Intro to R

Data Cleaning

Recap on summarization

- `summary(x)`: quantile information

- `summarize`: creates a summary table of columns of interest

  - combine with `across()` to programmatically select columns

- `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 will be TRUE if ANY are true

  - any(is.na(x)) - do we have any NA's in x?

```r
A = c(1, 2, 4, NA)
B = c(1, 2, 3, 4)
any(is.na(A)) # are there any NAs - YES/TRUE
```

```
[1] TRUE
```

```r
any(is.na(B)) # are there any NAs- NO/FALSE
```

```
[1] FALSE
```

Fining **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. (You need a data frame to use this)

```
library(jhur)
bike <- read_bike()

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.

The `pct_complete()` function shows the percentage that is complete for a given data object, (vector or data frame).

```r
#install.packages("naniar")
library(naniar)
x = c(0, NA, 2, 3, 4, -0.5, 0.2)
pct_complete(x)
```

```
[1] 85.71429
```

```
test
```

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

```r
pct_complete(test) # doesn't count infinite values as missing
```

```
[1] 33.33333
```

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
# … with 143 more rows
```

naniar: `pct_complete()`

```
pct_complete(airquality)
```
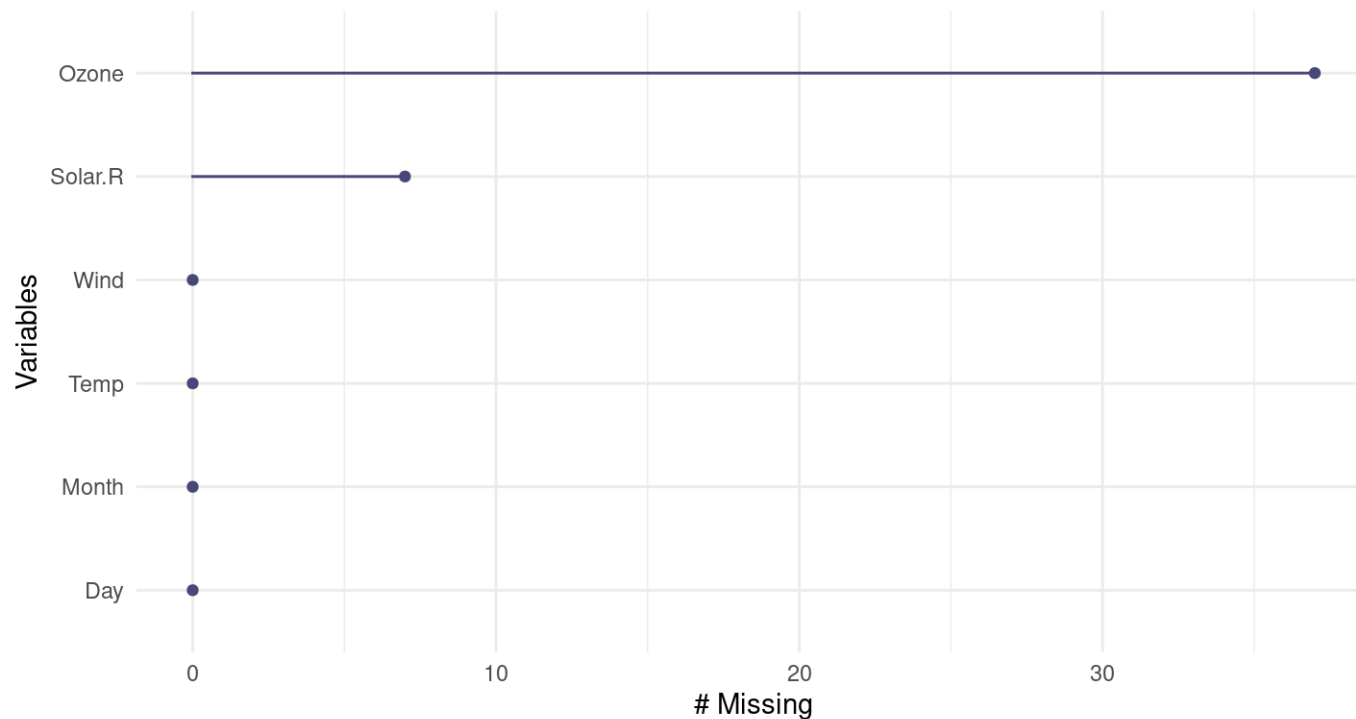
```
[1] 95.20697
```

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

```
[1] 75.81699
```

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 NAs.

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

```
[1] NA
```

```r
mean(c(2,4,NA))
```

```
[1] NA
```

```r
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.

