

Data Cleaning

Recap on summarization

- `summary(x)`: quantile information
- `count(x)`: what unique values do you have?
 - `distinct()`: what are the distinct values?
 - `n_distinct()` with `pull()`: how many distinct values?
- `group_by()`: changes all subsequent functions
 - combine with `summarize()` to get statistics per group
 - combine with `mutate()` to add column
- `summarize()` with `n()` gives the count (NAs included)

▮ [Day 4 Cheatsheet](#)

Recap on data classes

- There are two types of number class objects: integer and double
- Logic class objects only have **TRUE** or **FALSE** (without quotes)
- `class()` can be used to test the class of an object `x`
- `as.CLASS_NAME(x)` can be used to change the class of an object `x`
- Factors are a special character class that has levels - more on that soon!
- tibbles show column classes!
- two dimensional object classes include: data frames, tibbles, matrices, and lists
- Dates can be handled with the `lubridate` package
- Make sure you choose the right function for the way the date is formatted!

▮ [Day 4 Cheatsheet](#)

Data Cleaning

In general, data cleaning is a process of investigating your data for inaccuracies, or recoding it in a way that makes it more manageable.

□ MOST IMPORTANT RULE - LOOK □ AT YOUR DATA! □

Dealing with Missing Data

Missing data types

One of the most important aspects of data cleaning is missing values.

Types of “missing” data:

- **NA** - general missing data
- **NaN** - stands for “**N**ot **a** **N**umber”, happens when you do $0/0$.
- **Inf** and **-Inf** - Infinity, happens when you divide a positive number (or negative number) by 0.

Finding Missing data

- `is.na` - looks for NAN and NA
- `is.nan` - looks for NAN
- `is.infinite` - looks for Inf or -Inf

```
test <- c(0, NA, -1)  
test/0
```

```
[1] NaN  NA -Inf
```

```
test <- test/0  
is.na(test)
```

```
[1] TRUE TRUE FALSE
```

```
is.nan(test)
```

```
[1] TRUE FALSE FALSE
```

```
is.infinite(test)
```

```
[1] FALSE FALSE TRUE
```

Useful checking functions

`any()` can help you check if there are any NA values in a vector

test

```
[1] NaN NA -Inf
```

```
any(is.na(test))
```

```
[1] TRUE
```


Finding NA values with count()

Check the values for your variables, are they what you expect?

count() is a great option because it helps you check if rare values make sense.

Let's look at the CO heat-related ER visits dataset again.

```
er <- read_csv(file =  
  "https://daseh.org/data/CO\_ER\_heat\_visits.csv")  
er %>% count(visits)
```

```
# A tibble: 37 × 2  
  visits      n  
  <dbl> <int>  
1      0    339  
2     11      2  
3     12      5  
4     13      8  
5     14      3  
6     15      2  
7     16      3  
8     17      3  
9     18      5  
10    19      7  
#   27 more rows
```

naniar

Sometimes you need to look at lots of data though... the naniar package is a good option.

```
#install.packages("naniar")  
library(naniar)
```



“Artwork by @allison_horst”. <https://allisonhorst.com/>

naniar: pct_complete()

This can tell you if there are missing values in the dataset.

```
pct_complete(er)
```

```
[1] 73.65451
```

Or for a particular variable:

```
er %>% select(visits) %>%  
  pct_complete()
```

```
[1] 60.54688
```

```
er %>% select(rate) %>%  
  pct_complete()
```

```
[1] 60.54688
```

naniar:miss_var_summary()

To get the percent missing (and counts) for each variable as a table, use this function:

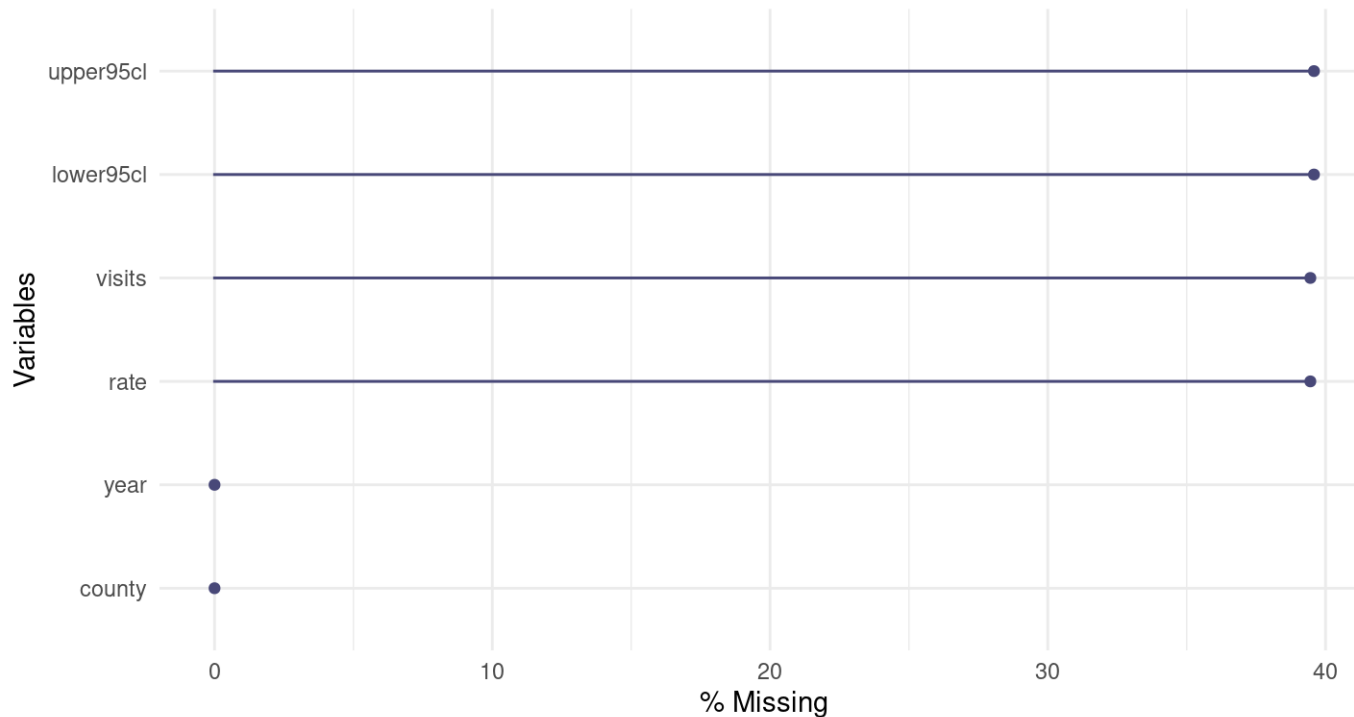
```
miss_var_summary(er)
```

```
# A tibble: 6 × 3
  variable n_miss pct_miss
  <chr>    <int>    <num>
1 lower95cl    304    39.6
2 upper95cl    304    39.6
3 rate        303    39.5
4 visits      303    39.5
5 county         0     0
6 year          0     0
```

