

# 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)

▢ [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!

▮ [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.

```
library(tidyverse)
bike <- read_csv(file =
  "http://jhudatascience.org/intro_to_r/data/Bike_Lanes.csv")
bike %>% count(subType)
```

```
# A tibble: 4 × 2
  subType      n
  <chr>    <int>
1 STCLN         1
2 STRALY        3
3 STRPRD     1623
4 <NA>          4
```

# 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(bike)
```

```
[1] 89.21589
```

Or for a particular variable:

```
bike%>% select(route) %>%  
pct_complete()
```

```
[1] 22.19497
```

## naniar:miss\_var\_summary()

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

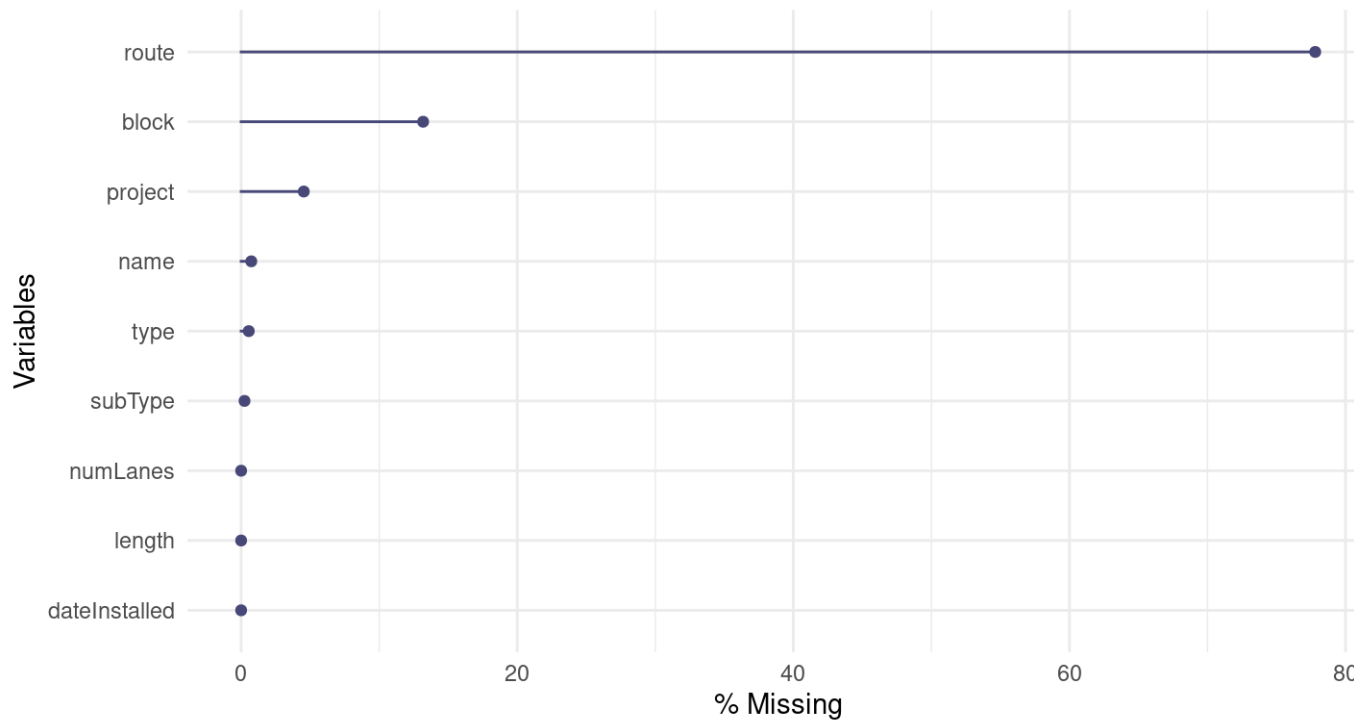
```
miss_var_summary(bike)
```

```
# A tibble: 9 × 3
  variable      n_miss pct_miss
  <chr>         <int>   <num>
1 route         1269    77.8
2 block          215    13.2
3 project         74     4.54
4 name           12     0.736
5 type            9     0.552
6 subType         4     0.245
7 numLanes        0         0
8 length          0         0
9 dateInstalled  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). Using `show_pct = TRUE` shows the percent missing.

```
gg_miss_var(bike, 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.

