Intro to R

Data Cleaning

Recap on summarization

- `summary(x)`: quantile information
- `summarize`: creates a summary table of columns of interest
- `count(variable)`: how many of each unique value do you have
- `group_by()`: changes all subsequent functions
    - combine with `summarize()` to get statistics per group
- `plot()` and `hist()` are great for a **quick snapshot** of the data

Cheatsheet

Recap on data classes

- tibbles show column classes!
- `as.CLASS_NAME(x)` can be used to change the class of an object x
- `class()` can test what class an object is
- Logic class objects only have `TRUE` or `False` (without quotes)
- Two kinds of numeric subclasses - integer (whole numbers) and double (fractional values)
- Character class values need quotes
- Factors are a special character class that has levels
- matrix has columns and rows but is all one data class
- lists can contain multiples of any other class of data including lists!
- The `lubridate` package is helpful for dates and times
    - 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 gives you:

1. The unique values
2. The amount of these values

Check if rare values make sense.

```
library(jhur)
bike <- read_csv(file = "http://jhudatascience.org/intro_to_r/data/Bike_Lanes.
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)
```

Air quality data

The `airquality` dataset comes with R about air quality in New York in 1973.

```
?airquality # use this to find out more about the data
airqual <- tibble(airquality)
airqual
```

```
# A tibble: 153 × 6
   Ozone Solar.R  Wind  Temp Month   Day
   <int>   <int> <dbl> <int> <int> <int>
 1    41     190   7.4    67     5     1
 2    36     118   8      72     5     2
 3    12     149  12.6    74     5     3
 4    18     313  11.5    62     5     4
 5    NA      NA  14.3    56     5     5
 6    28      NA  14.9    66     5     6
 7    23     299   8.6    65     5     7
 8    19      99  13.8    59     5     8
 9     8      19  20.1    61     5     9
10    NA     194   8.6    69     5    10
# … 143 more rows
```

naniar: `pct_complete()`

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

```
pct_complete(airqual)
```

```
[1] 95.20697
```

Or for a particular variable:

```
airqual %>% select(Ozone) %>%
pct_complete()
```

```
[1] 75.81699
```

naniar::`miss_var_summary()`

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

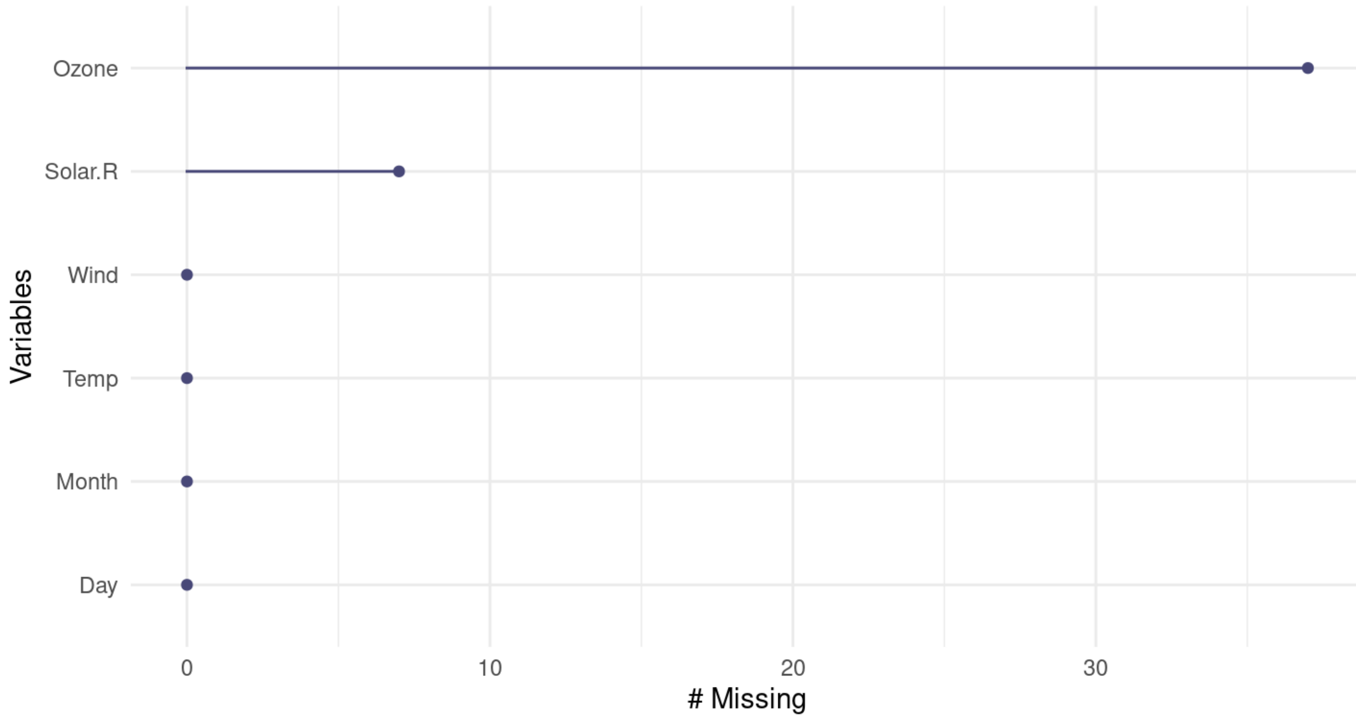
```
miss_var_summary(airqual)
```

```
# A tibble: 6 × 3
  variable n_miss pct_miss
  <chr>     <int>    <dbl>
1 Ozone        37    24.2
2 Solar.R       7     4.58
3 Wind          0     0
4 Temp          0     0
5 Month         0     0
6 Day           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(airqual)
```



