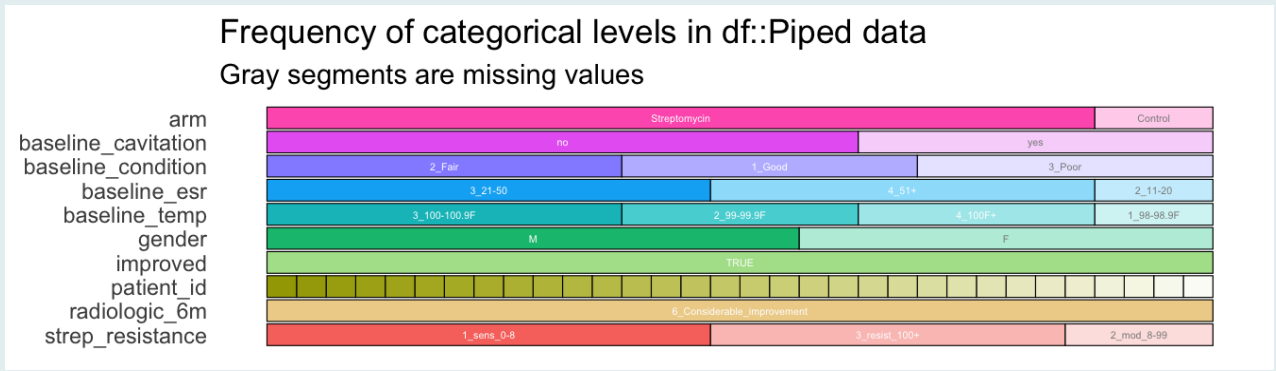


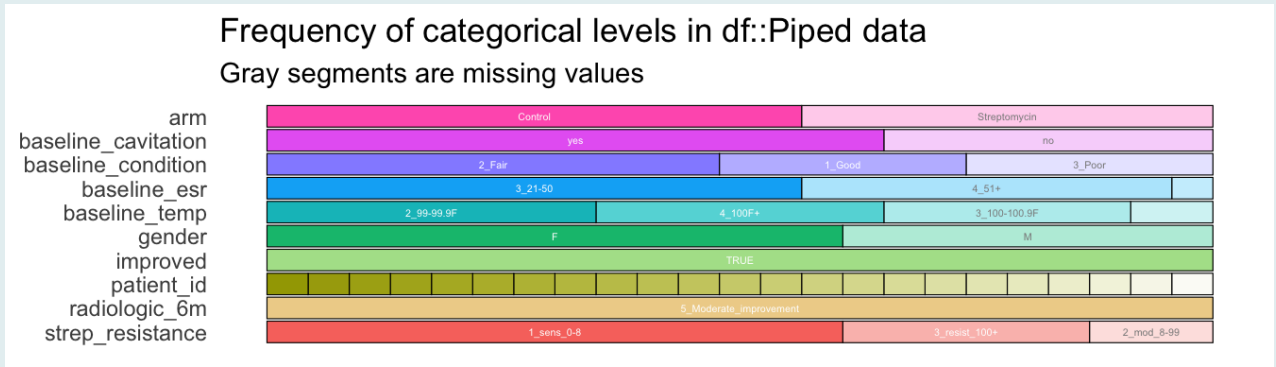
Note that in this case, the list elements are *named*, rather than just numbered. This is because we used the `level` variable as the index in the loop.

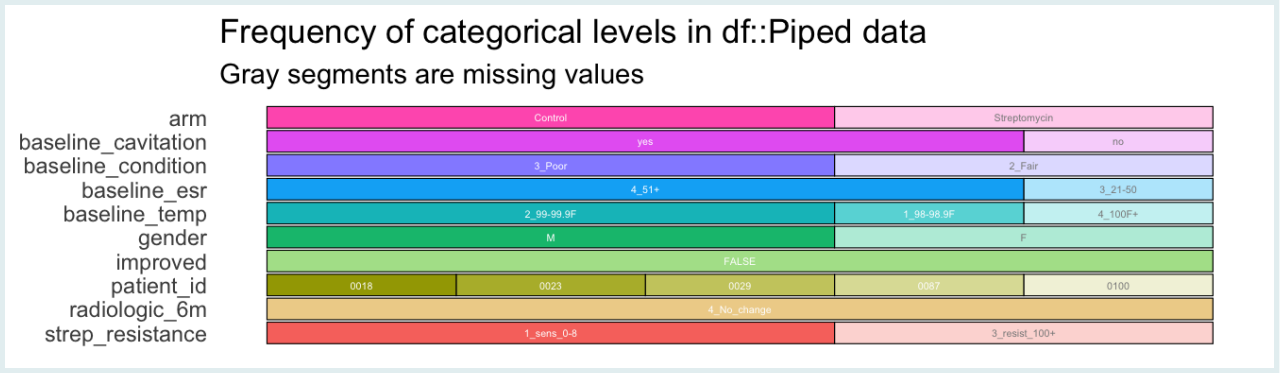
To display all plots at once, we simply call the entire list.

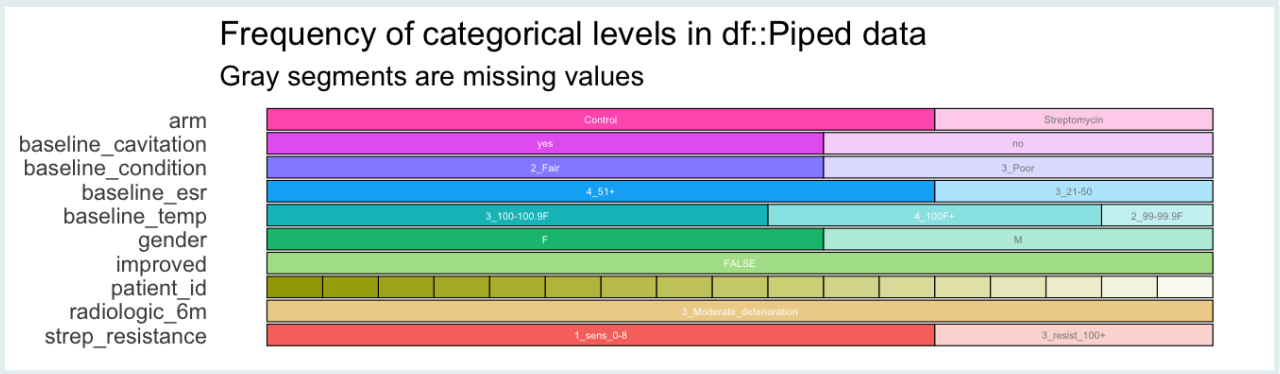
```
cat_plot_list
```

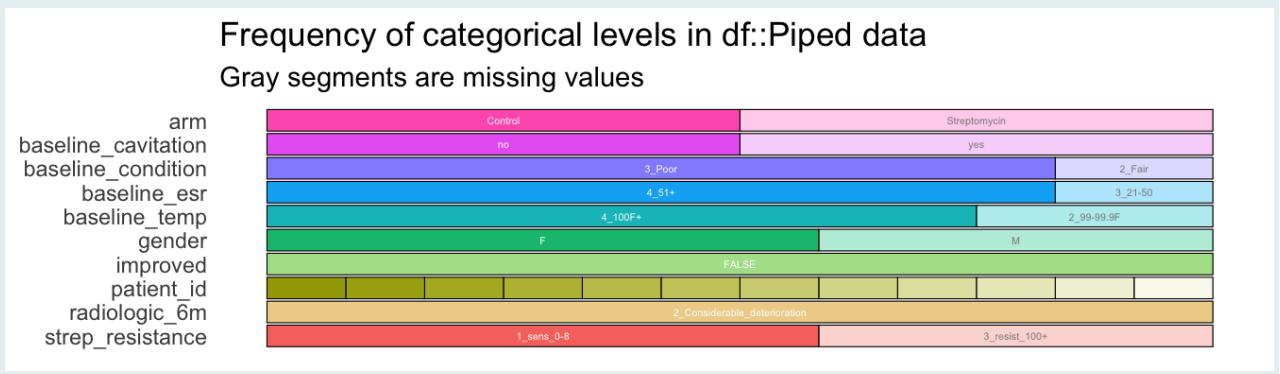
```
## $`6_Considerable_improvement`


##
## $`5_Moderate_improvement`


##
## $`4_No_change`


##
## $`3_Moderate_deterioration`


##
## $`2_Considerable_deterioration`


##
## $`1_Death`
```

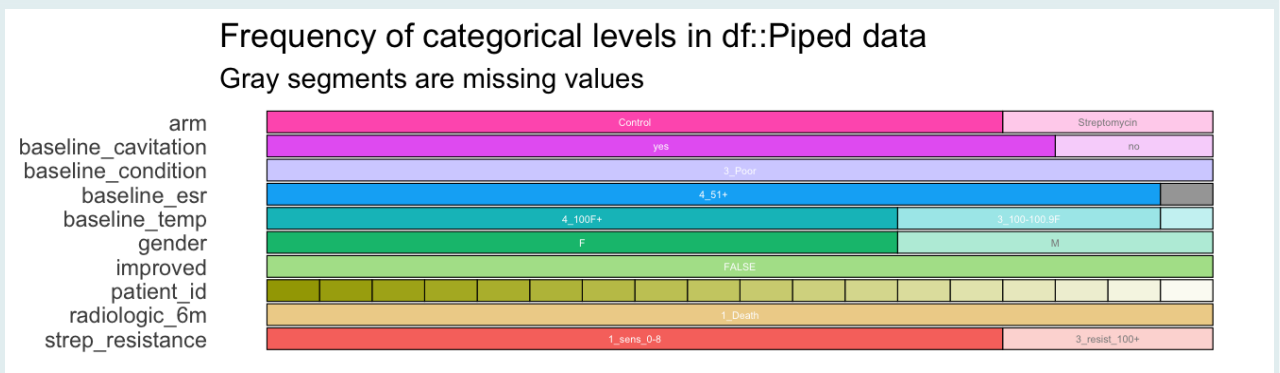












Visualizing TB Cases

First, we'll prepare the data:

```
tb_child_cases <- tidyr::who2 %>%
  transmute(country, year,
            tb_cases_children = sp_m_014 + sp_f_014 + sn_m_014
+ sn_f_014) %>%
  filter(country %in% c("Brazil", "Colombia", "Argentina",
                        "Uruguay", "Chile", "Guyana"))  %>%
  filter(year >= 2006)

tb_child_cases
```


```
## # A tibble: 48 × 3
##    country    year tb_cases_children
##    <chr>     <dbl>             <dbl>
##  1 Argentina  2006               880
##  2 Argentina  2007              1162
##  3 Argentina  2008               961
##  4 Argentina  2009               593
##  5 Argentina  2010               491
##  6 Argentina  2011               867
##  7 Argentina  2012               745
##  8 Argentina  2013                NA
##  9 Brazil     2006              2254
## 10 Brazil     2007              2237
## # ℹ 38 more rows
```

Now, fill in the blanks in the template below to create a line graph for each country using a `for` loop:


```
# Get list of countries. Hint: Use unique() on the country
column
countries <- ____________________________________________

# Create list to store plots. Hint: Initialize an empty list
tb_child_cases_plots <- vector("list", _________________)
names(tb_child_cases_plots) <- countries # Set names of list
elements

# Loop through countries
for (country in _____________) {

  # Filter data for each country
  tb_child_cases_filtered <-
_____________________________________________

  # Make plot
  tb_child_cases_plot <-
_____________________________________________

  # Append to list. Hint: Use double brackets
  tb_child_cases_plots[[country]] <- tb_child_cases_plot
}

tb_child_cases_plots
```

```
## Error: <text>:2:15: unexpected input
## 1: # Get list of countries. Hint: Use unique() on the
country column
## 2: countries <- __
##                  ^
```

Wrap Up!