naniar plots

The `gg_miss_var()` function creates a nice plot about the number of missing values for each variable, (need a data frame).

```
gg_miss_var(er, show_pct = TRUE)
```



Missing Data Issues

Recall that mathematical operations with NA often result in NAs.

```
sum(c(1, 2, 3, NA))
```

```
[1] NA
```

```
mean(c(1, 2, 3, NA))
```

```
[1] NA
```

```
median(c(1, 2, 3, NA))
```

```
[1] NA
```

Missing Data Issues

This is also true for logical data. Recall that **TRUE** is evaluated as 1 and **FALSE** is evaluated as 0.

```
x <- c(TRUE, TRUE, TRUE, TRUE, FALSE, NA)  
sum(x)
```

```
[1] NA
```

```
sum(x, na.rm = TRUE)
```

```
[1] 4
```

filter() and missing data

Be **careful** with missing data using subsetting!

filter() removes missing values by default. Because R can't tell for sure if an NA value meets the condition. To keep them need to add `is.na()` conditional.

Think about if this is OK or not - it depends on your data!

filter() and missing data

What if NA values represent values that are so low it is undetectable?

Filter will drop them from the data.

```
er %>% filter(visits > 0)
```

```
# A tibble: 126 × 6
```

	county	rate	lower95cl	upper95cl	visits	year
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Adams	6.73	NA	9.24	29	2011
2	Adams	4.84	2.85	NA	23	2012
3	Adams	6.84	4.36	9.31	31	2013
4	Adams	3.08	1.71	4.85	15	2014
5	Adams	3.36	1.89	5.23	16	2015
6	Adams	8.85	6.12	11.6	42	2016
7	Adams	6.63	4.29	8.98	32	2017
8	Adams	7.11	4.77	9.44	37	2018
9	Adams	6.76	4.53	8.99	36	2019
10	Adams	4.76	2.82	6.70	24	2020

```
# 116 more rows
```

filter() and missing data

`is.na()` can help us keep them.

```
er %>% filter(visits > 0 | is.na(visits))
```

```
# A tibble: 429 × 6
```

	county	rate	lower95cl	upper95cl	visits	year
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Adams	6.73	NA	9.24	29	2011
2	Adams	4.84	2.85	NA	23	2012
3	Adams	6.84	4.36	9.31	31	2013
4	Adams	3.08	1.71	4.85	15	2014
5	Adams	3.36	1.89	5.23	16	2015
6	Adams	8.85	6.12	11.6	42	2016
7	Adams	6.63	4.29	8.98	32	2017
8	Adams	7.11	4.77	9.44	37	2018
9	Adams	6.76	4.53	8.99	36	2019
10	Adams	4.76	2.82	6.70	24	2020

```
#   419 more rows
```

To remove rows with NA values for a variable use `drop_na()`

A function from the `tidyr` package. (Need a data frame to start!)

Disclaimer: Don't do this unless you have thought about if dropping NA values makes sense based on knowing what these values mean in your data. **Also consider if you need those rows for values for other variables.**

```
dim(er)
```

```
[1] 768    6
```

```
er_drop <- er %>% drop_na(lower95cl)  
dim(er_drop)
```

```
[1] 464    6
```

Let's take a look

Can still have NAs for other columns

er_drop

```
# A tibble: 464 × 6
  county rate lower95cl upper95cl visits year
  <chr>   <dbl>   <dbl>   <dbl>   <dbl> <dbl>
1 Adams  4.84     2.85    NA      23    2012
2 Adams  6.84     4.36    9.31    31    2013
3 Adams  3.08     1.71    4.85    15    2014
4 Adams  3.36     1.89    5.23    16    2015
5 Adams  8.85     6.12   11.6    42    2016
6 Adams  6.63     4.29    8.98    32    2017
7 Adams  7.11     4.77    9.44    37    2018
8 Adams  6.76     4.53    8.99    36    2019
9 Adams  4.76     2.82    6.70    24    2020
10 Adams 6.93     4.61    9.25    35    2021
#   454 more rows
```

To remove rows with **NA** values for a data frame use **drop_na()**

This function of the `tidyr` package drops rows with **any** missing data in **any** column when used on a df.

```
er_drop <- er %>% drop_na()  
er_drop
```

```
# A tibble: 463 × 6
```

	county	rate	lower95cl	upper95cl	visits	year
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Adams	6.84	4.36	9.31	31	2013
2	Adams	3.08	1.71	4.85	15	2014
3	Adams	3.36	1.89	5.23	16	2015
4	Adams	8.85	6.12	11.6	42	2016
5	Adams	6.63	4.29	8.98	32	2017
6	Adams	7.11	4.77	9.44	37	2018
7	Adams	6.76	4.53	8.99	36	2019
8	Adams	4.76	2.82	6.70	24	2020
9	Adams	6.93	4.61	9.25	35	2021
10	Adams	8.23	5.81	10.6	45	2022

```
#   453 more rows
```