Missing Data Issues

Recall that mathematical operations with `NA` often result in `NA`s.

```
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.

```
airqual %>% filter(Ozone < 5)
```

```
# A tibble: 2 × 6
  Ozone Solar.R  Wind  Temp Month   Day
  <int>   <int> <dbl> <int> <int> <int>
1     1       8   9.7    59     5    21
2     4      25   9.7    61     5    23
```

filter() and missing data

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

```
airqual %>% filter(Ozone < 5 | is.na(Ozone))
```

```
# A tibble: 39 × 6
   Ozone Solar.R  Wind  Temp Month   Day
   <int>   <int> <dbl> <int> <int> <int>
 1    NA      NA  14.3    56     5     5
 2    NA     194   8.6    69     5    10
 3     1       8   9.7    59     5    21
 4     4      25   9.7    61     5    23
 5    NA      66  16.6    57     5    25
 6    NA     266  14.9    58     5    26
 7    NA      NA   8      57     5    27
 8    NA     286   8.6    78     6     1
 9    NA     287   9.7    74     6     2
10    NA     242  16.1    67     6     3
# ℹ 29 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.

```
dim(airqual)
```

```
[1] 153   6
```

```
airqual %>% drop_na(Ozone)
```

```
# A tibble: 116 × 6
   Ozone Solar.R  Wind  Temp Month   Day
   <int>   <int> <dbl> <int> <int> <int>
 1    41     190   7.4    67     5     1
 2    36     118   8      72     5     2
 3    12     149  12.6    74     5     3
 4    18     313  11.5    62     5     4
 5    28      NA  14.9    66     5     6
 6    23     299   8.6    65     5     7
 7    19      99  13.8    59     5     8
 8     8      19  20.1    61     5     9
 9     7      NA   6.9    74     5    11
10    16     256   9.7    69     5    12
# ℹ 106 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.

```
airqual %>% drop_na()
```

```
# A tibble: 111 × 6
   Ozone Solar.R  Wind  Temp Month   Day
   <int>   <int> <dbl> <int> <int> <int>
 1    41     190   7.4    67     5     1
 2    36     118   8      72     5     2
 3    12     149  12.6    74     5     3
 4    18     313  11.5    62     5     4
 5    23     299   8.6    65     5     7
 6    19      99  13.8    59     5     8
 7     8      19  20.1    61     5     9
 8    16     256   9.7    69     5    12
 9    11     290   9.2    66     5    13
10    14     274  10.9    68     5    14
# ℹ 101 more rows
```

Drop **columns** with any missing values

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

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

```
[1] "Ozone"   "Solar.R"
```

Drop **columns** with any missing values

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

```
airqual %>% select(!miss_var_which(airqual))
```

```
# A tibble: 153 × 4
    Wind  Temp Month   Day
   <dbl> <int> <int> <int>
 1   7.4    67     5     1
 2   8      72     5     2
 3  12.6    74     5     3
 4  11.5    62     5     4
 5  14.3    56     5     5
 6  14.9    66     5     6
 7   8.6    65     5     7
 8  13.8    59     5     8
 9  20.1    61     5     9
10   8.6    69     5    10
# ℹ 143 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!