In this lesson, we delved into for loops in R, demonstrating their utility from basic tasks to complex data analysis involving multiple datasets and plot generation. Despite R's preference for vectorized operations, for loops are indispensable in certain scenarios. Hopefully, this lesson has equipped you with the skills to confidently implement for loops in various data processing contexts.

Answer Key

Hours to Minutes Basic Loop

```
hours <- c(3, 4, 5) # Vector of hours

for (hour in hours) {
  minutes <- hour * 60
  print(minutes)
}
```

```
## [1] 180
## [1] 240
## [1] 300
```

Hours to Minutes Indexed Loop

```
hours <- c(3, 4, 5) # Vector of hours

for (i in 1:length(hours)) {
  minutes <- hours[i] * 60
  print(minutes)
}
```

```
## [1] 180
## [1] 240
## [1] 300
```

BMI Calculation Loop

```
weights <- c(30, 32, 35) # Weights in kg
heights <- c(1.2, 1.3, 1.4) # Heights in meters

for(i in 1:length(weights)) {
  bmi <- weights[i] / (heights[i] ^ 2)

  print(paste("Weight:", weights[i],
              "Height:", heights[i],
              "BMI:", bmi))
}
```

```
## [1] "Weight: 30 Height: 1.2 BMI: 20.8333333333333"
## [1] "Weight: 32 Height: 1.3 BMI: 18.9349112426035"
```

```
## [1] "Weight: 35 Height: 1.4 BMI: 17.8571428571429"
```

Height cm to m

```
height_cm <- c(180, 170, 190, 160, 150) # Heights in cm

height_m <- vector("numeric", length = length(height_cm))

for (i in 1:length(height_cm)) {
  height_m[i] <- height_cm[i] / 100
}
height_m
```

```
## [1] 1.8 1.7 1.9 1.6 1.5
```

Temperature Classification

```
body_temps <- c(35, 36.5, 37, 38, 39.5) # Body temperatures in Celsius
classif_vec <- vector("character", length = length(body_temps)) # character
vector

for (i in 1:length(body_temps)) {
    # Add your if-else logic here
    if (body_temps[i] < 36) {
        out <- "Hypothermia"
    } else if (body_temps[i] <= 37.5) {
        out <- "Normal"
    } else {
        out <- "Fever"
    }

    # Final print statement
    classif_vec[i] <- paste(body_temps[i], "°C is", out)
}
classif_vec
```

```
## [1] "35 °C is Hypothermia" "36.5 °C is Normal"    "37 °C is Normal"
"38 °C is Fever"
## [5] "39.5 °C is Fever"
```

File Properties

```
# Assuming the path and file structure are correct
csv_files <- list.files(path = "data/new_hiv_infections_gho",
                        pattern = "\\.csv$", full.names = TRUE)

data_frames_list <- vector("list", length = length(csv_files))

for (i in 1:length(csv_files)) {

  path <- csv_files[i]
  country_name <- tools::file_path_sans_ext(basename(path))

  size <- file.size(path)
  date <- file.mtime(path)

  hiv_dat_row <- tibble(country = country_name, size = size, date = date)

  data_frames_list[[i]] <- hiv_dat_row
}

hiv_file_info_final <- bind_rows(data_frames_list)
hiv_file_info_final
```

```
## # A tibble: 9 × 3
##   country                          size date
##   <chr>                           <dbl> <dttm>
## 1 Bangladesh                       6042 2023-12-11 17:34:28
## 2 Brazil                           5946 2023-12-11 17:34:28
## 3 Democratic_Republic_of_the_Congo 8028 2023-12-11 17:34:28
## 4 Egypt                            6181 2023-12-11 17:34:28
## 5 Ethiopia                         5754 2023-12-11 17:34:28
## 6 Indonesia                        6621 2023-12-11 17:34:28
## 7 Iran_(Islamic_Republic_of)       8037 2023-12-11 17:34:28
## 8 Philippines                      6321 2023-12-11 17:34:28
## 9 Viet_Nam                         6230 2023-12-11 17:34:28
```

Data Filtering Loop

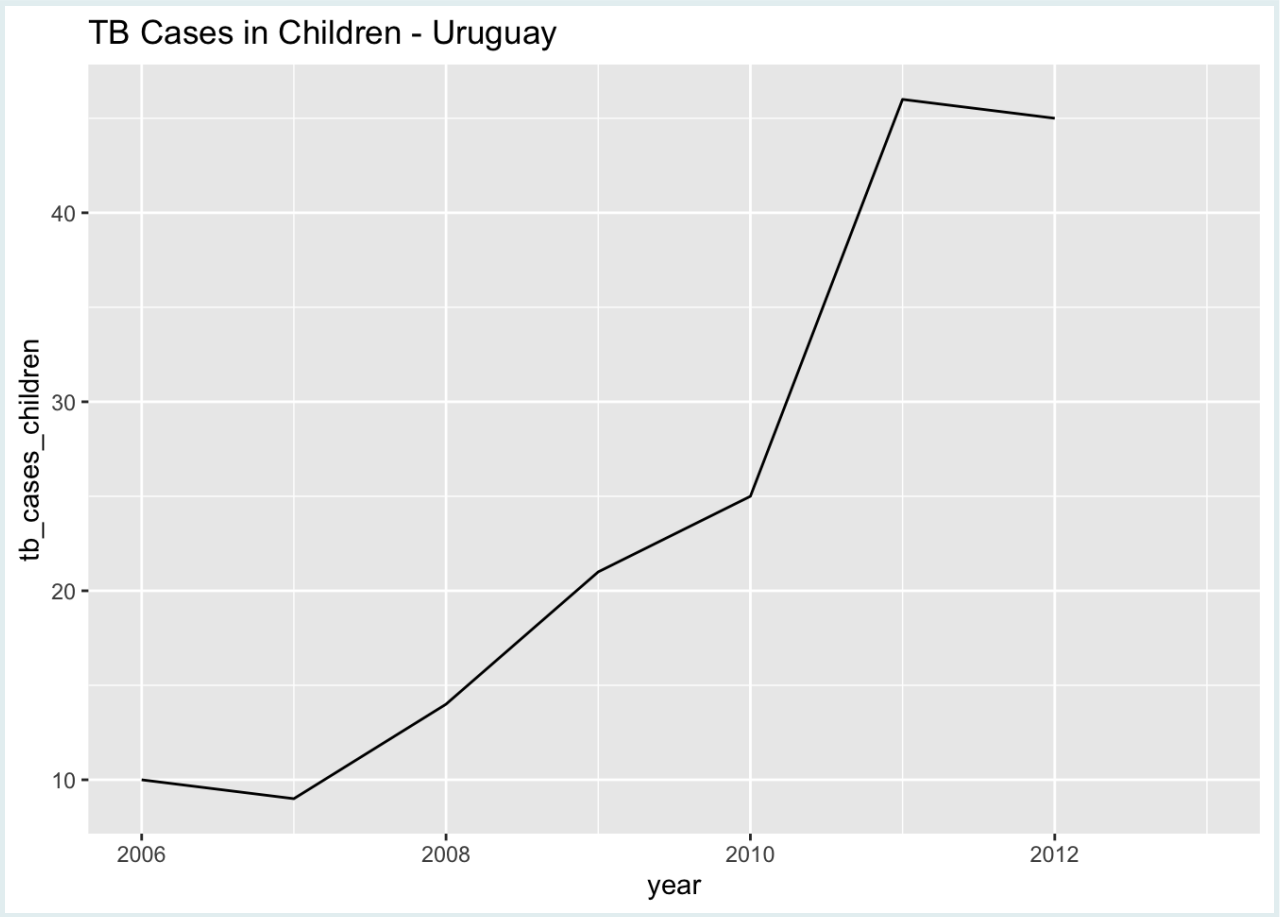
```
csv_files <- list.files(path = "data/new_hiv_infections_gho",
                        pattern = "*.csv", full.names = TRUE)

for (file in csv_files) {
  hiv_dat <- read_csv(file)

  hiv_dat_filtered <- hiv_dat %>% filter(Sex == "Female")

  output_file_name <- paste0(here(), "/outputs/", "Female_", basename(file))

  write_csv(hiv_dat_filtered, output_file_name)
}
```

```
## Rows: 89 Columns: 5
## ── Column specification
──────────────────────────────────────────────────────────────────────────────
## Delimiter: ","
## chr (4): Continent, Country, Sex, NewHIVCases
## dbl (1): Year
##
## 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.


## Error: Cannot open file for writing:
## *
'/Users/kendavidn/Dropbox/tgc_github_projects/fdar_staging/FDAR_EN_loops/outputs/Female_E
```

Visualizing TB Cases

```
# Assuming tb_child_cases is a dataframe with the necessary columns
countries <- unique(tb_child_cases$country)

# Create list to store plots
tb_child_cases_plots <- vector("list", length(countries))
names(tb_child_cases_plots) <- countries

# Loop through countries
for (countryname in countries) {

  # Filter data for each country
  tb_child_cases_filtered <- filter(tb_child_cases, country == countryname)

  # Make plot
  tb_child_cases_plot <- ggplot(tb_child_cases_filtered, aes(x = year, y =
tb_cases_children)) +
    geom_line() +
    ggtitle(paste("TB Cases in Children -", countryname))

  # Append to list
  tb_child_cases_plots[[countryname]] <- tb_child_cases_plot
}

tb_child_cases_plots[["Uruguay"]]
```

```
## Warning: Removed 1 row containing missing values (`geom_line()`).
```



Contributors

The following team members contributed to this lesson:

SABINA RODRIGUEZ VELÁSQUEZ

Project Manager and Scientific Collaborator, The GRAPH Network
Infectiously enthusiastic about microbes and Global Health

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

References

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

- Barnier, Julien. "Introduction à R et au tidyverse." https://juba.github.io/tidyverse
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science." https://r4ds.had.co.nz/
- Wickham, Hadley; Grolemund, Garrett. "R for Data Science (2e)." https://r4ds.hadley.nz/

Data Cleaning Pipeline 1: Data Diagnostics

Introduction

Data cleaning is the process of transforming raw, "messy" data into reliable data that can be properly analyzed. This entails identifying **inaccurate**, **incomplete**, or **improbable** data points and resolving data inconsistencies or errors, as well as renaming variable names to make them more clear and simple to manipulate.

Data cleaning tasks can often be tedious and time-consuming. A common joke among data analysts goes "80% of data science is cleaning the data and the remaining 20% is complaining about cleaning the data." But data cleaning is an essential step of the data analysis process. A little bit of cleaning at the start of the data analysis process will go a long way to improving the quality of your analyses and the ease with which these analyses can be done. And a range of packages and functions in R can significantly simplify the data cleaning process.

In this lesson, we will begin to look at a typical data cleaning pipeline in R. The cleaning steps covered here are unlikely to be exactly what is needed for your own datasets, but they will certainly be a good starting point.

Let's get started!

Learning objectives

- You can list typical operations involved in data cleaning process
- You can diagnose dataset issues that warrant data cleaning through functions such as:
    - `visdat::vis_dat()`
    - `skimr::skim()`
    - `inspectdf::inspect_cat()`
    - `inspectdf::inspect_num()`
    - `gtsummary::tbl_summary()`
    - `DataExplorer::create_report()`

Packages

The packages loaded below will be required for this lesson:

```
if(!require("pacman")) install.packages("pacman")
pacman::p_load(visdat,
               skimr,
               inspectdf,
               gtsummary,
               DataExplorer,
               tidyverse)
```

Dataset

The primary dataset we will use in this lesson is from a study conducted in three healthcare centres in Zambezia, Mozambique. The study investigated individual factors associated with time to non-adherence to HIV care and treatment services. For the sake of this lesson, we will only be looking at a modified subset of the full dataset.