Drop columns with any missing values

Use the `miss_var_which()` function from `naniar`

```
miss_var_which(er)# which columns have missing values
```

```
[1] "rate"      "lower95cl" "upper95cl" "visits"
```

Drop columns with any missing values

`miss_var_which` and function from `naniar` (need a data frame)

```
er_drop <- er %>% select(!miss_var_which(er))  
er_drop
```

```
# A tibble: 768 × 2
```

```
  county  year
```

```
  <chr>  <dbl>
```

```
1 Adams  2011
```

```
2 Adams  2012
```

```
3 Adams  2013
```

```
4 Adams  2014
```

```
5 Adams  2015
```

```
6 Adams  2016
```

```
7 Adams  2017
```

```
8 Adams  2018
```

```
9 Adams  2019
```

```
10 Adams  2020
```

```
#   758 more rows
```

Change a value to be **NA**

Let's say we think that all 0 values should be **NA**.

Maybe we think the person who entered the CO heat-related ER visits data made a mistake in how they coded missing data.

```
er %>% count(visits)
```

```
# A tibble: 37 × 2
```

```
  visits      n  
  <dbl> <int>
```

```
1      0    339  
2     11      2  
3     12      5  
4     13      8  
5     14      3  
6     15      2  
7     16      3  
8     17      3  
9     18      5  
10    19      7
```

```
#   27 more rows
```


Change a value to be NA

The `na_if()` function of `dplyr` can be helpful for changing all 0 values to NA.

```
er_nozero <- er %>%  
  mutate(visits = na_if(visits, 0))
```

```
er_nozero %>% count(visits)
```

```
# A tibble: 36 × 2
```

	visits <dbl>	n <int>
1	11	2
2	12	5
3	13	8
4	14	3
5	15	2
6	16	3
7	17	3
8	18	5
9	19	7
10	20	2

```
# 26 more rows
```

Change NA to be a value

The `replace_na()` function (part of the `tidyr` package), can do the opposite of `na_if()`. (note that you must use numeric values as replacement - we will show how to replace with character strings soon)

```
er %>%  
  mutate(visits = replace_na(visits, 0)) %>%  
  count(visits)
```

```
# A tibble: 36 × 2
```

	visits	n
	<dbl>	<int>
1	0	642
2	11	2
3	12	5
4	13	8
5	14	3
6	15	2
7	16	3
8	17	3
9	18	5
10	19	7

```
#   26 more rows
```

Think about NA

THINK ABOUT YOUR DATA FIRST!

- ▯ Sometimes removing NA values leads to distorted math - be careful!
- ▯ Think about what your NA means for your data (are you sure ?).
 - Is an NA for values so low they could not be reported?
 - Or is it if it was too low and also if there was a different issue (like no one reported)?

Think about **NA**

If it is something more like a zero then you might want it included in your data like a zero instead of an **NA**.

Example: - survey reports **NA** if student has never tried cigarettes - survey reports 0 if student has tried cigarettes but did not smoke that week

□ You might want to keep the **NA** values so that you know the original sample size.

Word of caution

▮ Calculating percentages will give you a different result depending on your choice to include NA values.!

This is because the denominator changes.

Word of caution - Percentages with NA

```
count(er, visits) %>% mutate(percent = (n/(sum(n)) *100))
```

```
# A tibble: 37 × 3
```

	visits <dbl>	n <int>	percent <dbl>
1	0	339	44.1
2	11	2	0.260
3	12	5	0.651
4	13	8	1.04
5	14	3	0.391
6	15	2	0.260
7	16	3	0.391
8	17	3	0.391
9	18	5	0.651
10	19	7	0.911

```
#   27 more rows
```

Word of caution - Percentages with NA

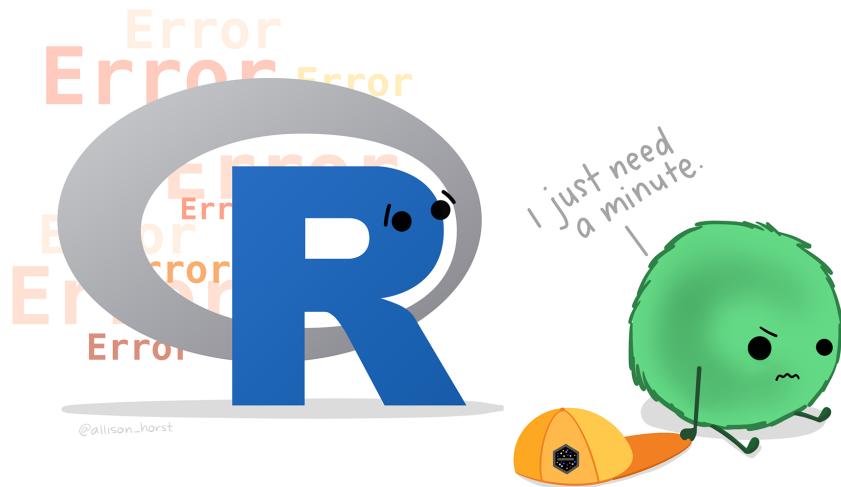
```
er %>% drop_na(visits) %>%  
  count(visits) %>% mutate(percent = (n/(sum(n)) *100))
```

```
# A tibble: 36 × 3  
  visits      n percent  
  <dbl> <int>   <dbl>  
1      0    339   72.9  
2     11      2   0.430  
3     12      5   1.08  
4     13      8   1.72  
5     14      3   0.645  
6     15      2   0.430  
7     16      3   0.645  
8     17      3   0.645  
9     18      5   1.08  
10    19      7   1.51  
#   26 more rows
```

Should you be dividing by the total count with NA values included?
It depends on your data and what NA might mean.
Pay attention to your data and your NA values!