```
bike %>% filter(subType == "STCLN")
```

```
# A tibble: 1 × 9
```

```
  subType name          block  type  numLanes project route  length dateInstalled
  <chr>   <chr>          <chr> <chr>  <dbl> <chr>  <chr> <dbl>          <dbl>
1 STCLN  EDMONDSON AVE 5300 ... BIKE...      1 OPERAT... <NA>    181.          2011
```

## filter() and missing data

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

```
bike %>% filter(subType == "STCLN" | is.na(subType))
```

```
# A tibble: 5 × 9
```

	subType	name	block	type	numLanes	project	route	length	dateInstalled
	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>	<dbl>
1	<NA>	<NA>	<NA>	BIKE...	1	GUILFO...	<NA>	436.	0
2	<NA>	<NA>	<NA>	SIDE...	1	<NA>	NORT...	1025.	2010
3	<NA>	<NA>	<NA>	SIGN...	1	SOUTHE...	<NA>	3749.	2010
4	<NA>	HUNTINGDON PA...	<NA>	SIDE...	1	<NA>	<NA>	0	0
5	STCLN	EDMONDSON AVE	5300...	BIKE...	1	OPERAT...	<NA>	181.	2011

## 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(bike)
```

```
[1] 1631    9
```

```
bike_drop <- bike %>% drop_na(route)
dim(bike_drop)
```

```
[1] 362    9
```

# Let's take a look

Can still have NAs for other columns

bike\_drop

```
# A tibble: 362 × 9
```

	subType	name	block	type	numLanes	project	route	length	dateInstalled
	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>	<dbl>
1	<NA>	<NA>	<NA>	SIDE...	1	<NA>	NORT...	1025.	2010
2	STRALY	WINSTON-ROSS...	1200...	SIGN...	2	COLLEG...	COLL...	148.	2007
3	STRALY	WINSTON-ROSS...	1200...	SIGN...	2	COLLEG...	COLL...	366.	2007
4	STRALY	WINSTON-STON...	1200...	SIGN...	2	COLLEG...	COLL...	262.	2007
5	STRPRD	<NA>	<NA>	SIGN...	1	COLLEG...	COLL...	49.3	2007
6	STRPRD	<NA>	<NA>	SIGN...	1	COLLEG...	COLL...	70.0	2007
7	STRPRD	<NA>	<NA>	SIGN...	1	COLLEG...	COLL...	765.	2007
8	STRPRD	<NA>	<NA>	SIGN...	2	COLLEG...	COLL...	170.	2007
9	STRPRD	<NA>	<NA>	SIGN...	2	COLLEG...	COLL...	1724.	2007
10	STRPRD	ALBEMARLE ST	100 ...	SIGN...	1	SOUTHE...	LITT...	250.	2011

```
# [ 352 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.

```
bike_drop <- bike %>% drop_na()  
bike_drop
```

```
# A tibble: 257 × 9
```

	subType	name	block	type	numLanes	project	route	length	dateInstalled
	<chr>	<chr>	<chr>	<chr>	<dbl>	<chr>	<chr>	<dbl>	<dbl>
1	STRALY	WINSTON-ROSS...	1200...	SIGN...	2	COLLEG...	COLL...	148.	2007
2	STRALY	WINSTON-ROSS...	1200...	SIGN...	2	COLLEG...	COLL...	366.	2007
3	STRALY	WINSTON-STON...	1200...	SIGN...	2	COLLEG...	COLL...	262.	2007
4	STRPRD	ALBEMARLE ST	100 ...	SIGN...	1	SOUTHE...	LITT...	250.	2011
5	STRPRD	ALBEMARLE ST	100 ...	SIGN...	1	SOUTHE...	LITT...	257.	2011
6	STRPRD	ALBEMARLE ST	200 ...	SIGN...	1	SOUTHE...	LITT...	251.	2011
7	STRPRD	ALBEMARLE ST	200 ...	SIGN...	1	SOUTHE...	LITT...	252.	2011
8	STRPRD	ALBEMARLE ST	300 ...	SIGN...	1	SOUTHE...	LITT...	252.	2011
9	STRPRD	ALBEMARLE ST	UNIT...	SIGN...	1	SOUTHE...	LITT...	130.	2011
10	STRPRD	ALBEMARLE ST	UNIT...	SIGN...	1	SOUTHE...	LITT...	143.	2011