The full dataset can be obtained from Zenodo, and the paper can be viewed here.

Let's take a look at this dataset:

```
non_adherence <- read_csv(here("data/non_adherence_moz.csv"))
```

```
non_adherence
```

```
## # A tibble: 5 × 15
##   patient_id District `Health unit` Sex   Age_35
##        <dbl>    <dbl>         <dbl> <chr> <chr>
## 1      10037        1             1 Male  over 35
## 2      10537        1             1 F     over 35
## 3       5489        2             3 F     Under 35
## 4       5523        2             3 Male  Under 35
## 5       4942        2             3 F     over 35
## # ℹ 10 more variables: `Age at ART initiation` <dbl>,
## #   Education <chr>, Occupation <chr>, …
```

The first step of data cleaning will be to explore this dataset in order to identify potential issues that warrant cleaning. This preliminary step is sometimes called "exploratory data analysis" or EDA.

Let's take a look at a few simple EDA commands in R that will help you identify possible data errors and inconsistencies.

Visualizing missing data with `visdat::vis_dat()`

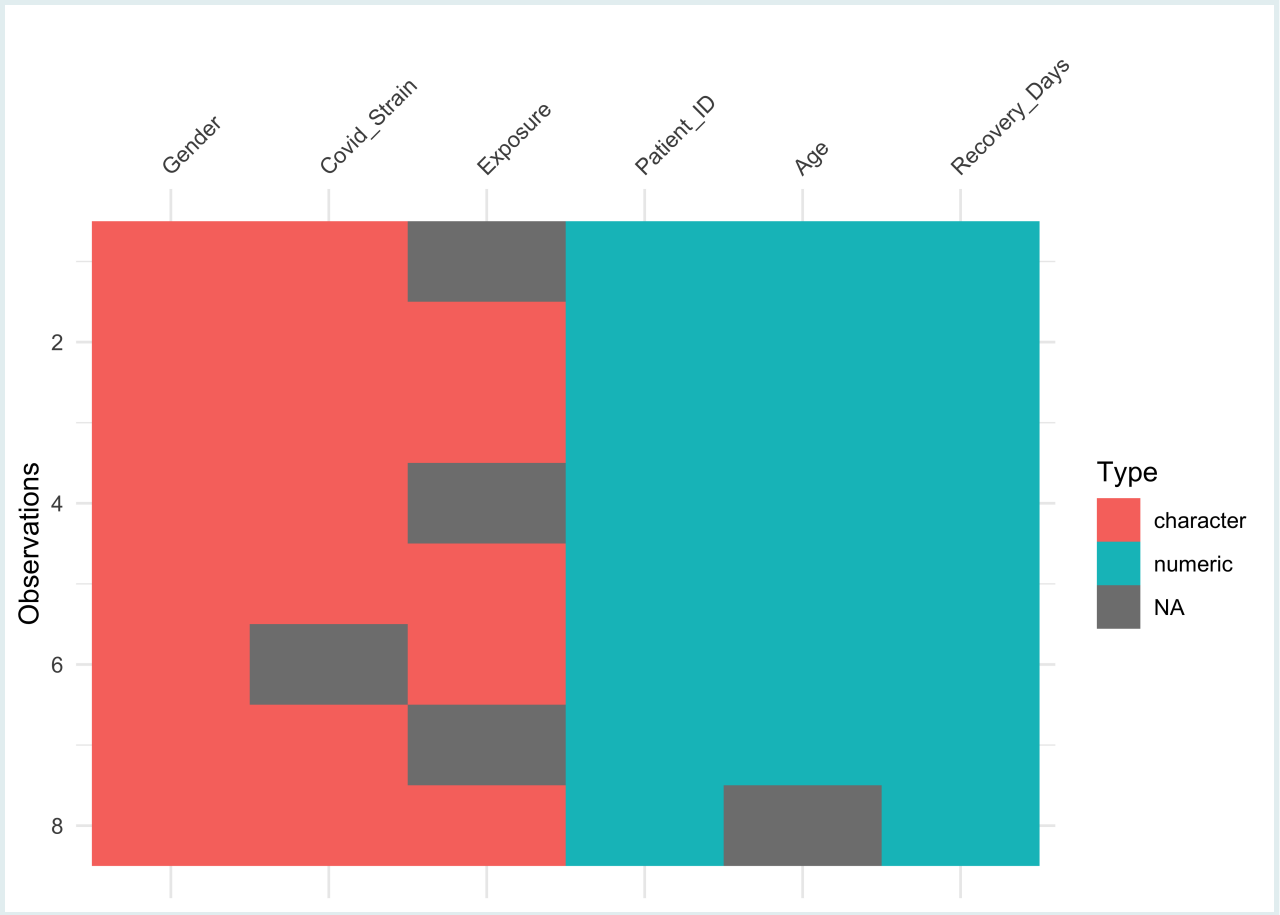
The `vis_dat()` function from the `visdat` package is a wonderful way to quickly visualize data types and missing values in a dataset. It creates a plot that shows a "zoomed out" spreadsheet view of your data frame: each row in the dataframe is represented by a single line on the plot.

Let's try it out with small mock dataset first to get an idea of how it works. Copy the following code to create a dataframe of 8 patients and their COVID-19 diagnosis and recovery information. As you can see below, some patients have missing information, represented as NA.

```
covid_pat <- tribble(
  ~Patient_ID, ~Age, ~Gender,  ~Covid_Strain, ~Exposure,    ~Recovery_Days,
  1,           25,   "Male",   "Alpha",        NA,          10,
  2,           32,   "Female", "Delta",       "Hospital",   15,
  3,           45,   "Male",   "Beta",        "Travel",      7,
  4,           19,   "Female", "Omicron",      NA,          21,
  5,           38,   "Male",   "Alpha",       "Unknown",    14,
  6,           55,   "Female",  NA,           "Community",  19,
  7,           28,   "Female", "Omicron",      NA,           8,
  8,           NA,   "Female", "Omicron",     "Travel",     26
)
covid_pat
```

Now, let's use the function `vis_dat()` on our dataframe to get visual representation of the data types and missing values.

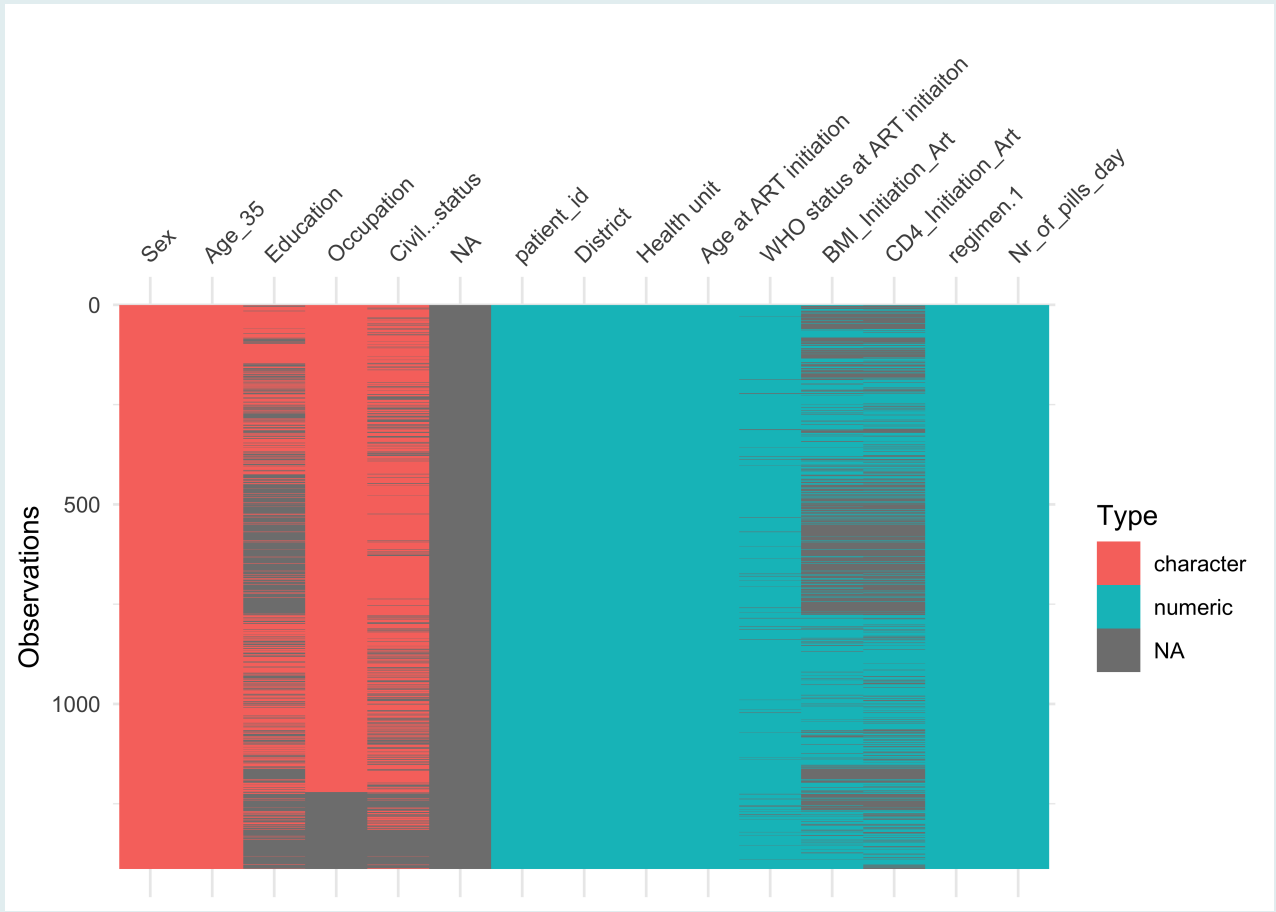
```
vis_dat(covid_pat)
```



That looks great! Each row in our dataframe is represented by a single line in the plot and different colours are used to illustrate the character (pink) and numeric (blue) variables, as well as missing values (grey). From this plot, we can tell that quite a few patients in our dataset are missing data for the `Exposure` variable.

Now let's turn back to our real-world dataset which is much larger and messier. Large real-world datasets may exhibit intricate patterns in the data that are difficult to spot without visualization, so using functions such as `vis_dat()` can be particularly useful. Let's try it out now!

```
vis_dat(non_adherence)
```



Great! From this view we can already see some problems:

- there seems to be a completely empty column in the data (the column `NA` which is fully gray)
- several variables have a lot of missing values (such as `Education`, `BMI_Initation_Art`, and `CD4_Initation_ART`)
- the names of some variables are unclear/unclean (e.g., `Age at ART initation` and `WHO status at ART initiaion` have whitespaces in their names and `Civil...status` and `regimen.1` have special characters, `.`)

In the next lesson, we will try to remedy these and other issues as best as possible during the data cleaning process. But for now, the goal is that we understand the functions used to identify them. So now that we have a solid grasp on how to visualize missing data with `vis_dat()`, let's take a look at another package and function that can help us generate summary statistics of our variables!

Spotting data issues with `vis_dat()`

The following dataset was adapted from a study that investigated missed opportunities for HIV testing among patients newly presenting for HIV care at a Swiss university hospital. The full dataset can be found here.

```
## Rows: 201 Columns: 16
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## chr (11): sex, clage, origine, adquired, chronic, acute, latepresent,
reason...
## dbl  (4): cd4, cd4_category, numberconsult, missed_ops
## lgl  (1): NaN.
##
## 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.
```

Use the `vis_dat()` function to get a visual representation of the data. What potential issues can you spot based on the output?