Don't forget about the common issues

- Extra or Missing commas
- Extra or Missing parentheses
- Case sensitivity
- Spelling



GUT CHECK: What function can be used to remove NA values from a full dataframe or for an individual column?

A. `drop_nulls()`

B. `drop_na()`

C. `rem_na()`

GUT CHECK: How can you keep NA values when using **filter**?

A. include `| is.na()`

B. include `& is.na()`

Summary

- `is.na()`, `any(is.na())`, `all(is.na())`, `count()`, and functions from `naniar` like `gg_miss_var()` and `miss_var_summary` can help determine if we have NA values
- `miss_var_which()` can help you drop columns that have any missing values.
- `filter()` automatically removes NA values - can't confirm or deny if condition is met (need `| is.na()` to keep them)
- `drop_na()` can help you remove NA values from a variable or an entire data frame
- NA values can change your calculation results
- think about what NA values represent - don't drop them if you shouldn't
- `na_if()` will make NA values for a particular value
- `replace_na()` will replace `NA values with a particular value

Lab Part 1

- ▯ [Class Website](#)
- ▯ [Lab.](#) ▯ [Day 5 Cheatsheet](#)

Recoding Variables

Example of Recoding

Let's upload some practice data about microplastics. Maybe we measured the level of microplastics in someone who eats a fish versus someone who is a vegetarian.

<https://www.medicalnewstoday.com/articles/what-do-we-know-about-microplastics-in-food#Common-microplastics-in-food>

<https://www.sciencedirect.com/science/article/pii/S0045653523028072>

<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5708064/>

```
plastics <- read_csv(file =  
  "https://daseh.org/data/microplastics_in_blood.csv")
```

Rows: 12 Columns: 4

— Column specification —

Delimiter: ",", "

chr (2): Foods, microplastic

dbl (2): blood_level_start_nM, blood_level_change_nM

- Use `spec()` to retrieve the full column specification for this data.
- Specify the column types or set `show_col_types = FALSE` to quiet this message

microplastics data

plastics

A tibble: 12 × 4

	Foods	microplastic	blood_level_start_nM	blood_level_change_nM
	<chr>	<chr>	<dbl>	<dbl>
1	A	Dioxin	1	2.5
2	B	Dioxin	7	2.5
3	B	Other	2	0.5
4	A	Bisphenol A	3	-0.5
5	B	BPA	5	0.5
6	B	dioxin	8	0.5
7	A	bpa	6	2.5
8	B	0	5	3.5
9	B	dioxin	2	0.5
10	A	bpa	1	1.5
11	B	BPA	7	1.5
12	B	0	3	2.5

Oh dear...

This needs lots of recoding.

```
plastics %>%  
  count(microplastic)
```

```
# A tibble: 7 × 2  
  microplastic      n  
  <chr>          <int>  
1 BPA              2  
2 Bisphenol A      1  
3 Dioxin            2  
4 0                 2  
5 Other             1  
6 bpa               2  
7 dioxin            2
```


dp1yr can help!

Using Excel to find all of the different ways `microplastic` has been coded, could be hectic! In `dp1yr` you can use the `case_when` function.

Or you can use `case_when()`

The `case_when()` function of `dplyr` can help us to do this as well.

It is more flexible and powerful.

(need mutate here too!)

Or you can use `case_when()`

Need quotes for conditions and new values!

```
plastics %>%  
  mutate(microplastic_recoded = case_when(  
    microplastic == "0" ~ "Other",  
    microplastic == "Bisphenol A" ~ "BPA",  
    microplastic == "bpa" ~ "BPA",  
    microplastic == "dioxin" ~ "Dioxin")) %>%  
  count(microplastic, microplastic_recoded)
```

```
# A tibble: 7 × 3  
  microplastic microplastic_recoded     n  
  <chr>         <chr>         <int>  
1 BPA          <NA>           2  
2 Bisphenol A  BPA            1  
3 Dioxin       <NA>           2  
4 0            Other          2  
5 Other        <NA>           1  
6 bpa          BPA            2  
7 dioxin       Dioxin         2
```

What happened?

We seem to have NA values!

We didn't specify what happens to values that were already **Other** or **Dioxin** or **BPA**.

```
plastics %>%  
  mutate(microplastic_recoded = case_when(  
    microplastic == "0" ~ "Other",  
    microplastic == "Bisphenol A" ~ "BPA",  
    microplastic == "bpa" ~ "BPA",  
    microplastic == "dioxin" ~ "Dioxin")) %>%  
  count(microplastic, microplastic_recoded)
```

case_when() drops unspecified values

Note that automatically values not reassigned explicitly by case_when() will be NA unless otherwise specified.

General Format - this is not code!

```
{data_input} %>%  
  mutate({variable_to_fix} = case_when({Variable_fixing}  
    /some condition/ ~ {value_for_con},  
    TRUE ~ {value_for_not_meeting_condition})
```

{value_for_not_meeting_condition} could be something new or it can be the original values of the column

case_when with TRUE ~ original variable name

```
plastics %>%
  mutate(microplastic_recoded = case_when(
    microplastic == "0" ~ "Other",
    microplastic == "Bisphenol A" ~ "BPA",
    microplastic == "bpa" ~ "BPA",
    microplastic == "dioxin" ~ "Dioxin",
    TRUE ~ microplastic)) %>%

  count(microplastic_recoded)
```

A tibble: 3 × 2

	microplastic_recoded	n
	<chr>	<int>
1	BPA	5
2	Dioxin	4
3	Other	3

Typically it is good practice to include the TRUE statement

You never know if you might be missing something - and if a value already was an NA it will stay that way.

```
plastics %>%  
  mutate(microplastic_recoded = case_when(  
    microplastic == "0" ~ "Other",  
    microplastic == "Bisphenol A" ~ "BPA",  
    microplastic == "bpa" ~ "BPA",  
    microplastic == "dioxin" ~ "Dioxin",  
    TRUE ~ microplastic)) %>%  
  count(microplastic, microplastic_recoded)
```

case_when() can also overwrite/update a variable

You need to specify what we want in the first part of mutate.

```
plastics %>%  
  mutate(microplastic = case_when(  
    microplastic == "Bisphenol A" ~ "BPA",  
    microplastic == "bpa" ~ "BPA",  
    microplastic == "dioxin" ~ "Dioxin",  
    microplastic == "0" ~ "Other",  
    TRUE ~ microplastic)) %>%  
  count(microplastic)
```