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

Lab Part 1

- Class Website
- Lab

Recoding Variables

Example of Recoding

```
set.seed(124)
data_diet <- tibble(Diet = rep(c("A", "B", "B"),
                               times = 4),
                    Treatment = c("Ginger",
                                  "Ginger",
                                  "Other",
                                  "peppermint",
                                  "peppermint",
                                  "Ginger",
                                  "Mint",
                                  "O",
                                  "Ginger",
                                  "mint",
                                  "Mint",
                                  "O"),
                    Weight_start = sample(100:250, size = 12),
                    Weight_change = sample(-10:20, size = 12))
```

Reading in the data if it were an excel sheet

Data is also here:

http://jhudatascience.org/intro_to_r/data/cleaning_diet_data.xlsx

```
library(readxl)
data_diet<- read_excel(here::here("data", "cleaning_diet_data.xlsx"))
```

Say we have some data about samples in a diet study:

```
data_diet
```

```
# A tibble: 12 × 4
   Diet  Treatment  Weight_start Weight_change
   <chr> <chr>             <dbl>         <dbl>
 1 A     Ginger              164            16
 2 B     Ginger              104            -4
 3 B     Other               233            -7
 4 A     peppermint          173             1
 5 B     peppermint          242             9
 6 B     ginger              190             6
 7 A     Mint                119            -5
 8 B     O                   121             0
 9 B     ginger              229            -1
10 A     mint                125            -3
11 B     Mint                129            13
12 B     O                   222            15
```

Oh dear…

This needs lots of recoding.

```
data_diet %>%
  count(Treatment)
```

```
# A tibble: 7 × 2
  Treatment      n
  <chr>      <int>
1 Ginger         2
2 Mint           2
3 O              2
4 Other          1
5 ginger         2
6 mint           1
7 peppermint     2
```

`dplyr` can help!

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

(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.

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

recode()

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

```
# A tibble: 7 × 3
  Treatment  Treatment_recoded     n
  <chr>      <chr>             <int>
1 Ginger     Ginger                2
2 Mint       Peppermint            2
3 O          Other                 2
4 Other      Other                 1
5 ginger     ginger                2
6 mint       Peppermint            1
7 peppermint Peppermint            2
```

Can update or overwrite variables with recode too!

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

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

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

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

```
# A tibble: 7 × 3
  Treatment  Treatment_recoded     n
  <chr>      <chr>             <int>
1 Ginger     <NA>                  2
2 Mint       Peppermint            2
3 O          Other                 2
4 Other      <NA>                  1
5 ginger     <NA>                  2
6 mint       Peppermint            1
7 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_diet %>%
  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_diet %>%
  mutate(Treatment_recoded = case_when(
                             Treatment == "O" ~ "Other",
                             Treatment == "Mint" ~ "Peppermint",
                             Treatment == "mint" ~ "Peppermint",
                             Treatment == "peppermint" ~ "Peppermint",
                              TRUE ~ Treatment)) %>%
  count(Treatment, Treatment_recoded)
```

```
# A tibble: 7 × 3
  Treatment  Treatment_recoded     n
  <chr>      <chr>             <int>
1 Ginger     Ginger                2
2 Mint       Peppermint            2
3 O          Other                 2
4 Other      Other                 1
5 ginger     ginger                2
6 mint       Peppermint            1
7 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_diet %>%
  mutate(Treatment_recoded = case_when(
                               Treatment == "O" ~ "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_diet %>%
  mutate(Treatment_recoded = case_when(Treatment == "Ginger" ~ "Ginger",
                                       Treatment == "Mint" ~ "Peppermint",
                                       Treatment == "mint" ~ "Peppermint",
                                       Treatment == "peppermint" ~ "Peppermint")) %>%
  count(Treatment, Treatment_recoded)
```

```
# A tibble: 7 × 3
  Treatment  Treatment_recoded     n
  <chr>      <chr>             <int>
1 Ginger     Ginger                2
2 Mint       Peppermint            2
3 O          <NA>                  2
4 Other      <NA>                  1
5 ginger     <NA>                  2
6 mint       Peppermint            1
7 peppermint Peppermint            2
```

case_when() can also overwrite/update a variable

Just like recode, just need to specify what we want in the first part of `mutate`.

```
data_diet %>%
  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         2
2 Peppermint     5
3 <NA>           5
```

More complicated case_when()

`case_when` can do more complicated statements than `recode` and can match many patterns at a time.

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

  count(Treatment, Treatment_recode)
```

```
# A tibble: 7 × 3
  Treatment  Treatment_recode     n
  <chr>      <chr>            <int>
1 Ginger     Ginger               2
2 Mint       Peppermint           2
3 O          Other                2
4 Other      Other                1
5 ginger     <NA>                 2
6 mint       Peppermint           1
7 peppermint Peppermint           2
```

Another reason for `case_when()`

`case_when` can do very sophisticated comparisons!

Here we create a new variable called `Effect`.

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

head(data_diet)
```

```
# A tibble: 6 × 5
  Diet  Treatment  Weight_start Weight_change Effect
  <chr> <chr>             <dbl>         <dbl> <chr>
1 A     Ginger              164            16 Increase
2 B     Ginger              104            -4 Decrease
3 B     Other               233            -7 Decrease
4 A     peppermint          173             1 Increase
5 B     peppermint          242             9 Increase
6 B     ginger              190             6 Increase
```