Generating summary statistics with `skimr::skim()`

The `skim()` function from the `skimr` package provides a console-based overview of the data frame and a summary of every column (by class/type). Let's try it on our `covid_pat` mock data first to get an idea of how it works. First, as a reminder, here is our `covid_pat` dataframe:

```
covid_pat
```

Great, now let's run the `skim()` function!

```
skimr::skim(covid_pat)
```

Table 1: Data summary

Name	covid_pat
Number of rows	8
Number of columns	6
_____	
Column type frequency:	
character	3
numeric	3

Group variables None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
Gender	0	1.00	4	6	0	2	0
Covid_Strain	1	0.88	4	7	0	4	0
Exposure	3	0.62	6	9	0	4	0

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
Patient_ID	0	1.00	4.50	2.45	1	2.75	4.5	6.25	8	
Age	1	0.88	34.57	12.39	19	26.50	32.0	41.50	55	
Recovery_Days	0	1.00	15.00	6.68	7	9.50	14.5	19.50	26	

Amazing! As we can see, this function provides:

- An overview of the rows and columns of the dataframe
- The data type for each variable
- `n_missing`, the number of missing values for each variable
- `complete_rate`, the completeness rate for each variable
- A set of summary statistics: the mean, standard deviation and quartiles for numerical variables; and the frequency and proportions for categorical variables
- Spark histograms for the numerical variables

Now we can tell why this is so valuable with large dataframes! Let's return to our `non-adherence` dataset and run the `skim()` function on it.

```
non_adherence
```

```
skimr::skim(non_adherence)
```

Table 2: Data summary

Name	non_adherence
Number of rows	1413
Number of columns	15
_____	
Column type frequency:	
character	5
logical	1
numeric	9
_____	
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
Sex	0	1.00	1	4	0	2	0
Age_35	0	1.00	7	8	0	2	0
Education	776	0.45	4	10	0	5	0
Occupation	193	0.86	4	22	0	50	0
Civil...status	412	0.71	6	12	0	4	0

Variable type: logical

skim_variable	n_missing	complete_rate	mean	count
NA	1413	0	NaN	:

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100
skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100
patient_id	0	1.00	7364.75	3312.95	147.0	4705.00	7355.0	10098.0	24377.0
District	0	1.00	1.35	0.48	1.0	1.00	1.0	2.0	2.0
Health unit	0	1.00	1.87	0.90	1.0	1.00	2.0	3.0	3.0
Age at ART initiation	0	1.00	32.07	9.51	15.1	25.20	30.4	36.8	75.3
WHO status at ART initiaiton	45	0.97	1.80	0.98	1.0	1.00	1.0	3.0	4.0
BMI_Initiation_Art	588	0.58	20.37	3.54	9.0	18.12	20.1	22.2	52.0
CD4_Initiation_Art	674	0.52	398.19	243.64	3.0	233.00	343.0	538.0	1401.0
regimen.1	0	1.00	4.64	1.54	1.0	3.00	6.0	6.0	7.0
Nr_of_pills_day	0	1.00	1.51	0.59	1.0	1.00	1.0	2.0	3.0

From this output, we can identify some potential issues with our data:

- we can confirm that the column NA, which you saw from the `vis_dat()` output, is indeed completely empty: it has a `complete_rate` of 0
- the distribution for `Age at ART initation` is skewed

Genertaing summary stats with `skim()`

Use `skim()` to obtain a detailed overview of the `missed_ops` dataset.

Great! Now that we know how to generate a short report of summary statistics for our variables. In the next section, we'll learn about a couple useful functions from the `inspectdf` package that allows us to visualize different summary stats.

Visualizing summary statistics with functions from `inspectdf`

While the `skimr::skim()` function gives you variable summaries in the console, you may sometimes want variables summaries in richer graphical form instead. For this, the functions `inspectdf::inspect_cat()` and `inspectdf::inspect_num()` can be used.

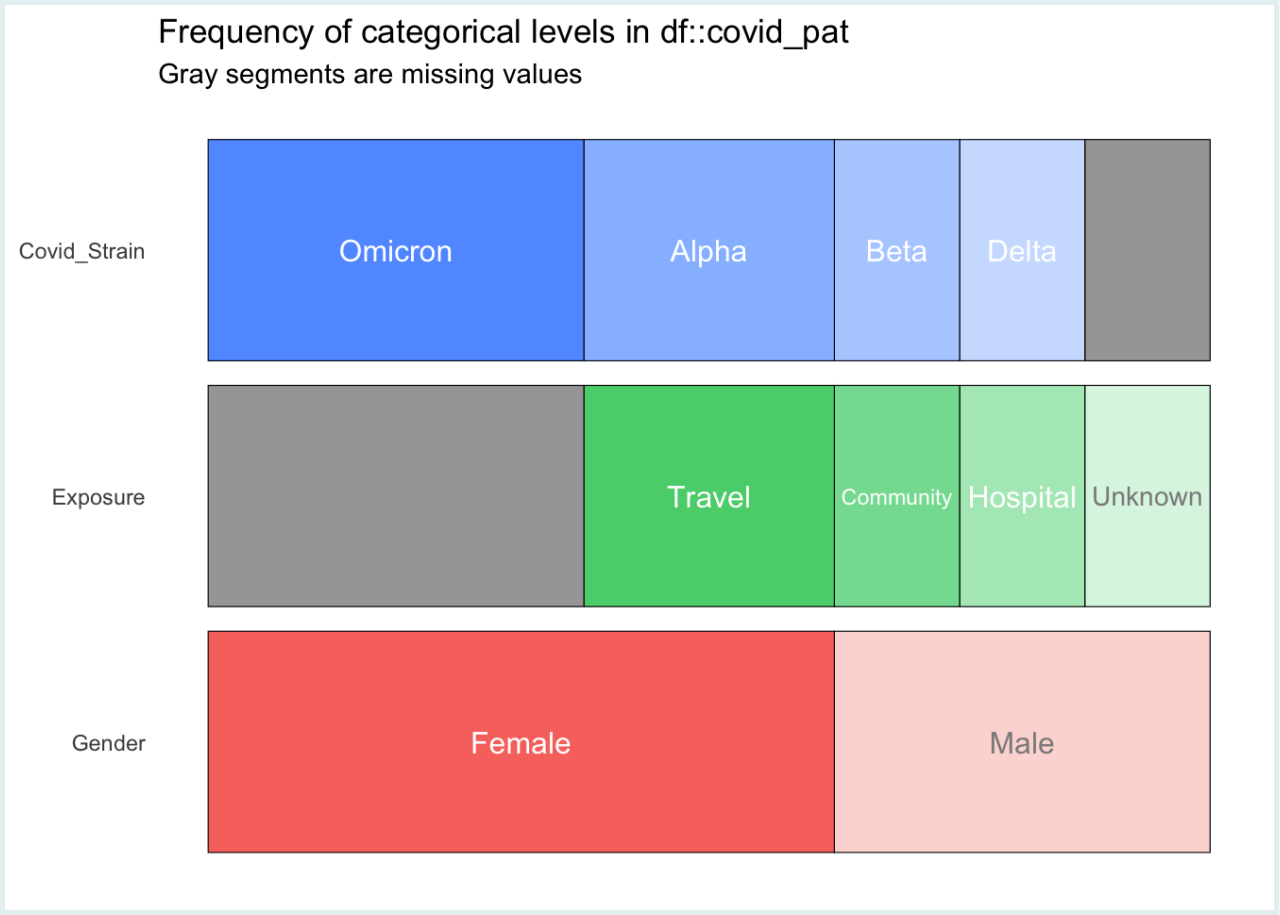
If you run `inspect_cat()` on a data object, you get a tabular summary of the categorical variables in the dataset (the important information is hidden in the `levels` column). Let's try it out on the `covid_pat` dataset first. As a reminder, here is our dataset:

```
covid_pat
```

```
inspect_cat(covid_pat)
```

The magic happens when you run `show_plot()` on the result from `inspect_cat()`:

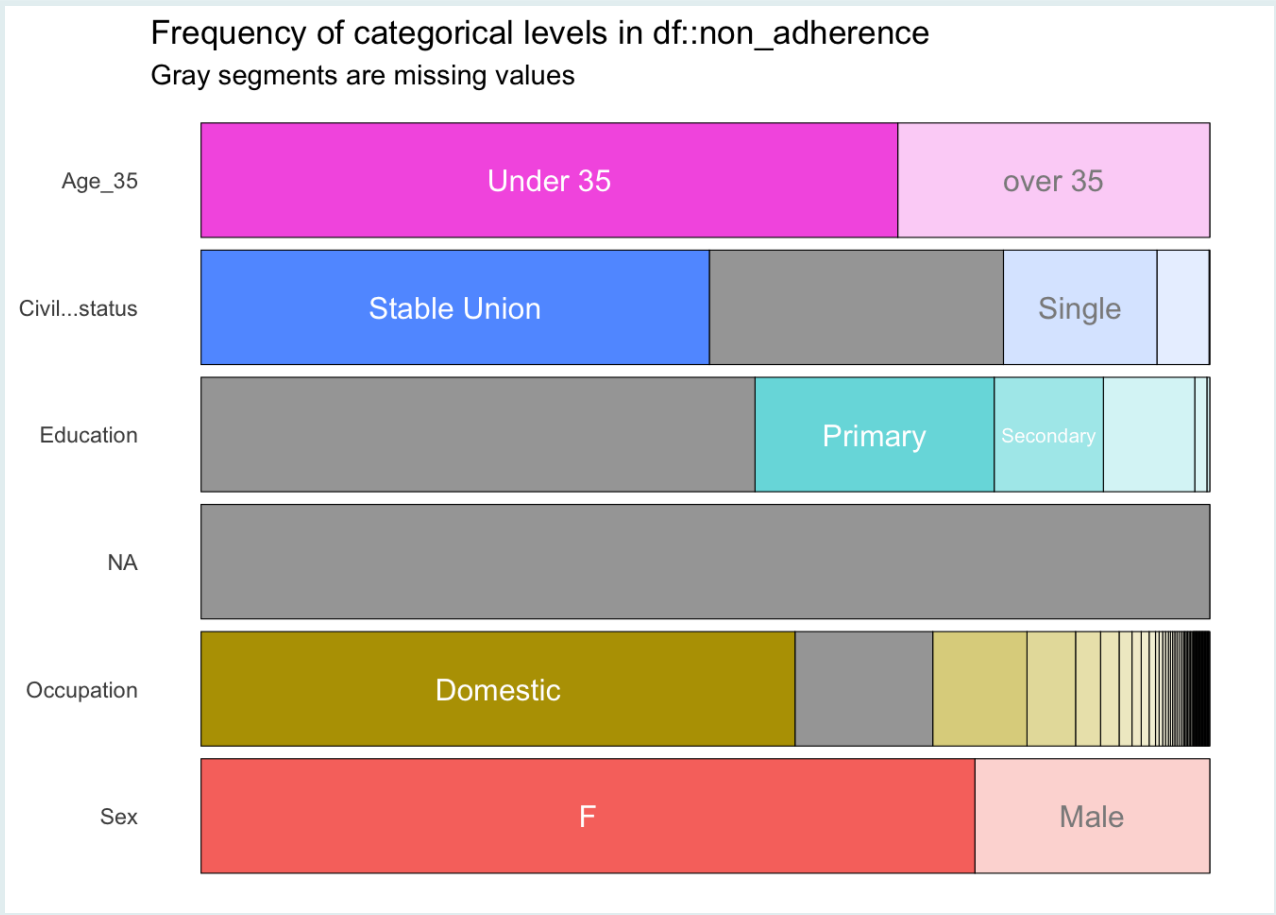
```
inspect_cat(covid_pat) %>%
  show_plot()
```



That looks great! You get a nice summary figure showing the distribution of categorical variables! Variable levels are also nicely labelled (if there is sufficient space to show a label).