A tibble: 3 × 2

	microplastic <chr>	n <int>
1	BPA	5
2	Dioxin	4
3	Other	3

More complicated case_when()

case_when can do complicated statements and can match many patterns at a time.

```
plastics %>%  
  mutate(microplastic_recoded = case_when(  
    microplastic %in% c("dioxin", "Dioxin") ~ "dioxin",  
    microplastic %in% c("BPA", "bpa", "Bisphenol A") ~ "BPA",  
    microplastic %in% c("0", "Other") ~ "Other",  
    TRUE ~ microplastic)) %>%
```

```
count(microplastic, microplastic_recoded)
```

```
# A tibble: 7 × 3  
  microplastic microplastic_recoded     n  
  <chr>         <chr>         <int>  
1 BPA          BPA             2  
2 Bisphenol A  BPA             1  
3 Dioxin       dioxin          2  
4 0            Other          2  
5 Other        Other           1  
6 bpa          BPA             2  
7 dioxin       dioxin          2
```

Another reason for `case_when()`

`case_when` can do very sophisticated comparisons!

Here we create a new variable called `Effect`.

```
plastics <- plastics %>%  
  mutate(Effect = case_when(  
    blood_level_change_nM > 0 ~ "Increase",  
    blood_level_change_nM == 0 ~ "Same",  
    blood_level_change_nM < 0 ~ "Decrease"))
```

```
head(plastics)
```

```
# A tibble: 6 × 5
```

	Foods	microplastic	blood_level_start_nM	blood_level_change_nM	Effect
	<chr>	<chr>	<dbl>	<dbl>	<chr>
1	A	Dioxin	1	2.5	Increase
2	B	Dioxin	7	2.5	Increase
3	B	Other	2	0.5	Increase
4	A	Bisphenol A	3	-0.5	Decrease
5	B	BPA	5	0.5	Increase
6	B	dioxin	8	0.5	Increase

Now it is easier to see what is happening

```
plastics %>%  
  count(Foods, Effect)
```

```
# A tibble: 3 × 3  
  Foods Effect      n  
  <chr> <chr>   <int>  
1 A     Decrease    1  
2 A     Increase    3  
3 B     Increase    8
```

GUT CHECK: we need to use what function with `case_when()` to modify or create a new variable?

A. `modify()`

B. `select()`

C. `mutate()`

GUT CHECK: If we want all unspecified values to remain the same with **case_when()**, how should we complete the **TRUE ~** statement?

- A. With the name of the variable we are modifying or using as source
- B. With the word "same"

Working with strings

Strings in R

- R can do much more than find exact matches for a whole string!



The **stringr** package

The `stringr` package:

- Modifying or finding **part** or all of a character string
- We will not cover `grep` or `gsub` - base R functions
 - are used on forums for answers
- Almost all functions start with `str_*`

stringr

`str_detect`, and `str_replace` search for matches to argument pattern within each element of a **character vector** (not data frame or tibble!).

- `str_detect` - returns TRUE if pattern is found
- `str_replace` - replaces pattern with replacement

str_detect()

The `string` argument specifies what to check

The `pattern` argument specifies what to check for (case sensitive)

```
Effect <- pull(plastics) %>% head(n = 6)
```

```
Effect
```

```
[1] "Increase" "Increase" "Increase" "Decrease" "Increase" "Increase"
```

```
str_detect(string = Effect, pattern = "d")
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
str_detect(string = Effect, pattern = "D")
```

```
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

str_replace()

The `string` argument specifies what to check

The `pattern` argument specifies what to check for

The `replacement` argument specifies what to replace the pattern with

```
str_replace(string = Effect, pattern = "D", replacement = "d")
```

```
[1] "Increase" "Increase" "Increase" "decrease" "Increase" "Increase"
```

str_replace() only replaces the first instance of the pattern in each value

str_replace_all() can be used to replace all instances within each value

```
str_replace(string = Effect, pattern = "e", replacement = "E")
```

```
[1] "IncrEase" "IncrEase" "IncrEase" "DEcrease" "IncrEase" "IncrEase"
```

```
str_replace_all(string = Effect, pattern = "e", replacement = "E")
```

```
[1] "IncrEasE" "IncrEasE" "IncrEasE" "DEcrEasE" "IncrEasE" "IncrEasE"
```

Subsetting part of a string

`str_sub()` allows you to subset part of a string

The `string` argument specifies what strings to work with

The `start` argument specifies position of where to start

The `end` argument specifies position of where to end

```
str_sub(string = Effect, start = 1, end = 3)
```

```
[1] "Inc" "Inc" "Inc" "Dec" "Inc" "Inc"
```

filter and stringr functions

```
head(plastics, n = 4)
```

```
# A tibble: 4 × 5
```

	Foods	microplastic	blood_level_start_nM	blood_level_change_nM	Effect
	<chr>	<chr>	<dbl>	<dbl>	<chr>
1	A	Dioxin	1	2.5	Increase
2	B	Dioxin	7	2.5	Increase
3	B	Other	2	0.5	Increase
4	A	Bisphenol A	3	-0.5	Decrease

```
plastics %>%  
  filter(str_detect(string = microplastic,  
                    pattern = "B"))
```