```
# [ 247 more rows
```

## Drop columns with any missing values

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

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

```
[1] "subType" "name"      "block"    "type"     "project" "route"
```

# Drop columns with any missing values

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

```
bike_drop <- bike %>% select(!miss_var_which(bike))  
bike_drop
```

```
# A tibble: 1,631 × 3  
  numLanes length dateInstalled  
  <dbl>   <dbl>         <dbl>  
1         1  436.             0  
2         1 1025.          2010  
3         1 3749.          2010  
4         1    0             0  
5         1  181.          2011  
6         2  148.          2007  
7         2  366.          2007  
8         2  262.          2007  
9         1  696.          2009  
10        1  43.1          2007  
# 1,621 more rows
```

## Change a value to be **NA**

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

```
count(bike, dateInstalled)
```

```
# A tibble: 9 × 2
  dateInstalled     n
  <dbl> <int>
1         0     126
2      2006         2
3      2007     368
4      2008     206
5      2009      86
6      2010     625
7      2011     101
8      2012     107
9      2013      10
```



## Change a value to be **NA**

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

```
bike <- bike %>%  
  mutate(dateInstalled = na_if(dateInstalled, 0))  
count(bike, dateInstalled)
```

```
# A tibble: 9 × 2  
  dateInstalled     n  
  <dbl> <int>  
1     2006         2  
2     2007        368  
3     2008        206  
4     2009         86  
5     2010        625  
6     2011        101  
7     2012        107  
8     2013         10  
9         NA        126
```

## 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)

```
bike %>%  
  mutate(dateInstalled = replace_na(dateInstalled, 2005)) %>%  
  count(dateInstalled)
```

```
# A tibble: 9 × 2  
  dateInstalled    n  
    <dbl> <int>  
1     2005    126  
2     2006     2  
3     2007    368  
4     2008    206  
5     2009     86  
6     2010    625  
7     2011    101  
8     2012    107  
9     2013     10
```

## 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(bike, dateInstalled) %>% mutate(percent = (n/(sum(n)) *100))
```

```
# A tibble: 9 × 3
```

	dateInstalled	n	percent
	<dbl>	<int>	<dbl>
1	2006	2	0.123
2	2007	368	22.6
3	2008	206	12.6
4	2009	86	5.27
5	2010	625	38.3
6	2011	101	6.19
7	2012	107	6.56
8	2013	10	0.613
9	NA	126	7.73

## Word of caution - Percentages with NA

```
bike %>% drop_na(dateInstalled) %>%  
  count(dateInstalled) %>% mutate(percent = (n/(sum(n)) *100))
```

```
# A tibble: 8 × 3
```

	dateInstalled	n	percent
	<dbl>	<int>	<dbl>
1	2006	2	0.133
2	2007	368	24.5
3	2008	206	13.7
4	2009	86	5.71
5	2010	625	41.5
6	2011	101	6.71
7	2012	107	7.11
8	2013	10	0.664

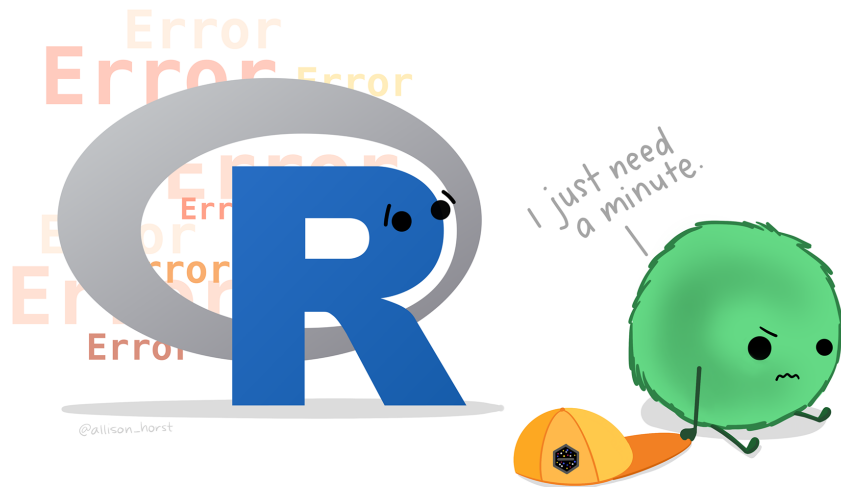
Should you be dividing by the total count with NA values included?