Now, let's try it out on our `non_adherence` dataset:

```
inspect_cat(non_adherence) %>%
  show_plot()
```



From here you can observe some issues with a few categorical variables:

- The variable level `Under 35` is capitalized, whereas `over 35` is not. It could be worth standardizing this.
- The variable sex has the levels `F` and `Male`. This too could be worth standardizing.
- As we have previously seen, `NA` is completely empty

Spotting data issues with `inspect_cat()`

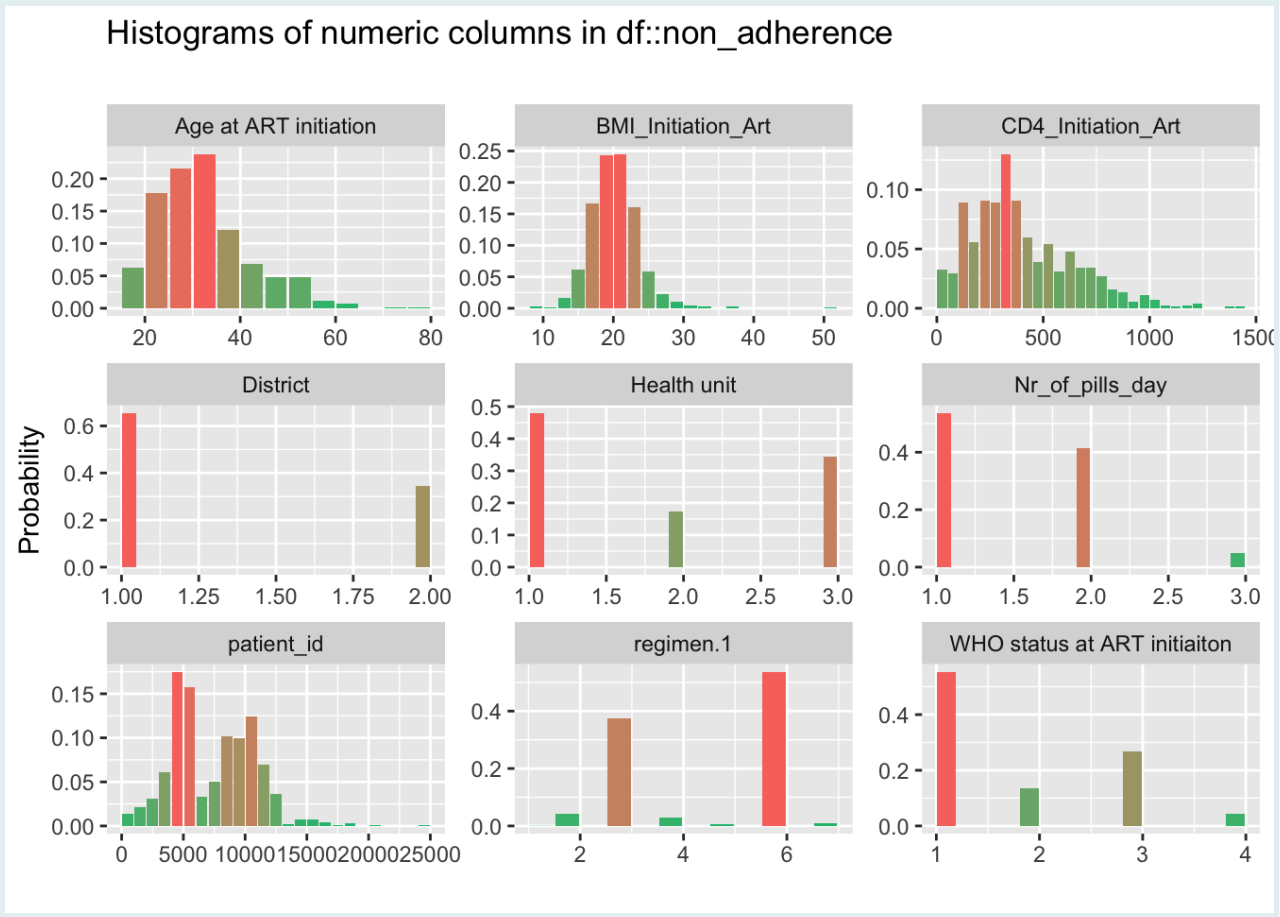
Complete the following code to obtain a visual summary of the categorical variables in the `missed_op` dataset.

```
inspect___() %>%
  __________
```

How many potential data issues can you spot?

Similarly, you can obtain a summary plot for the numerical variables in the dataset with `inspect_num()`. Let's run this on our `non_adherence` dataset:

```
inspect_num(non_adherence) %>%
  show_plot()
```



From this output, you can notice that many variables which should be factor variables are coded as numeric. In fact, the only true numeric variables are `Age at ART initiation`, `BMI_Initiation_ART`, `CD4_Initation_ART`, and `Nr_of_pills_day`. We will fix these issues in the next lesson when we move on to data cleaning. For now though, let's take a look at our another function that's particularly helpful for categorical varibales!

Variable types with `inspect_num()`

Use `inspect_num` to create a histograms of your numeric variables in the `missed_op` dataset. Are the numeric variable types correct?

Exploring categorical variable levels with `gtsummary::tbl_summary()`

While the `inspect_cat()` function is useful for a graphical overview of categorical variables, it doesn't provide frequency counts or percentages for the various levels. For this, the `tbl_summary()` from the `gtsummary` package is particularly helpful! The output

is particularly long so we'll look at the tibble form and show a photo of the important part for our dataset. You can explore the whole output as you code along at home.

Let's try it out on our `non_adherence` dataset:

```
gtsummary::tbl_summary(non_adherence) %>%
  as_tibble()
```

```
## # A tibble: 104 × 2
##    `**Characteristic**` `**N = 1,413**`
##    <chr>                <chr>
##  1 patient_id           7,355 (4,705, 10,098)
##  2 District             <NA>
##  3 1                    925 (65%)
##  4 2                    488 (35%)
##  5 Health unit          <NA>
##  6 1                    678 (48%)
##  7 2                    247 (17%)
##  8 3                    488 (35%)
##  9 Sex                  <NA>
## 10 F                    1,084 (77%)
## # ℹ 94 more rows
```

That looks great! As we can see it outputs a summary of the frequency and percentages for categorical variables and the median and IQR for numeric variables.

Below is a photo of part of the output where we can see additional issues with our data that wasn't clear using the `inspect_cat()` function. Some values from our `Occupation` variable are capitalized, whereas others are all in lower case.

Office worker	1 (<0.1%)
Police	3 (0.2%)
professor	11 (0.9%)
Professor	35 (2.9%)
Receptionist	1 (<0.1%)
Retired	2 (0.2%)
rock cutter	9 (0.7%)

This means that R doesn't recognize them as being the same value which would be problematic during analysis. We'll fix these errors in the next lesson, for now though let's move on to our last function for exploratory data analysis!

The `tbl_summary()` function is not appropriate for checking variable names. As you may have noticed, all whitespaces were replaced by a period. For example, `Age at ART initation` becomes `Age.at.ART.initation`.

Spotting data issues with `tbl_summary()`

Use `tbl_summary()` to output a summary of your `missed_ops` dataset. Can you identify any additional data issues?

Creating data reports with `DataExplorer::create_report()`

Finally, the `create_report()` function from the `DataExplorer` package creates a full data profile of a provided data frame: an HTML file with basic statistics and distribution visualizations.

Let's run this function on the `non_adherence` dataframe and observe its output. Note that it may take a while to run. If it takes too long, you can run it on a dataset sample, rather than the entire dataset.

```
create_report(non_adherence)
```

As you can see, the report is quite extensive. We won't go over all of the outputs from this data report because many of the graphical outputs are the same as those we have seen with previous functions! However, some new outputs include:

- QQ plot to assess the normality of numeric variables
- Correlation analysis (when there are sufficient complete rows)
- Principal component analysis (when there are sufficient complete rows)

Feel free to explore the package documentation on your own time.

Data report with `create_report()`

Create a data report for your `missed_ops` data using the `create_report()` function!

Wrap up!

By familiarizing ourselves with the data, we have been able to identify some potential problems that may need to be addressed before the data are used in an analysis.

And as you have seen, the actual code needed to do this data exploration is very little; other R developers have done the difficult work for us by building amazing packages to quickly scan datasets and identify issues.

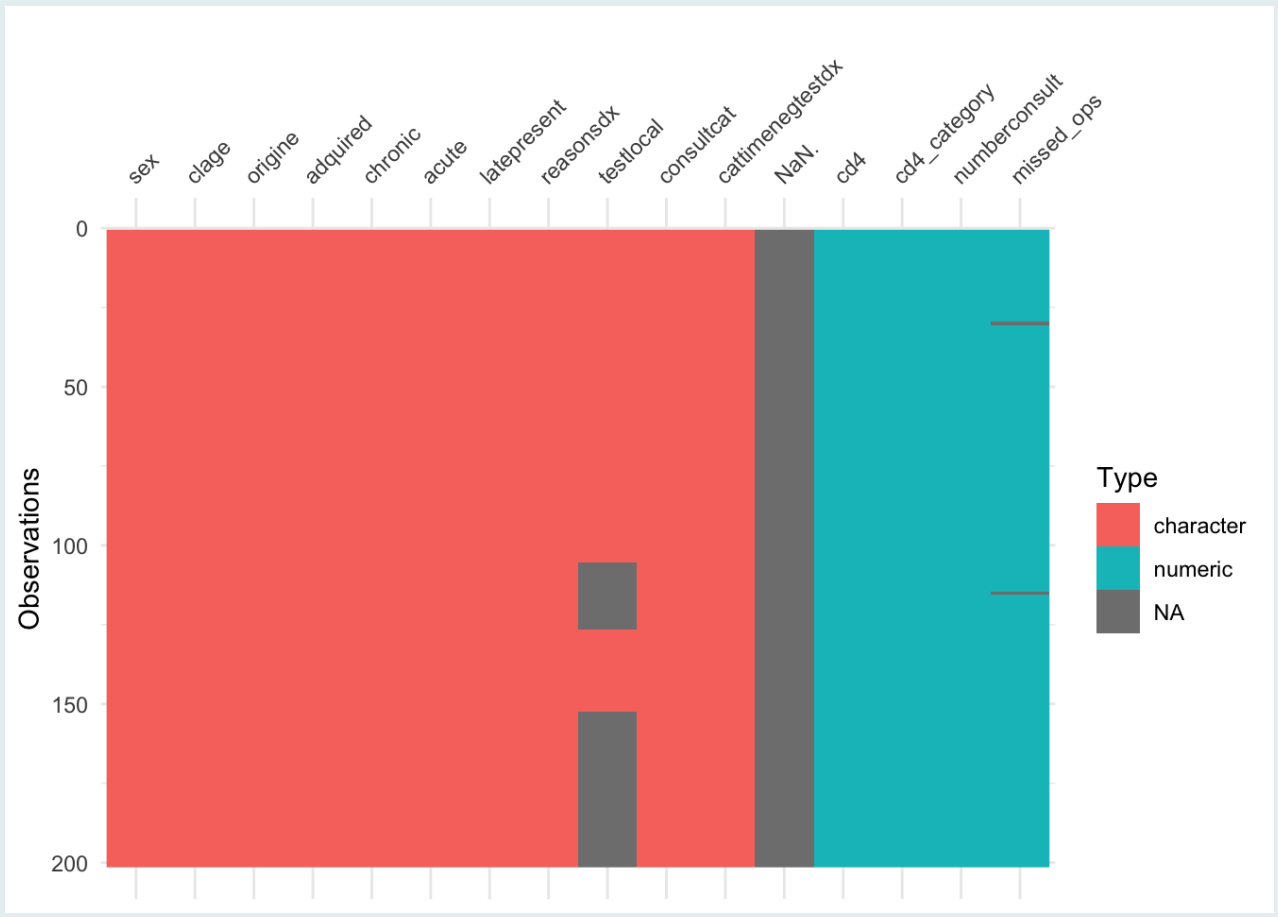
From the next lesson, we will begin to take on these identified issues one by one, starting with the problem of inconsistent, messy variable names.