```r
x = c(TRUE, TRUE, TRUE, TRUE, FALSE, NA)
sum(x)
```

```
[1] NA
```

```r
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

```r
df <-tibble(Dog = c(0, NA, 2, 3, 1, 1),
            Cat = c(NA, 8, 6, NA, 2, NA))
```

filter() and missing data

```
df
```

```
# A tibble: 6 × 2
    Dog   Cat
  <dbl> <dbl>
1     0    NA
2    NA     8
3     2     6
4     3    NA
5     1     2
6     1    NA
```

```
df %>% filter(Dog < 3)
```

```
# A tibble: 4 × 2
    Dog   Cat
  <dbl> <dbl>
1     0    NA
2     2     6
3     1     2
4     1    NA
```

filter() and missing data

```
df %>% filter(Dog < 3 | is.na(Dog))
```

```
# A tibble: 5 × 2
    Dog   Cat
  <dbl> <dbl>
1     0    NA
2    NA     8
3     2     6
4     1     2
5     1    NA
```

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.

```
df %>% drop_na(Dog)
```

```
# A tibble: 5 × 2
    Dog   Cat
  <dbl> <dbl>
1     0    NA
2     2     6
3     3    NA
4     1     2
5     1    NA
```

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.

```
df %>% drop_na()
```

```
# A tibble: 2 × 2
    Dog   Cat
  <dbl> <dbl>
1     2     6
2     1     2
```

Drop **columns** with any missing values

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

```
df<-df %>% mutate(test = c(1,2,3,4,5,6))
df
```

```
# A tibble: 6 × 3
    Dog   Cat  test
  <dbl> <dbl> <dbl>
1     0    NA     1
2    NA     8     2
3     2     6     3
4     3    NA     4
5     1     2     5
6     1    NA     6
```

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

```
[1] "Dog" "Cat"
```

Drop **columns** with any missing values

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

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

```
# A tibble: 6 × 1
   test
  <dbl>
1     1
2     2
3     3
4     4
5     5
6     6
```

Change a value to be NA

The `na_if()` function of `dplyr` can be helpful for this. Let's say we think that all 0 values should be NA.

```
df %>% head(n = 3)
```

```
# A tibble: 3 × 3
    Dog   Cat  test
  <dbl> <dbl> <dbl>
1     0    NA     1
2    NA     8     2
3     2     6     3
```

```
df %>% mutate(Dog = na_if(x = Dog, y = 0))
```

```
# A tibble: 6 × 3
    Dog   Cat  test
  <dbl> <dbl> <dbl>
1    NA    NA     1
2    NA     8     2
3     2     6     3
4     3    NA     4
5     1     2     5
6     1    NA     6
```

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.!

```r
red_blue <- tibble(color = rep(c("red", "blue", NA), 3))
red_blue <- red_blue %>% count(color)
red_blue <- rename(red_blue, col_count = n)
red_blue
```

```
# A tibble: 3 × 2
  color col_count
  <chr>     <int>
1 blue          3
2 red           3
3 <NA>          3
```

Word of caution - Percentages with **NA**

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

This is because the denominator changes.

```r
total <- red_blue %>% pull(col_count) %>% sum()
red_blue %>% mutate(percent =
                    (col_count/total)*100)
```

```
# A tibble: 3 × 3
  color col_count percent
  <chr>     <int>   <dbl>
1 blue          3    33.3
2 red           3    33.3
3 <NA>          3    33.3
```

Word of caution - Percentages with **NA**

```
total <- red_blue %>% drop_na() %>% pull(col_count) %>% sum()
red_blue %>% mutate(percent =
                    (col_count/total)*100)
```

```
# A tibble: 3 × 3
  color col_count percent
  <chr>     <int>   <dbl>
1 blue          3      50
2 red           3      50
3 <NA>          3      50
```

Should you be dividing by 9 or 6?

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())`, `count()`, and functions from `naniar` like `gg_miss_var()` can help determine if we have `NA` 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

```r
data_diet <- tibble(Diet = rep(c("A", "B", "B"),
                               times = 4),
                    Gender = c("Male",
                               "m",
                               "Other",
                               "F",
                               "Female",
                               "M",
                               "f",
                               "O",
                               "Man",
                               "f",
                               "F",
                               "O"),
                    Weight_start = sample(100:250, size = 12),
                    Weight_change = sample(-10:20, size = 12))
```

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

```
data_diet
```

```
# A tibble: 12 × 4
   Diet  Gender Weight_start Weight_change
   <chr> <chr>         <int>         <int>
 1 A     Male            159            13
 2 B     m               199            19
 3 B     Other           120             0
 4 A     F               201            -5
 5 B     Female          233             8
 6 B     M               108            12
 7 A     f               118             4
 8 B     O               121            10
 9 B     Man             139            11
10 A     f               128             3
11 B     F               208            16
12 B     O               137            20
```

Oh dear...