```
# A tibble: 3 × 5
```

	Foods	microplastic	blood_level_start_nM	blood_level_change_nM	Effect
	<chr>	<chr>	<dbl>	<dbl>	<chr>
1	A	Bisphenol A	3	-0.5	Decrease
2	B	BPA	5	0.5	Increase
3	B	BPA	7	1.5	Increase

OK back to our original problem

```
count(plastics, microplastic)
```

```
# A tibble: 7 × 2
  microplastic      n
  <chr>         <int>
1 BPA           2
2 Bisphenol A   1
3 Dioxin        2
4 0             2
5 Other         1
6 bpa           2
7 dioxin        2
```

case_when() made an improvement

But we still might miss a strange value - like a misspelling

```
plastics %>%  
  mutate(microplastic_recoded = case_when(  
    microplastic %in% c("Dioxin", "dioxin") ~ "Dioxin",  
    microplastic %in% c("BPA", "bpa", "Bisphenol A") ~ "BPA",  
    microplastic %in% c("O", "Other") ~ "Other",  
    TRUE ~ microplastic))
```


case_when() improved with stringr

^ indicates the beginning of a character string \$ indicates the end

```
plastics %>%  
  mutate(microplastic_recoded = case_when(  
    str_detect(string = microplastic, pattern = "^b|B") ~ "BPA",  
    str_detect(string = microplastic, pattern = "^o|^O") ~ "Other",  
    str_detect(string = microplastic, pattern = "^d|^D") ~ "Dioxin",  
    TRUE ~ microplastic)) %>%  
  count(microplastic, microplastic_recoded)
```

```
# A tibble: 7 × 3  
  microplastic microplastic_recoded     n  
  <chr>         <chr>         <int>  
1 BPA          BPA             2  
2 Bisphenol A  BPA             1  
3 Dioxin       Dioxin          2  
4 O           Other           2  
5 Other       Other           1  
6 bpa        BPA             2  
7 dioxin     Dioxin          2
```

This is a more robust solution! It will catch typos as long as the first letter is correct.

That's better!



GUT CHECK: What **stringr** function helps us find a string pattern?

A. `str_replace()`

B. `str_find()`

C. `str_detect()`

Separating and uniting data

Uniting columns

The `unite()` function can help combine columns

The `col` argument specifies new column name

The `sep` argument specifies what separator to use when combining -default is "_"

The `remove` argument specifies if you want to drop the old columns

```
plastics_comb <- plastics %>%  
  unite(Foods, Effect, col = "change", remove = TRUE)
```

```
plastics_comb
```

```
# A tibble: 12 × 4
```

	change <chr>	microplastic <chr>	blood_level_start_nM <dbl>	blood_level_change_nM <dbl>
1	A_Increase	Dioxin	1	2.5
2	B_Increase	Dioxin	7	2.5
3	B_Increase	Other	2	0.5
4	A_Decrease	Bisphenol A	3	-0.5
5	B_Increase	BPA	5	0.5
6	B_Increase	dioxin	8	0.5
7	A_Increase	bpa	6	2.5
8	B_Increase	0	5	3.5
9	B_Increase	dioxin	2	0.5
10	A_Increase	bpa	1	1.5
11	B_Increase	BPA	7	1.5
12	B_Increase	0	3	2.5

Separating columns based on a separator

The `separate()` function from `tidyr` can split a column into multiple columns.

The `col` argument specifies what column to work with

The `into` argument specifies names of new columns

The `sep` argument specifies what to separate by

```
plastics_comb <- plastics_comb %>%  
  separate(col = change, into = c("Foods", "Change"), sep = "_" )  
plastics_comb
```

```
# A tibble: 12 × 5
```

	Foods <chr>	Change <chr>	microplastic <chr>	blood_level_start_nM <dbl>	blood_level_change_nM <dbl>
1	A	Increase	Dioxin	1	2.5
2	B	Increase	Dioxin	7	2.5
3	B	Increase	Other	2	0.5
4	A	Decrease	Bisphenol A	3	-0.5
5	B	Increase	BPA	5	0.5
6	B	Increase	dioxin	8	0.5
7	A	Increase	bpa	6	2.5
8	B	Increase	0	5	3.5
9	B	Increase	dioxin	2	0.5
10	A	Increase	bpa	1	1.5
11	B	Increase	BPA	7	1.5
12	B	Increase	0	3	2.5

Summary

- `case_when()` requires `mutate()` when working with dataframes/tibbles
- `case_when()` can recode **entire values** based on **conditions** (need quotes for conditions and new values)
 - remember `case_when()` needs `TRUE ~ variable` to keep values that aren't specified by conditions, otherwise will be NA

Note: you might see the `recode()` function, it only does some of what `case_when()` can do, so we skipped it, but it is in the extra slides at the end.

Summary continued

dplyr::case_when() IF ELSE...
(but you love it?)

df %>% ^{ADD COLUMN 'danger'}
mutate(danger = case_when(
 IF type is kraken THEN danger is extreme!
 TRUE ~ "high")
 OTHERWISE, danger is high.



@allison_horst

"Artwork by @allison_horst". <https://allisonhorst.com/>

Summary Continued

- `stringr` package has great functions for looking for specific **parts of values** especially `filter()` and `str_detect()` combined
- `stringr` also has other useful string functions like `str_detect()` (finding patterns in a column or vector), `str_subset()` (parsing text), `str_replace()` (replacing the first instance in values), `str_replace_all()` (replacing all instances in each value) and **more!**
- `separate()` can split columns into additional columns
- `unite()` can combine columns
- `:` can indicate when you want to start and end with columns next to one another

Lab Part 2

▮ [Class Website](#)

▮ [Lab.](#) ▮ [Day 5 Cheatsheet](#) ▮ [Posit's stringr Cheatsheet](#)



Image by [Gerd Altmann](#) from [Pixabay](#)

Extra Slides

recode() function

This is similar to `case_when()` but it can't do as much.

(need mutate for data frames/tibbles!)

General Format - this is not code!