See you in the next lesson!

Answer Key

Q: Spotting data issues with `vis_dat()`

```
vis_dat(missed_ops)
```



- The column NaN is completely empty

Q: Genertaing summary stats with `skim()`

```
skim(missed_ops)
```

Table 3: Data summary

Name	missed_ops
Number of rows	201
Number of columns	16
_____	
Column type frequency:	
character	11
logical	1
numeric	4
_____	
Group variables	None

Variable type: character

skim_variable	n_missing	complete_rate	min	max	empty	n_unique	whitespace
sex	0	1.00	1	6	0	3	0
clage	0	1.00	3	5	0	3	0
origine	0	1.00	6	18	0	3	0
adquired	0	1.00	3	12	0	4	0
chronic	0	1.00	2	3	0	2	0
acute	0	1.00	3	9	0	2	0
latepresent	0	1.00	2	3	0	2	0
reasonsdx	0	1.00	8	46	0	8	0
testlocal	70	0.65	19	22	0	3	0
consultcat	0	1.00	24	26	0	3	0
cattimenegtestdx	0	1.00	23	33	0	3	0

Variable type: logical

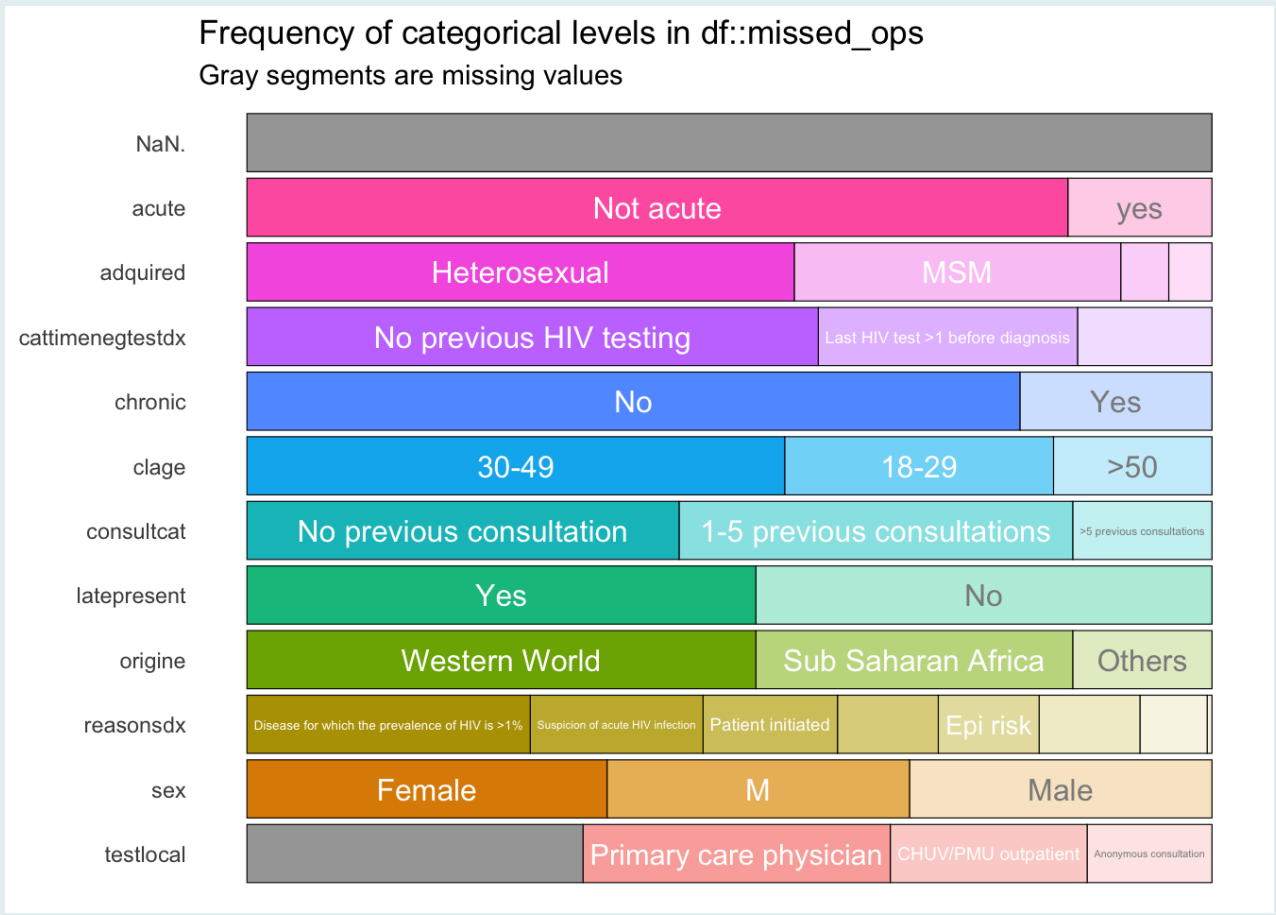
skim_variable	n_missing	complete_rate	mean	count
NaN.	201	0	NaN	:

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
cd4	0	1.00	342.13	261.68	2	147	293	452	1261	
cd4_category	0	1.00	2.27	1.12	1	1	2	3	4	
numberconsult	0	1.00	1.98	2.90	0	0	1	3	21	
missed_ops	2	0.99	1.83	3.01	0	0	0	3	17	

Q: Spotting data issues with `inspect_cat()`

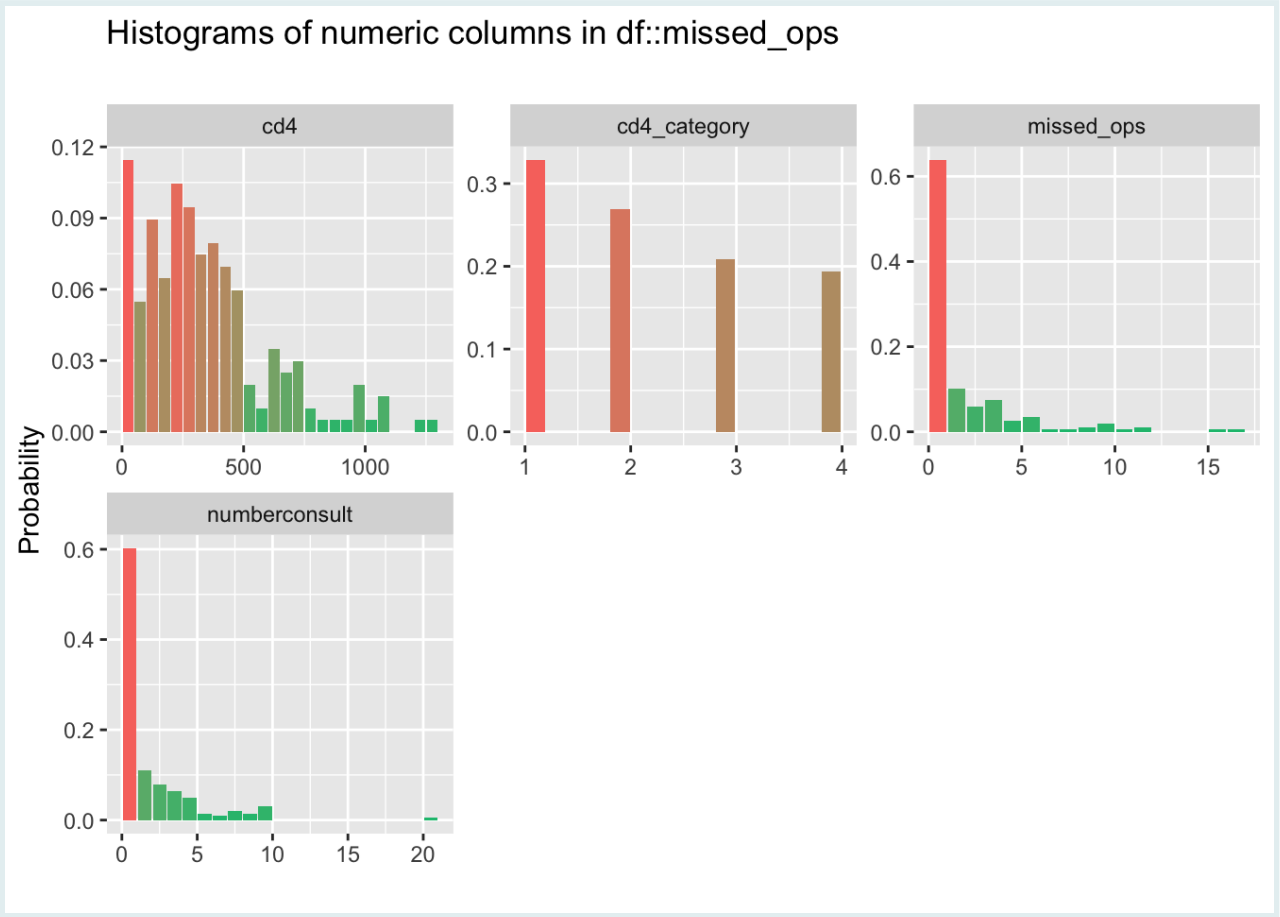
```
inspect_cat(missed_ops) %>%
  show_plot()
```



- The variable `acute` has 2 levels: `Not acute` and `yes`. This should be standardized.
- The varibale `sex` has 3 levels: `Female`, `Male`, and `M`. The `M` should be changed to `Male`.

Q: Variable types with `inspect_num()`

```
inspect_num(missed_ops) %>%
  show_plot()
```



- The variable `cd4category` should be a factor variabale

Q: Spotting data issues with `tbl_summary()`

```
tbl_summary(missed_ops)
```

Characteristic	N = 201[1]
sex	
Female	75 (37%)
M	63 (31%)
Male	63 (31%)
clage	
>50	33 (16%)
18-29	56 (28%)
30-49	112 (56%)
origine	
Others	29 (14%)
Sub Saharan Africa	66 (33%)
Western World	106 (53%)
adquired	
Heterosexual	114 (57%)
IV drug use	9 (4.5%)
MSM	68 (34%)
Other	10 (5.0%)
chronic	40 (20%)
cd4	293 (147, 452)
cd4_category	
1	66 (33%)
2	54 (27%)
3	42 (21%)
4	39 (19%)

Characteristic	N = 201[1]
Not acute	171 (85%)
yes	30 (15%)
latepresent	106 (53%)
reasonsdx	
AIDS defining illness	21 (10%)
Disease for which the prevalence of HIV is >1%	59 (29%)
Epi risk	21 (10%)
Epidemiological risk	21 (10%)
Introduction of immunosuppressive treatment	1 (0.5%)
Patient initiated	28 (14%)
Pregnancy	14 (7.0%)
Suspicion of acute HIV infection	36 (18%)
testlocal	
Anonymous consultation	26 (20%)
CHUV/PMU outpatient	41 (31%)
Primary care physician	64 (49%)
Unknown	70
numberconsult	1.00 (0.00, 3.00)
consultcat	
>5 previous consultations	29 (14%)
1-5 previous consultations	82 (41%)
No previous consultation	90 (45%)
cattimenegtestdx	
HIV testing the previous year	28 (14%)
Last HIV test >1 before diagnosis	54 (27%)
No previous HIV testing	119 (59%)
missed_ops	0.00 (0.00, 3.00)
Unknown	2
NaN.	0 (NA%)
Unknown	201

[1] n (%); Median (IQR)

- For the variable `reasonsdx`, there are the categories `Epidemiological risk` and `Epi risk` which should be a single category

Q: Data report with `create_report()`

```
DataExplorer::create_report(missed_ops)
```

References

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

- Batra, Neale, et al. The Epidemiologist R Handbook. 2021. *Cleaning data and core functions*. https://epirhandbook.com/en/cleaning-data-and-core-functions.html#cleaning-data-and-core-functions
- Waring E, Quinn M, McNamara A, Arino de la Rubia E, Zhu H, Ellis S (2022). skimr: Compact and Flexible Summaries of Data. https://docs.ropensci.org/skimr/ (website), https://github.com/ropensci/skimr/.

Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAURE VANCAUWENBERGHE

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

Data Cleaning Pipeline 2: Fixing Inconsistencies

Introduction

In the previous lesson, we learned a range of functions for diagnosing data issues. Now, let's focus on some common techniques and functions for fixing those issues. Let's get started!

Learning Objectives

By the end of this lesson, you will be able to:

- Understand how to clean column names, both automatically and manually.
- Remove empty columns and rows with ease.
- Effectively eliminate duplicate entries.
- Correct and fix string values in your data.
- Convert and modify data types as required.

Packages

Load the following packages for this lesson:

Dataset

The dataset we will be using for this lesson is a slightly modified version of the dataset we used in the first `Data Cleaning` lesson; here, we've added slightly more errors to clean! Check out lesson 1 for an explanation of this dataset.

```
non_adherence <- read_csv(here("data/non_adherence_messy.csv"))
```

```
non_adherence
```

```
## # A tibble: 5 × 15
##   patient_id District `Health unit` Sex   Age_35
##        <dbl>    <dbl>         <dbl> <chr> <chr>
## 1      10037        1             1 Male  over 35
## 2      10537        1             1 F     over 35
## 3       5489        2             3 F     Under 35
## 4       5523        2             3 Male  Under 35
## 5       4942        2             3 F     over 35
## # ℹ 10 more variables: `Age at ART initiation` <dbl>,
## #   EDUCATION_OF_PATIENT <chr>, …
```

Cleaning column names

As a general rule, column names should have a "clean", standardized syntax so that we can easily work with them and so that our code is readable to other coders.

Ideally, column names:

- should be short
- should have no spaces or periods(space and periods should be replaced by underscore "_")
- should have no unusual characters(&, #, <, >)
- should have a similar style

To check out our column names, we can use the `names()` function from base R.

```
names(non_adherence)
```

```
##  [1] "patient_id"                "District"
##  [3] "Health unit"               "Sex"
##  [5] "Age_35"                    "Age at ART initiation"
##  [7] "EDUCATION_OF_PATIENT"      "OCCUPATION_OF_PATIENT"
##  [9] "Civil...status"            "WHO status at ART initiaiton"
## [11] "BMI_Initiation_Art"        "CD4_Initiation_Art"
```

```
## [13] "regimen.1"                    "Nr_of_pills_day"
## [15] "NA"
```

Here we can see that:

- some names contain spaces
- some names contain special characters such as `...`
- some names are in upper case while some are not

Automatic column name cleaning with `janitor::clean_names()`

A handy function for standardizing column names is the `clean_names()` from the `janitor` {janitor} package. The function `clean_names()` converts all names to consist of only underscores, numbers, and letters, using the snake case style.

```
non_adherence %>%
  clean_names() %>%
  names()
```

```
##  [1] "patient_id"                "district"
##  [3] "health_unit"               "sex"
##  [5] "age_35"                    "age_at_art_initiation"
##  [7] "education_of_patient"      "occupation_of_patient"
##  [9] "civil_status"              "who_status_at_art_initiaiton"
## [11] "bmi_initiation_art"        "cd4_initiation_art"
## [13] "regimen_1"                 "nr_of_pills_day"
## [15] "na"
```

From this output, we can see that:

- upper case variable names were converted to lower case (e.g., `EDUCATION_OF_PATIENT` is now `education_of_patient`)
- spaces inside the variable names have been converted to underscores (e.g., `Age at ART initiation` is now `age_at_art_initiation`)
- periods(`.`) have all been replaced by underscores (e.g., `Civil...status` is now `civil_status`)

Let's save this cleaned dataset as `non_adherence_clean`.

```
non_adherence_clean <-
  non_adherence %>%
  clean_names()
non_adherence_clean
```

```
## # A tibble: 1,420 × 15
##    patient_id district health_unit sex   age_35
```

```
##         <dbl>    <dbl>       <dbl> <chr> <chr>
##  1      10037        1           1 Male  over 35
##  2      10537        1           1 F     over 35
##  3       5489        2           3 F     Under 35
##  4       5523        2           3 Male  Under 35
##  5       4942        2           3 F     over 35
##  6       4742        2           3 Male  over 35
##  7      10879        1           1 Male  over 35
##  8       2885        2           3 Male  over 35
##  9       4861        2           3 F     over 35
## 10       5180        2           3 Male  over 35
## # ℹ 1,410 more rows
## # ℹ 10 more variables: age_at_art_initiation <dbl>, …
```

Q: Automatic cleaning

The following dataset has been adapted from a study that used retrospective data to characterize the tmporal and spatial dynamics of typhoid fever epidemics in Kasene, Uganda.

```
typhoid <- read_csv(here("data/typhoid_uganda.csv"))
```

```
## Rows: 215 Columns: 31
## ── Column specification
──────────────────────────────────────────────────────
## Delimiter: ","
## chr (18): Householdmembers, Positioninthehousehold,
Watersourcedwithinhouseh...
## dbl (11): UniqueKey, CaseorControl, Age, Sex, Levelofeducation,
Below10years...
## lgl  (2): NA, NAN
##
## 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.
```

Use the `clean_names()` function from `janitor` to clean the variables names in the `typhoid` dataset.

{stringr} and `dplyr::rename_with()` for renaming columns

To make more manual changes to column names, you should already know how to use the `rename()` function from the `dplyr` package. However, this function requires you to specify the old and new column names. This can be tedious if you have many columns to rename.

An alternative is to use the `rename_with()` function from the `dplyr` package. This function allows you to apply a function to all column names. For example, if we want to

convert all column names to upper case, we can use the `toupper()` function inside `rename_with()`.

```
non_adherence_clean %>%
  rename_with(.cols = everything(), .fn = toupper)
```

```
## # A tibble: 1,420 × 15
##    PATIENT_ID DISTRICT HEALTH_UNIT SEX   AGE_35
##         <dbl>    <dbl>       <dbl> <chr> <chr>
##  1      10037        1           1 Male  over 35
##  2      10537        1           1 F     over 35
##  3       5489        2           3 F     Under 35
##  4       5523        2           3 Male  Under 35
##  5       4942        2           3 F     over 35
##  6       4742        2           3 Male  over 35
##  7      10879        1           1 Male  over 35
##  8       2885        2           3 Male  over 35
##  9       4861        2           3 F     over 35
## 10       5180        2           3 Male  over 35
## # ℹ 1,410 more rows
## # ℹ 10 more variables: AGE_AT_ART_INITIATION <dbl>, …
```

```
# We can also omit the .cols argument, which defaults to everything()
non_adherence_clean %>%
  rename_with(.fn = toupper)
```

```
## # A tibble: 1,420 × 15
##    PATIENT_ID DISTRICT HEALTH_UNIT SEX   AGE_35
##         <dbl>    <dbl>       <dbl> <chr> <chr>
##  1      10037        1           1 Male  over 35
##  2      10537        1           1 F     over 35
##  3       5489        2           3 F     Under 35
##  4       5523        2           3 Male  Under 35
##  5       4942        2           3 F     over 35
##  6       4742        2           3 Male  over 35
##  7      10879        1           1 Male  over 35
##  8       2885        2           3 Male  over 35
##  9       4861        2           3 F     over 35
## 10       5180        2           3 Male  over 35
## # ℹ 1,410 more rows
## # ℹ 10 more variables: AGE_AT_ART_INITIATION <dbl>, …
```

But of course, this is not a recommended name style. Let's try something more useful. In the `non_adherence` dataset, some of the column names end with `_of_patient`. This is not necessary, as we know that all the variables are of the patient. We can use the `str_replace_all()` function from the `stringr` package to remove the `_of_patient` from all column names.

Here is the syntax for `str_replace_all()`:

```r
test_string <- "this is a test string"
str_replace_all(string = test_string, pattern = "test", replacement = "new")
```

```
## [1] "this is a new string"
```

It has three arguments, `string`, the string to be modified, `pattern`, the pattern to be replaced, and `replacement`, the new pattern to replace the old pattern.

Let's apply this to our column names.

```r
non_adherence_clean_2 <-
  non_adherence_clean %>%
  # manually re-name columns
  rename_with(str_replace_all, pattern = "_of_patient", replacement = "")

non_adherence_clean_2
```

```
## # A tibble: 5 × 15
##   patient_id district health_unit sex   age_35
##        <dbl>    <dbl>       <dbl> <chr> <chr>
## 1      10037        1           1 Male  over 35
## 2      10537        1           1 F     over 35
## 3       5489        2           3 F     Under 35
## 4       5523        2           3 Male  Under 35
## 5       4942        2           3 F     over 35
## # ℹ 10 more variables: age_at_art_initiation <dbl>,
## #   education <chr>, occupation <chr>, …
```

That looks great! We'll use this dataset with the clean columns in the next section.

In the cleaning examples in this lesson, we are assigning a lof of intermediate objects, such as `non_adherence_clean` and `non_adherence_clean_2`. This is not usually necessary; we are doing it here for the sake of clarity in the tutorial. In your own code, you will typically combine multiple cleaning steps into a single pipe chain:

```r
non_adherence_clean <-
  non_adherence %>%
  # cleaning step 1 %>%
  # cleaning step 2 %>%
  # cleaning step 3 %>%
  # and so on
```

Q: Complete cleaning of column names

Standardize the column names in the `typhoid` dataset with `clean_names()` then;

- replace `or_` with _
- replace `of` with _

Removing Empty Rows and Columns in R

Empty rows and columns in a dataset, filled entirely with NA values, can be misleading and affect your analysis. It's crucial to identify and remove these to ensure that every row represents a valid data point and every column represents a meaningful variable. Here's a straightforward guide to doing this in R.

How to Remove Empty Columns

First, let's focus on columns. Use the `inspect_na()` function from the `inspectdf` package to identify columns filled entirely with NA values.

```
inspectdf::inspect_na(non_adherence_clean_2)
```

If a column shows 100% NA values, it's empty and should be removed. To do this, use the `remove_empty()` function from the `janitor` package, specifying `"cols"` to target columns:

```
# Before removal
ncol(non_adherence_clean_2)
```

```
## [1] 15
```

```
# Removing empty columns
non_adherence_clean_3 <- non_adherence_clean_2 %>%
  remove_empty("cols")

# After removal
ncol(non_adherence_clean_3)
```

```
## [1] 14
```

How to Remove Empty Rows

Identifying empty rows is a little more complicated, as there is no function (that we know of) to do this directly. Here is a construction that works:

```
non_adherence_clean_3 %>%
  mutate(missing_count = rowSums(is.na(.))) %>%
  select(missing_count)
```

```
## # A tibble: 1,420 × 1
##    missing_count
##            <dbl>
##  1             2
##  2             0
##  3             3
##  4             3
##  5             2
##  6             2
##  7             1
##  8             2
##  9             3
## 10             2
## # ℹ 1,410 more rows
```

We can then filter the dataset to show only rows with a `missing_count` of 14, indicating that all values in the row are NA:

```
non_adherence_clean_3 %>%
  mutate(missing_count = rowSums(is.na(.))) %>%
  filter(missing_count == 14)
```

```
## # A tibble: 3 × 15
##   patient_id district health_unit sex   age_35
##        <dbl>    <dbl>       <dbl> <chr> <chr>
## 1         NA       NA          NA <NA>  <NA>
## 2         NA       NA          NA <NA>  <NA>
## 3         NA       NA          NA <NA>  <NA>
## # ℹ 10 more variables: age_at_art_initiation <dbl>,
## #   education <chr>, occupation <chr>, …
```

These rows can then be removed using the same `remove_empty()` function from the `janitor` package. This time, specify `"rows"` to target rows:

```
# Before removal
nrow(non_adherence_clean_3)
```

```
## [1] 1420
```

```r
# Removing empty rows
non_adherence_clean_4 <- non_adherence_clean_3 %>%
  remove_empty("rows")

# After removal
nrow(non_adherence_clean_4)
```

```
## [1] 1417
```

Notice the change in the row count, indicating the removal of empty rows.