This needs lots of recoding.

```
data_diet %>%
  count(Gender)
```

```
# A tibble: 9 × 2
  Gender     n
  <chr>  <int>
1 f          2
2 F          2
3 Female     1
4 m          1
5 M          1
6 Male       1
7 Man        1
8 O          2
9 Other      1
```

`dplyr` can help!

Using Excel to find all of the different ways `gender` has been coded, could be hectic!

In `dplyr` you can use the `recode` function (need `mutate` here too!):

```
# General Format - this is not code!
{data_input} %>%
  mutate({variable_to_fix} = recode({Variable_fixing}, {old_value} = {new_valu
                                    {another_old_value} = {new_value}))
```

```
data_diet %>%
  mutate(Gender = recode(Gender, M = "Male",
                                 m = "Male",
                               Man = "Male",
                                 O = "Other",
                                 f = "Female",
                                 F = "Female")) %>%

  count(Gender, Diet)
```

recode()

```
data_diet %>%
  mutate(Gender = recode(Gender, M = "Male",
                                 m = "Male",
                               Man = "Male",
                                 O = "Other",
                                 f = "Female",
                                 F = "Female")) %>%
  count(Gender, Diet)

# A tibble: 5 × 3
  Gender Diet      n
  <chr>  <chr> <int>
1 Female A         3
2 Female B         2
3 Male   A         1
4 Male   B         3
5 Other  B         3
```

Or you can use **case_when()**

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

```
# General Format - this is not code!
{data_input} %>%
  mutate({variable_to_fix} = case_when({Variable_fixing}
                             /some condition/  ~ {value_for_con})
```

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

```
data_diet %>%
  mutate(Gender = case_when(Gender == "M" ~ "Male"))
```

```
# A tibble: 12 × 4
   Diet  Gender Weight_start Weight_change
   <chr> <chr>         <int>         <int>
 1 A     <NA>            159            13
 2 B     <NA>            199            19
 3 B     <NA>            120             0
 4 A     <NA>            201            -5
 5 B     <NA>            233             8
 6 B     Male            108            12
 7 A     <NA>            118             4
 8 B     <NA>            121            10
 9 B     <NA>            139            11
10 A     <NA>            128             3
11 B     <NA>            208            16
12 B     <NA>            137            20
```

Use of `case_when()` without automatic `NA`

::: codeexample

```
# 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})
```

Here we use the original values of `Gender` to replace all values of `Gender` that do not meet the condition `== "M"`.

```
data_diet %>%
  mutate(Gender = case_when(Gender == "M" ~ "Male",
                                TRUE ~ Gender))
```

```
# A tibble: 12 × 4
   Diet  Gender Weight_start Weight_change
   <chr> <chr>         <int>         <int>
 1 A     Male            159            13
 2 B     m               199            19
 3 B     Other           120             0
 4 A     F               201            -5
 5 B     Female          233             8
 6 B     Male            108            12
 7 A     f               118             4
 8 B     O               121            10
 9 B     Man             139            11
10 A     f               128             3
11 B     F               208            16
```

More complicated case_when()

```r
data_diet %>%
  mutate(Gender = case_when(
    Gender %in% c("M", "male", "Man", "m", "Male") ~ "Male",
    Gender %in% c("F", "Female", "f", "female") ~ "Female",
    Gender %in% c("O", "Other") ~ "Other"))
```

```
# A tibble: 12 × 4
   Diet  Gender Weight_start Weight_change
   <chr> <chr>         <int>         <int>
 1 A     Male            159            13
 2 B     Male            199            19
 3 B     Other           120             0
 4 A     Female          201            -5
 5 B     Female          233             8
 6 B     Male            108            12
 7 A     Female          118             4
 8 B     Other           121            10
 9 B     Male            139            11
10 A     Female          128             3
11 B     Female          208            16
12 B     Other           137            20
```

Another reason for **case_when( )**

`case_when` can do very sophisticated comparisons

```r
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  Gender Weight_start Weight_change Effect
  <chr> <chr>         <int>         <int> <chr>
1 A     Male            159            13 Increase
2 B     m               199            19 Increase
3 B     Other           120             0 Same
4 A     F               201            -5 Decrease
5 B     Female          233             8 Increase
6 B     M               108            12 Increase

# A tibble: 4 × 3
  Diet  Effect       n
  <chr> <chr>    <int>
1 A     Decrease     1
2 A     Increase     3
3 B     Increase     7
4 B     Same         1
```

Effect of diet A & B on participants

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

```
x<-c("cat", "dog", "mouse")
str_detect(string = x, pattern = "d")
```

```
[1] FALSE  TRUE 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

```
x<-c("cat", "dog", "mouse")
str_replace(string = x, pattern = "d", replacement = "D")