```
{data_input} %>%
```

```
  mutate({variable_to_fix_or_new} = recode({Variable_fixing}, {old_value} = {new_value},  
                                           {another_old_value} = {new_value}))
```

recode() function

Need quotes for new values! Tolerates quotes for old values.

```
plastics %>%  
  mutate(microplastic_recoded = recode(microplastic,  
    "Bisphenol A" = "BPA",  
    "bpa" = "BPA",  
    "dioxin" = "Dioxin",  
    "0" = "Other")) %>%  
  count(microplastic, microplastic_recoded)
```

recode()

```
plastics %>%
  mutate(microplastic_recoded = recode(microplastic,
    "Bisphenol A" = "BPA",
    "bpa" = "BPA",
    "dioxin" = "Dioxin",
    "0" = "Other")) %>%
  count(microplastic, microplastic_recoded)
```

```
# A tibble: 7 × 3
  microplastic microplastic_recoded     n
  <chr>         <chr>                 <int>
1 BPA          BPA                     2
2 Bisphenol A  BPA                     1
3 Dioxin       Dioxin                   2
4 0            Other                    2
5 Other        Other                    1
6 bpa          BPA                     2
7 dioxin       Dioxin                   2
```

Can update or overwrite variables with recode too!

Just use the same variable name to change the variable within mutate.

```
plastics %>%  
  mutate(microplastic = recode(microplastic,  
                                "Bisphenol A" = "BPA",  
                                "bpa" = "BPA",  
                                "dioxin" = "Dioxin",  
                                "0" = "Other")) %>%  
  count(microplastic)
```

```
# A tibble: 3 × 2  
  microplastic      n  
  <chr>         <int>  
1 BPA             5  
2 Dioxin          4  
3 Other           3
```

String Splitting

- `str_split(string, pattern)` - splits strings up - returns list!

```
library(stringr)
x <- c("I really like writing R code")
df <- tibble(x = c("I really", "like writing", "R code programs"))
y <- unlist(str_split(x, " "))
y
```

```
[1] "I"          "really"    "like"      "writing"   "R"         "code"
```

```
length(y)
```

```
[1] 6
```


A bit on Regular Expressions

- <http://www.regular-expressions.info/reference.html>
- They can use to match a large number of strings in one statement
- `.` matches any single character
- `*` means repeat as many (even if 0) more times the last character
- `?` makes the last thing optional
- `^` matches start of vector `^a` - starts with "a"
- `$` matches end of vector `b$` - ends with "b"

Let's look at modifiers for **stringr**

?modifiers

- `fixed` - match everything exactly
- `ignore_case` is an option to not have to use `tolower`

Using a fixed expression

One example case is when you want to split on a period ".". In regular expressions . means **ANY** character, so we need to specify that we want R to interpret "." as simply a period.

```
str_split("I.like.strings", ".")
```

```
[[1]]  
[1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
```

```
str_split("I.like.strings", fixed("."))
```

```
[[1]]  
[1] "I"      "like"    "strings"
```

```
str_split("I.like.strings", "\\.")
```

```
[[1]]  
[1] "I"      "like"    "strings"
```

Pasting strings with `paste` and `paste0`

Paste can be very useful for joining vectors together:

```
paste("Visit", 1:5, sep = "_")
```

```
[1] "Visit_1" "Visit_2" "Visit_3" "Visit_4" "Visit_5"
```

```
paste("Visit", 1:5, sep = "_", collapse = "_")
```

```
[1] "Visit_1_Visit_2_Visit_3_Visit_4_Visit_5"
```

and paste0 can be even simpler see ?paste0

```
paste0("Visit", 1:5) # no space!
```

```
[1] "Visit1" "Visit2" "Visit3" "Visit4" "Visit5"
```

Comparison of **stringr** to base R -
not covered

Splitting Strings

Substringing

stringr

- `str_split(string, pattern)` - splits strings up - returns list!

Splitting String:

In `stringr`, `str_split` splits a vector on a string into a list

```
library(stringr)
x <- c("I really", "like writing", "R code programs")
y <- str_split(x, pattern = " ") # returns a list
y
```

```
[[1]]
[1] "I"      "really"
```

```
[[2]]
[1] "like"    "writing"
```

```
[[3]]
[1] "R"        "code"     "programs"
```


'Find' functions: stringr compared to base R

Base R does not use these functions. Here is a "translator" of the `stringr` function to base R functions

- `str_detect` - similar to `grep1` (return logical)
- `grep(value = FALSE)` is similar to `which(str_detect())`
- `str_subset` - similar to `grep(value = TRUE)` - return value of matched
- `str_replace` - similar to `sub` - replace one time
- `str_replace_all` - similar to `gsub` - replace many times

Important Comparisons

Base R:

- Argument order is (pattern, x)
- Uses option (fixed = TRUE)

stringr

- Argument order is (string, pattern) aka (x, pattern)
- Uses function fixed(pattern)

some data to work with

```
Sal <- read_csv(file =  
  "https://daseh.org/data/Baltimore_City_Employee_Salaries_FY2015.csv")
```

Showing difference in `str_extract`

`str_extract` extracts just the matched string

```
ss <- str_extract(Sal$Name, "Rawling")
```

Warning: Unknown or uninitialised column: `Name`.