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

```
set.seed(124)
data_ginger_mint <- tibble(Group = rep(c("A", "B", "B"),
                                     times = 4),
                          Treatment = c("Ginger",
                                        "Ginger",
                                        "Other",
                                        "peppermint",
                                        "peppermint",
                                        "Ginger",
                                        "Mint",
                                        "0",
                                        "Ginger",
                                        "mint",
                                        "Mint",
                                        "0"),
                          Measurement_start = sample(70:250, size = 12),
                          Measurement_change = sample(-20:40, size = 12))
```

## Reading in the data if it were an excel sheet

Data is also here:

[http://jhudatascience.org/intro\\_to\\_r/data/cleaning\\_data.xlsx](http://jhudatascience.org/intro_to_r/data/cleaning_data.xlsx)

```
library(readxl)
data_ginger_mint<- read_excel(here::here("data", "cleaning_data.xlsx"))
```

Say we have some data about samples in a study:

```
data_ginger_mint
```

```
# A tibble: 12 × 4
```

```
  Group Treatment Measurement_start Measurement_change
  <chr> <chr>           <int>             <int>
1 A     Ginger         134                39
2 B     Ginger         236                17
3 B     Other           224                13
4 A     peppermint        74                36
5 B     peppermint       203               -19
6 B     Ginger           242               -14
7 A     Mint             143                29
8 B     0                230               -16
9 B     Ginger           212                 5
10 A    mint             160                 9
11 B    Mint             89                -13
12 B    0                91                 6
```



# Oh dear...

This needs lots of recoding.

```
data_ginger_mint %>%  
  count(Treatment)
```

```
# A tibble: 6 × 2  
  Treatment      n  
  <chr>      <int>  
1 Ginger         4  
2 Mint           2  
3 0              2  
4 Other          1  
5 mint           1  
6 peppermint     2
```

## **dp1yr can help!**

Using Excel to find all of the different ways Treatment 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!

```
data_ginger_mint %>%  
  mutate(Treatment_recoded = case_when(  
    Treatment == "0" ~ "Other",  
    Treatment == "Mint" ~ "Peppermint",  
    Treatment == "mint" ~ "Peppermint",  
    Treatment == "peppermint" ~ "Peppermint")) %>%  
  count(Treatment, Treatment_recoded)
```

```
# A tibble: 6 × 3
```

	Treatment	Treatment_recoded	n
	<chr>	<chr>	<int>
1	Ginger	<NA>	4
2	Mint	Peppermint	2
3	0	Other	2
4	Other	<NA>	1
5	mint	Peppermint	1
6	peppermint	Peppermint	2

# What happened?

We seem to have NA values!

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

```
data_ginger_mint %>%  
  mutate(Treatment = case_when(  
    Treatment == "O" ~ "Other",  
    Treatment == "Mint" ~ "Peppermint",  
    Treatment == "mint" ~ "Peppermint",  
    Treatment == "peppermint" ~ "Peppermint"))
```

## 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

```
data_ginger_mint %>%
  mutate(Treatment_recoded = case_when(
    Treatment == "0" ~ "Other",
    Treatment == "Mint" ~ "Peppermint",
    Treatment == "mint" ~ "Peppermint",
    Treatment == "peppermint" ~ "Peppermint",
    TRUE ~ Treatment)) %>%
  count(Treatment, Treatment_recoded)
```

# A tibble: 6 × 3

	Treatment	Treatment_recoded	n
	<chr>	<chr>	<int>
1	Ginger	Ginger	4
2	Mint	Peppermint	2
3	0	Other	2
4	Other	Other	1
5	mint	Peppermint	1
6	peppermint	Peppermint	2

## 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.

```
data_ginger_mint %>%  
  mutate(Treatment_recoded = case_when(  
    Treatment == "0" ~ "Other",  
    Treatment == "Mint" ~ "Peppermint",  
    Treatment == "mint" ~ "Peppermint",  
    Treatment == "peppermint" ~ "Peppermint",  
    TRUE ~ Treatment)) %>%  
  count(Treatment, Treatment_recoded)
```