Now it is easier to see what is happening

```
data_diet %>%
  count(Diet, Effect)
```

```
# A tibble: 5 × 3
  Diet  Effect       n
  <chr> <chr>    <int>
1 A     Decrease     2
2 A     Increase     2
3 B     Decrease     3
4 B     Increase     4
5 B     Same         1
```

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_diet) %>% head(n = 6)
Effect
```

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

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

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

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

```
[1] FALSE  TRUE  TRUE FALSE 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" "decrease" "decrease" "Increase" "Increase" "Increase"
```

`st_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" "DEcrease" "DEcrease" "IncrEase" "IncrEase" "IncrEase"
```

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

```
[1] "IncrEasE" "DEcrEasE" "DEcrEasE" "IncrEasE" "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" "Dec" "Dec" "Inc" "Inc" "Inc"
```

`filter` and `stringr` functions

```r
head(data_diet, n = 4)
```

```
# A tibble: 4 × 5
  Diet  Treatment  Weight_start Weight_change Effect
  <chr> <chr>             <dbl>         <dbl> <chr>
1 A     Ginger              164            16 Increase
2 B     Ginger              104            -4 Decrease
3 B     Other               233            -7 Decrease
4 A     peppermint          173             1 Increase
```

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

```
# A tibble: 5 × 5
  Diet  Treatment  Weight_start Weight_change Effect
  <chr> <chr>             <dbl>         <dbl> <chr>
1 A     peppermint          173             1 Increase
2 B     peppermint          242             9 Increase
3 A     Mint                119            -5 Decrease
4 A     mint                125            -3 Decrease
5 B     Mint                129            13 Increase
```

OK back to our original problem

```
count(data_diet, Treatment)
```

```
# A tibble: 7 × 2
  Treatment      n
  <chr>      <int>
1 Ginger         2
2 Mint           2
3 O              2
4 Other          1
5 ginger         2
6 mint           1
7 peppermint     2
```

Recode was nice but what if miss something?

```
data_diet %>%
  mutate(Treatment_recoded = recode(Treatment, G = "Ginger",
                                              g = "Ginger",
                                         ginger = "Ginger",
                                              O = "Other",
                                           Mint = "Peppermint",
                                           mint = "Peppermint",
                                     peppermint = "Peppermint"))
```

`case_when()` was an improvement

But we still might miss a strange value

```
data_diet %>%
  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_diet %>%
  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: 7 × 3
  Treatment  Treatment_recoded     n
  <chr>      <chr>             <int>
1 Ginger     Ginger                2
2 Mint       Peppermint            2
3 O          Other                 2
4 Other      Other                 1
5 ginger     ginger                2
6 mint       Peppermint            1
7 peppermint Peppermint            2
```

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

That’s better!

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

```
diet_comb <- data_diet %>%
  unite(Diet, Effect, col = "change", remove = TRUE)
```

```
diet_comb
```

```
# A tibble: 12 × 4
   change     Treatment  Weight_start Weight_change
   <chr>      <chr>             <dbl>         <dbl>
 1 A_Increase Ginger              164            16
 2 B_Decrease Ginger              104            -4
 3 B_Decrease Other               233            -7
 4 A_Increase peppermint          173             1
 5 B_Increase peppermint          242             9
 6 B_Increase ginger              190             6
 7 A_Decrease Mint                119            -5
 8 B_Same     O                   121             0
 9 B_Decrease ginger              229            -1
10 A_Decrease mint                125            -3
11 B_Increase Mint                129            13
12 B_Increase O                   222            15
```

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

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

```
# A tibble: 12 × 5
   Diet  Change   Treatment  Weight_start Weight_change
   <chr> <chr>    <chr>             <dbl>         <dbl>
 1 A     Increase Ginger              164            16
 2 B     Decrease Ginger              104            -4
 3 B     Decrease Other               233            -7
 4 A     Increase peppermint          173             1
 5 B     Increase peppermint          242             9
 6 B     Increase ginger              190             6
 7 A     Decrease Mint                119            -5
 8 B     Same     O                   121             0
 9 B     Decrease ginger              229            -1
10 A     Decrease mint                125            -3
11 B     Increase Mint                129            13
12 B     Increase O                   222            15
```

Summary

- `recode()` and `case_when()` require `mutate()` when working with dataframes/tibbles
- `recode()` can help with simple recoding (not based on condition but **exact** swap) (only need quotes for new values - but it tolerates redundant quotes)
- `case_when()` can recode **entire values** based on **conditions** (need quotes for conditions and new values)
    - remember `case_when()` needs `TRUE ~ varaible` to keep values that aren't specified by conditions, otherwise will be `NA`

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

Image by Gerd Altmann from Pixabay

Extra Slides

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 `grepl` (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_salaries() # or
```

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:

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

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

```r
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
```