```
head(ss)
```

```
character(0)
```

```
ss[ !is.na(ss)]
```

```
character(0)
```

Showing difference in `str_extract` and `str_extract_all`

`str_extract_all` extracts all the matched strings

```
head(str_extract(Sal$AgencyID, "\\d"))
```

```
[1] "0" "2" "6" "9" "4" "9"
```

```
head(str_extract_all(Sal$AgencyID, "\\d"), 2)
```

```
[[1]]  
[1] "0" "3" "0" "3" "1"
```

```
[[2]]  
[1] "2" "9" "0" "4" "5"
```

Using Regular Expressions

- Look for any name that starts with:
 - Payne at the beginning,
 - Leonard and then an S
 - Spence then capital C

```
head(grep("^Payne.*", x = Sal$name, value = TRUE), 3)
```

```
[1] "Payne El,Boaz L"      "Payne El,Jackie"  
[3] "Payne Johnson,Nickole A"
```

```
head(grep("Leonard.?S", x = Sal$name, value = TRUE))
```

```
[1] "Payne,Leonard S"      "Szumlanski,Leonard S"
```

```
head(grep("Spence.*C.*", x = Sal$name, value = TRUE))
```

```
[1] "Spencer,Charles A"    "Spencer,Clarence W"  "Spencer,Michael C"
```

Using Regular Expressions: `stringr`

```
head(str_subset( Sal$name, "^Payne.*"), 3)
```

```
[1] "Payne El,Boaz L"      "Payne El,Jackie"  
[3] "Payne Johnson,Nickole A"
```

```
head(str_subset( Sal$name, "Leonard.?S"))
```

```
[1] "Payne,Leonard S"      "Szumlanski,Leonard S"
```

```
head(str_subset( Sal$name, "Spence.*C.*"))
```

```
[1] "Spencer,Charles A"    "Spencer,Clarence W"  "Spencer,Michael C"
```

Replace

Let's say we wanted to sort the data set by Annual Salary:

```
class(Sal$AnnualSalary)
```

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

```
sort(c("1", "2", "10")) # not sort correctly (order simply ranks the data)
```

```
[1] "1" "10" "2"
```

```
order(c("1", "2", "10"))
```

```
[1] 1 3 2
```


Replace

So we must change the annual pay into a numeric:

```
head(Sal$AnnualSalary, 4)
```

```
[1] "$55314.00" "$74000.00" "$64500.00" "$46309.00"
```

```
head(as.numeric(Sal$AnnualSalary), 4)
```

```
Warning in head(as.numeric(Sal$AnnualSalary), 4): NAs introduced by coercion
```

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

R didn't like the \$ so it thought turned them all to NA.

`sub()` and `gsub()` can do the replacing part in base R.

Replacing and subbing

Now we can replace the \$ with nothing (used `fixed=TRUE` because \$ means ending):

```
Sal$AnnualSalary <- as.numeric(gsub(pattern = "$", replacement="",  
                                   Sal$AnnualSalary, fixed=TRUE))  
Sal <- Sal[order(Sal$AnnualSalary, decreasing=TRUE), ]  
Sal[1:5, c("name", "AnnualSalary", "JobTitle")]
```

```
# A tibble: 5 × 3
```

	name <chr>	AnnualSalary <dbl>	JobTitle <chr>
1	Mosby, Marilyn J	238772	STATE'S ATTORNEY
2	Batts, Anthony W	211785	Police Commissioner
3	Wen, Leana	200000	Executive Director III
4	Raymond, Henry J	192500	Executive Director III
5	Swift, Michael	187200	CONTRACT SERV SPEC II

Replacing and subbing: **stringr**

We can do the same thing (with 2 piping operations!) in dplyr

```
dplyr_sal <- Sal
dplyr_sal <- dplyr_sal %>% mutate(
  AnnualSalary = AnnualSalary %>%
    str_replace(
      fixed("$"),
      "" ) %>%
    as.numeric) %>%
  arrange(desc(AnnualSalary))
check_Sal = Sal
rownames(check_Sal) = NULL
all.equal(check_Sal, dplyr_sal)
```

```
[1] TRUE
```

Creating Two-way Tables

A two-way table. If you pass in 2 vectors, `table` creates a 2-dimensional table.

```
tab <- table(c(0, 1, 2, 3, 2, 3, 3, 2, 2, 3),  
             c(0, 1, 2, 3, 2, 3, 3, 4, 4, 3),  
             useNA = "always")
```

tab

	0	1	2	3	4	<NA>
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	2	0	2	0
3	0	0	0	4	0	0
<NA>	0	0	0	0	0	0

Creating Two-way Tables

```
tab_df <- tibble(x = c(0, 1, 2, 3, 2, 3, 3, 2, 2, 3),  
                 y = c(0, 1, 2, 3, 2, 3, 3, 4, 4, 3))  
tab_df %>% count(x, y)
```

```
# A tibble: 5 × 3  
      x     y     n  
  <dbl> <dbl> <int>  
1     0     0     1  
2     1     1     1  
3     2     2     2  
4     2     4     2  
5     3     3     4
```

Creating Two-way Tables

```
tab_df %>%  
  count(x, y) %>%  
  group_by(x) %>% mutate(pct_x = n / sum(n))
```

```
# A tibble: 5 × 4
```

```
# Groups:   x [4]
```

	x	y	n	pct_x
	<dbl>	<dbl>	<int>	<dbl>
1	0	0	1	1
2	1	1	1	1
3	2	2	2	0.5
4	2	4	2	0.5
5	3	3	4	1

Creating Two-way Tables

```
library(scales)
tab_df %>%
  count(x, y) %>%
  group_by(x) %>% mutate(pct_x = percent(n / sum(n)))
```

```
# A tibble: 5 × 4
# Groups:   x [4]
      x     y     n pct_x
  <dbl> <dbl> <int> <chr>
1     0     0     1 100%
2     1     1     1 100%
3     2     2     2  50%
4     2     4     2  50%
5     3     3     4 100%
```

Removing columns with threshold of percent missing values

```
is.na(df) %>% head(n = 3)
```

```
      x  
[1,] FALSE  
[2,] FALSE  
[3,] FALSE
```

```
colMeans(is.na(df))#TRUE and FALSE treated like 0 and 1
```

```
x  
0
```

```
which(colMeans(is.na(df)) < 0.2) #the location of the columns <.2
```

```
x  
1
```

```
df %>% select(which(colMeans(is.na(df)) < 0.2))# remove if over 20% missing
```

```
# A tibble: 3 × 1
```

```
      x  
  <chr>  
1 I really  
2 like writing  
3 R code programs
```