## But maybe we want NA?

Perhaps we want values that are O or Other to actually be NA, then `case_when` can be helpful for this. We simply specify everything else.

```
data_ginger_mint %>%
  mutate(Treatment_recoded = case_when(
    Treatment == "Ginger" ~ "Ginger",
    Treatment == "Mint" ~ "Peppermint",
    Treatment == "mint" ~ "Peppermint",
    Treatment == "peppermint" ~ "Peppermint")) %>%
  count(Treatment, Treatment_recoded)
```

```
# A tibble: 6 × 3
  Treatment Treatment_recoded     n
  <chr>      <chr>          <int>
1 Ginger    Ginger          4
2 Mint      Peppermint      2
3 O         <NA>            2
4 Other     <NA>            1
5 mint      Peppermint      1
6 peppermint Peppermint      2
```

## case\_when() can also overwrite/update a variable

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

```
data_ginger_mint %>%
  mutate(Treatment = case_when(
    Treatment == "Ginger" ~ "Ginger",
    Treatment == "Mint" ~ "Peppermint",
    Treatment == "mint" ~ "Peppermint",
    Treatment == "peppermint" ~ "Peppermint")) %>%
  count(Treatment)
```

# A tibble: 3 × 2

	Treatment	n
	<chr>	<int>
1	Ginger	4
2	Peppermint	5
3	<NA>	3

## More complicated case\_when()

case\_when can do complicated statements and can match many patterns at a time.

```
data_ginger_mint %>%
  mutate(Treatment_recoded = case_when(
    Treatment == "Ginger" ~ "Ginger", # keep it the same!
    Treatment %in%
c("Mint", "mint", "Peppermint", "peppermint") ~ "Peppermint",
    Treatment %in% c("0", "Other") ~ "Other")) %>%

  count(Treatment, Treatment_recoded)
```

```
# A tibble: 6 × 3
  Treatment Treatment_recoded     n
  <chr>      <chr>          <int>
1 Ginger     Ginger           4
2 Mint       Peppermint       2
3 0          Other            2
4 Other      Other            1
5 mint       Peppermint       1
6 peppermint Peppermint       2
```

## Another reason for `case_when()`

`case_when` can do very sophisticated comparisons!

Here we create a new variable called `Effect`.

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

```
head(data_ginger_mint)
```

```
# A tibble: 6 × 5  
  Group Treatment Measurement_start Measurement_change Effect  
  <chr> <chr>           <int>           <int> <chr>  
1 A     Ginger           134             39 Increase  
2 B     Ginger           236             17 Increase  
3 B     Other             224             13 Increase  
4 A     peppermint        74              36 Increase  
5 B     peppermint       203            -19 Decrease  
6 B     Ginger           242            -14 Decrease
```

## Now it is easier to see what is happening

```
data_ginger_mint %>%  
  count(Group, Effect)
```

```
# A tibble: 3 × 3
```

```
  Group Effect      n  
  <chr> <chr>   <int>  
1 A     Increase  4  
2 B     Decrease  4  
3 B     Increase  4
```

**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(data_ginger_mint) %>% head(n = 6)
```

```
Effect
```

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

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

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

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

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

## **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" "Increase" "decrease" "decrease"
```

**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" "IncrEase" "DEcrease" "DEcrease"
```

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

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

## 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" "Inc" "Dec" "Dec"
```

## filter and stringr functions

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

```
# A tibble: 4 × 5
  Group Treatment Measurement_start Measurement_change Effect
<chr> <chr>          <int>          <int> <chr>
1 A      Ginger          134             39 Increase
2 B      Ginger          236             17 Increase
3 B      Other            224             13 Increase
4 A      peppermint         74             36 Increase
```

```
data_ginger_mint %>%
  filter(str_detect(string = Treatment,
                    pattern = "int"))
```

```
# A tibble: 5 × 5
  Group Treatment Measurement_start Measurement_change Effect
<chr> <chr>          <int>          <int> <chr>
1 A      peppermint         74             36 Increase
2 B      peppermint        203            -19 Decrease
3 A      Mint             143             29 Increase
4 A      mint             160              9 Increase
5 B      Mint              89            -13 Decrease
```

## OK back to our original problem

```
count(data_ginger_mint, Treatment)
```

```
# A tibble: 6 × 2
  Treatment     n
  <chr>       <int>
1 Ginger         4
2 Mint           2
3 0              2
4 Other          1
5 mint           1
6 peppermint     2
```