Q: Removing empty columns and rows

Remove both empty rows and empty columns from the `typhoid` dataset. How many rows and columns are left?

Removing Duplicate Rows

Duplicated rows in datasets can occur due to multiple data sources or survey responses. It's essential to identify and remove these duplicates for accurate analysis.

Use `janitor::get_dupes()` to quickly identify duplicate rows. This function helps you visually inspect duplicates before removal. For example:

```r
non_adherence_clean_4 %>%
  get_dupes()
```

```
## No variable names specified - using all columns.
```

This output shows duplicated rows, identifiable by common variables like `patient_id`.

After identifying duplicates, use `dplyr::distinct()` to remove them, keeping only the unique rows. For instance:

```r
# Before removal
nrow(non_adherence_clean_4)
```

```
## [1] 1417
```

```r
# Removing duplicates
non_adherence_distinct <- non_adherence_clean_4 %>%
  distinct()

# After removal
nrow(non_adherence_distinct)
```

```
## [1] 1413
```

We can see that the number of rows has decreased, indicating the removal of duplicates. We can check this by using `get_dupes()` again:

```r
non_adherence_distinct %>%
  get_dupes()
```

```
## No variable names specified - using all columns.
```

```
## No duplicate combinations found of: patient_id, district, health_unit,
sex, age_35, age_at_art_initiation, education, occupation, civil_status, ...
and 5 other variables
```

```
## # A tibble: 0 × 15
## # ℹ 15 variables: patient_id <dbl>, district <dbl>,
## #   health_unit <dbl>, sex <chr>, age_35 <chr>, …
```

Q: Removing duplicates

Identify the duplicates in the `typhoid` dataset using `get_dupes()`, then remove them using `distinct()`.

Homogenize strings

You may remember that with the `skim` output from the first data cleaning pipeline lesson, we had instances where our string characters were inconstent regarding capitalization. For example, for our `occupation` variable, we had both `Professor` and `professor`:

```r
non_adherence_distinct %>%
  count(occupation, name = "Count") %>%
  arrange(-str_detect(occupation, "rofessor")) # to show the professor rows
first
```

```
## # A tibble: 51 × 2
##    occupation             Count
##    <chr>                  <int>
##  1 Professor                 35
##  2 professor                 11
##  3 Accountant                 1
##  4 Administrator              1
##  5 Agriculture technician     3
##  6 Artist                     1
##  7 Basic service agent        2
##  8 Boat captain               1
##  9 Business                   3
## 10 Commercial                18
## # ℹ 41 more rows
```

In order to address this, we can transform our character columns to a specific case. Here we will use title case, since that looks better on graphics and reports.

```
non_adherence_case_corrected <-
  non_adherence_distinct %>%
  mutate(across(.cols = c(sex, age_35, education, occupation, civil_status),
                .fns = ~ str_to_title(.x))) #
```

If we count the number of unique levels for the `occupation` variable, we can see that we have reduced the number of unique levels from 49 to 47:

```
non_adherence_case_corrected %>%
  count(occupation, name = "Count")
```

```
## # A tibble: 49 × 2
##    occupation             Count
##    <chr>                  <int>
##  1 Accountant                 1
##  2 Administrator              1
##  3 Agriculture Technician     3
##  4 Artist                     1
##  5 Bartender                  1
##  6 Basic Service Agent        2
##  7 Boat Captain               1
##  8 Business                   3
##  9 Commercial                18
## 10 Cook                       3
## # ℹ 39 more rows
```

As we can see, we have gone from 51 to 49 unique levels for the `occupation` variable.

Q: Transforming to lowercase

Transform all the strings in the `typhoid` dataset to lowercase.

dplyr::case_match() for String Cleaning

For more bespoke string cleaning, we can use the `case_match()` function from the {dplyr} package. This function allows us to specify a series of conditions and values to be applied to a vector.

Here is a simple example demonstrating how to use `case_match()`:

```
test_vector <- c("+", "-", "NA", "missing")
case_match(test_vector,
           "+" ~ "Positive",
           "-" ~ "Negative",
           .default = "Other")
```

```
## [1] "Positive" "Negative" "Other"    "Other"
```

As you can see, the `case_match()` function takes a vector as its first argument, followed by a series of conditions and values. The `.default` argument is optional and specifies the value to be returned if none of the conditions are met.

We'll apply this first on the `sex` column in the `non_adherence_distinct` dataset. First, observe the levels in this variable:

```
non_adherence_case_corrected %>% count(sex, name = "Count")
```

```
## # A tibble: 2 × 2
##   sex   Count
##   <chr> <int>
## 1 F      1084
## 2 Male    329
```

In this variable, we can see that there are inconsistencies in the way the levels have been coded. Let's use the `case_match()` function so that `F` is changed to `Female`.

```
non_adherence_case_corrected %>%
  mutate(sex = case_match(sex, "F" ~ "Female", .default = sex)) %>%
  count(sex, name = "Count")
```

```
## # A tibble: 2 × 2
##   sex    Count
##   <chr>  <int>
## 1 Female  1084
## 2 Male     329
```

The utility is more obvious when you have a lot of values to change. For example, let's make the following modifications on the occupation column:

- Replace "Worker" with "Laborer"
- Replace "Housewife" with "Homemaker"
- Replace "Truck Driver" and "Taxi Driver" with "Driver"

```
non_adherence_recoded <-
  non_adherence_case_corrected %>%
  mutate(sex = case_match(sex, "F" ~ "Female", .default = sex)) %>%
  mutate(occupation = case_match(occupation,
                                 "Worker" ~ "Laborer",
                                 "Housewife" ~ "Homemaker",
                                 "Truck Driver" ~ "Driver",
                                 "Taxi Driver" ~ "Driver",
                                 .default = occupation))
```

WATCH OUT

If you don't specify the `.default=column_name` argument then all of the values in that column that don't match the ones those you are changing and explicitly mentioned in your `case_match()` function will be returned as NA.

Q: Fixing strings

The variable `householdmembers` from the `typhoid` dataset should represent the number of individuals in a household. There is a value `01–May` in this variable. Recode this value to 1–5.

Converting data types

Columns containing values that are numbers, factors, or logical values (TRUE/FALSE) will only behave as expected if they are correctly classified. As such, you may need to redefine the type or class of your variable.

REMINDER

R has 6 basic data types/classes.

- `character`: strings or individual characters, quoted
- `numeric` : any real numbers (includes decimals)

REMINDER

- `integer`: any integer(s)/whole numbers
- `logical`: variables composed of TRUE or FALSE
- `factor`: categorical/qualitative variables
- `Date/POSIXct`: represents calendar dates and times

You may recall in the last lesson that our dataset contains 5 character variables and 9 numeric variables. Let's quickly take a look at our variables using the `skim()` function from last lesson:

```
skim(non_adherence_recoded) %>%
  select(skim_type) %>%
  count(skim_type)
```

Looking at our data, all our variables are categorical, except `age_at_art_initation`, `bmi_initiation_art`, `cd4_initiation_art`, and `nr_of_pills_day`. Let's change all the others to factor variables using the `as.factor()` function!

```
non_adherence_distinct %>%
  mutate(across(
    .cols = !c(
      age_at_art_initiation,
      bmi_initiation_art,
      cd4_initiation_art,
      nr_of_pills_day
    ),
    .fns = as.factor
  )) %>%
  skim() %>%
  select(skim_type) %>%
  count(skim_type)
```

Great, that's exactly what we wanted!

Q: Changing data types

Convert the variables in position 13 to 29 in the `typhoid` dataset to factor.

Wrap Up!

Congratulations on completing the two-part lesson on the data cleaning piepline! You are now better equipped to tackle the intricacies of real-world datasets. Remember, data cleaning is not just about tidying up messy data; it's about ensuring the reliability and accuracy of your analyses. By mastering techniques like handling column names, eliminating empty entries, addressing duplicates, refining string values, and managing

data types, you've honed your abilities to transform raw health data into a clean foundation for meaningful insights!

Answer Key

Q: Automatic cleaning

```
clean_names(typhoid)
```

Q: Complete cleaning of column names

```
typhoid %>%
  clean_names() %>%
  rename_with(str_replace_all, pattern = "or_",replacement = "_") %>%
  rename_with(str_replace_all, pattern = "of",replacement = "_") %>%
  names()
```

```
 ##  [1] "unique_key"                  "case_control"
 ##  [3] "age"                         "sex"
 ##  [5] "level_education"             "householdmembers"
 ##  [7] "below10years"                "n1119years"
 ##  [9] "n2035years"                  "n3644years"
 ## [11] "n4565years"                  "above65years"
 ## [13] "positioninthehousehold"      "watersourcedwithinhousehold"
 ## [15] "borehole"                    "river"
 ## [17] "tap"                         "rainwatertank"
 ## [19] "unprotectedspring"           "protectedspring"
 ## [21] "pond"                        "shallowwell"
 ## [23] "stream"                      "jerrycan"
 ## [25] "bucket"                      "county"
 ## [27] "subcounty"                   "parish"
 ## [29] "village"                     "na"
 ## [31] "nan"
```

Q: Removing empty columns and rows

```
typhoid_empty_removed <-
  typhoid %>%
  remove_empty("cols") %>%
  remove_empty("rows")

# identify how many empty columns were removed
ncol(typhoid) - ncol(typhoid_empty_removed)
# identify how many empty rows were removed
nrow(typhoid) - nrow(typhoid_empty_removed)
```

Q: Removing duplicates

```
# Identify duplicates
get_dupes(typhoid)
```

```
## No variable names specified - using all columns.
```

```
# Remove duplicates
typhoid_distinct <- typhoid %>%
  distinct()

# Ensure all distinct rows left
get_dupes(typhoid_distinct)
```

```
## No variable names specified - using all columns.
```

```
## No duplicate combinations found of: UniqueKey, CaseorControl, Age, Sex,
Levelofeducation, Householdmembers, Below10years, N1119years, N2035years, ...
and 22 other variables
```

Q: Transforming to lowercase

```
typhoid %>%
  mutate(across(where(is.character),
                ~ tolower(.x)))
```

Q: Fixing strings

```
typhoid %>%
  mutate(Householdmembers = case_match(Householdmembers, "01-May" ~ "1-5",
.default=Householdmembers)) %>%
  count(Householdmembers)
```

Q: Changing data types

```
typhoid %>%
  mutate(across(13:29, ~as.factor(.)))
```

Contributors

The following team members contributed to this lesson:

AMANDA MCKINLEY

R Developer and Instructor, the GRAPH Network

KENE DAVID NWOSU

Data analyst, the GRAPH Network
Passionate about world improvement

LAURE VANCAUWENBERGHE

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