## case\_when() made an improvement

But we still might miss a strange value

```
data_ginger_mint %>%  
  mutate(Treatment_recoded = case_when(  
    Treatment %in% c("G", "g", "Ginger", "ginger") ~ "Ginger",  
    Treatment %in%  
      c("Mint", "mint", "Peppermint", "peppermint") ~ "Peppermint",  
    Treatment %in% c("O", "Other") ~ "Other",  
    TRUE ~ Treatment))
```

## case\_when() improved with stringr

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

```
data_ginger_mint %>%
  mutate(Treatment_recoded = case_when(
    str_detect(string = Treatment, pattern = "int") ~ "Peppermint",
    str_detect(string = Treatment, pattern = "^o|^O") ~ "Other",
    TRUE ~ Treatment)) %>%
  count(Treatment, Treatment_recoded)
```

```
# A tibble: 6 × 3
  Treatment Treatment_recoded     n
  <chr>      <chr>          <int>
1 Ginger    Ginger          4
2 Mint      Peppermint      2
3 0         Other           2
4 Other     Other           1
5 mint     Peppermint      1
6 peppermint Peppermint      2
```

This is a more robust solution! It will catch typos as long as the first letter is correct or there is part of the word mint.

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

```
data_comb <- data_ginger_mint %>%  
  unite(Group, Effect, col = "change", remove = TRUE)
```

```
data_comb
```

```
# A tibble: 12 × 4
```

```
  change      Treatment Measurement_start Measurement_change  
  <chr>      <chr>             <int>             <int>  
1 A_Increase Ginger             134                39  
2 B_Increase Ginger             236                17  
3 B_Increase Other              224                13  
4 A_Increase peppermint          74                36  
5 B_Decrease peppermint         203               -19  
6 B_Decrease Ginger             242               -14  
7 A_Increase Mint              143                29  
8 B_Decrease 0                  230               -16  
9 B_Increase Ginger             212                 5  
10 A_Increase mint              160                 9  
11 B_Decrease Mint              89                -13  
12 B_Increase 0                  91                 6
```

## 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

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

```
# A tibble: 12 × 5
```

	Group <chr>	Change <chr>	Treatment <chr>	Measurement_start <int>	Measurement_change <int>
1	A	Increase	Ginger	134	39
2	B	Increase	Ginger	236	17
3	B	Increase	Other	224	13
4	A	Increase	peppermint	74	36
5	B	Decrease	peppermint	203	-19
6	B	Decrease	Ginger	242	-14
7	A	Increase	Mint	143	29
8	B	Decrease	0	230	-16
9	B	Increase	Ginger	212	5
10	A	Increase	mint	160	9
11	B	Decrease	Mint	89	-13
12	B	Increase	0	91	6

## 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!**  
type == "kraken" ~ "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  
                                           {another_old_value} = {new_value}))
```

## recode() function

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

```
data_ginger_mint %>%  
  mutate(Treatment_recoded = recode(Treatment,  
                                     0 = "Other",  
                                     Mint = "Peppermint",  
                                     mint = "Peppermint",  
                                     peppermint = "Peppermint")) %>%  
  count(Treatment, Treatment_recoded)
```

## recode()

```
data_ginger_mint %>%  
  mutate(Treatment_recoded = recode(Treatment,  
                                     0 = "Other",  
                                     Mint = "Peppermint",  
                                     mint = "Peppermint",  
                                     peppermint = "Peppermint")) %>%  
  count(Treatment, Treatment_recoded)
```

```
# A tibble: 6 × 3  
  Treatment Treatment_recoded     n  
  <chr>      <chr>          <int>  
1 Ginger    Ginger          4  
2 Mint     Peppermint      2  
3 0        Other           2  
4 Other    Other           1  
5 mint     Peppermint      1  
6 peppermint Peppermint      2
```

## Can update or overwrite variables with recode too!

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

```
data_ginger_mint %>%  
  mutate(Treatment= recode(Treatment,  
                            0 = "Other",  
                            Mint = "Peppermint",  
                            mint = "Peppermint",  
                            peppermint = "Peppermint")) %>%  
  count(Treatment)
```

```
# A tibble: 3 × 2  
  Treatment      n  
  <chr>         <int>  
1 Ginger         4  
2 Other          3  
3 Peppermint     5
```



## String Splitting

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

```
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("https://jhudatascience.org/intro_to_r/data/Baltimore_City_Emplo
```

## 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           AnnualSalary JobTitle
<chr>           <dbl> <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
```