[1] "cat"   "Dog"   "mouse"
```

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

```r
x<-c("cat", "dog", "mouse")
str_sub(string = x, start = 1, end = 2)
```

```
[1] "ca" "do" "mo"
```

filter and **stringr** functions

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

```
# A tibble: 4 × 5
  Diet  Gender Weight_start Weight_change Effect
  <chr> <chr>         <int>         <int> <chr>
1 A     Male            159            13 Increase
2 B     m               199            19 Increase
3 B     Other           120             0 Same
4 A     F               201            -5 Decrease
```

```
data_diet %>%
  filter(str_detect(string = Gender,
                    pattern = "M"))
```

```
# A tibble: 3 × 5
  Diet  Gender Weight_start Weight_change Effect
  <chr> <chr>         <int>         <int> <chr>
1 A     Male            159            13 Increase
2 B     M               108            12 Increase
3 B     Man             139            11 Increase
```

case_when() improved with stringr

```
count(data_diet, Gender)
```

```
# A tibble: 9 × 2
  Gender      n
  <chr>   <int>
1 f           2
2 F           2
3 Female      1
4 m           1
5 M           1
6 Male        1
7 Man         1
8 O           2
9 Other       1
```

```
data_diet %>%
  mutate(Gender = case_when(
    Gender %in% c("M", "male", "Man", "m", "Male") ~ "Male",
    Gender %in% c("F", "Female", "f", "female")~ "Female",
    Gender %in% c("O", "Other") ~ "Other"))
```
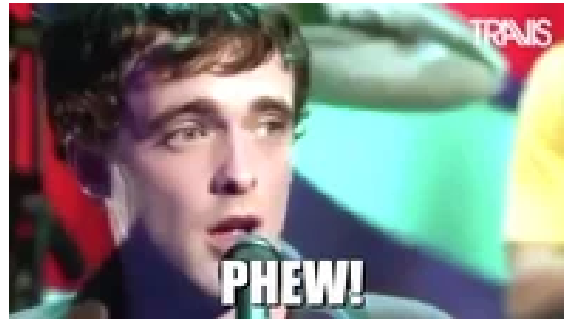
case_when() improved with stringr

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

```
data_diet %>%
  mutate(Gender = case_when(
    str_detect(string = Gender, pattern = "^m|^M") ~ "Male",
    str_detect(string = Gender, pattern = "^f|^F") ~ "Female",
    str_detect(string = Gender, pattern = "^o|^O") ~ "Other")) %>%
  count(Gender)

# A tibble: 3 × 2
  Gender     n
  <chr>  <int>
1 Female     5
2 Male       4
3 Other      3
```

That was easier!

Separating and uniting data

What if our data looked like this?

diet_comb

```
# A tibble: 4 × 2
  change          n
  <chr>       <int>
1 A_Decrease      1
2 A_Increase      3
3 B_Increase      7
4 B_Same          1
```

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

```
# A tibble: 4 × 3
  Diet  Change        n
  <chr> <chr>     <int>
1 A     Decrease      1
2 A     Increase      3
3 B     Increase      7
4 B     Same          1
```

Uniting columns

The `unite()` function can instead help combine columns.
The `col` argument specifies new column name
The `sep` argument specifies what separator to use when combining

```
# A tibble: 4 × 2
      id visit
   <int> <int>
1      1     1
2      2     1
3      3     1
4      4     1
```

```
df_united <- df %>% unite(col = "unique_id", id, visit, sep = "_")
head(df_united, 4)
```

```
# A tibble: 4 × 1
  unique_id
  <chr>
1 1_1
2 2_1
3 3_1
4 4_1
```

Combining multiple columns next to one another

```r
museums <-tibble(
     name = c("Baltimore Museum of Art", "Walters Art Museum",
              "American Visionary Art Museum"),
     street = c("10 Art Museum Dr", "600 N Charles St", "800 Key Hwy"),
     city = c("Baltimore", "Baltimore", "Baltimore"),
     State = rep("MD", 3),
     zip =  c(21218, 21201, 21230)
     )
```

Combining multiple columns next to one another

```
museums
```

```
# A tibble: 3 × 5
  name                        street           city      State   zip
  <chr>                       <chr>            <chr>     <chr> <dbl>
1 Baltimore Museum of Art     10 Art Museum Dr Baltimore MD    21218
2 Walters Art Museum          600 N Charles St Baltimore MD    21201
3 American Visionary Art Museum 800 Key Hwy    Baltimore MD    21230
```

```
museums %>% unite(col = address, street:zip, sep = ", ")
```

```
# A tibble: 3 × 2
  name                        address
  <chr>                       <chr>
1 Baltimore Museum of Art     10 Art Museum Dr, Baltimore, MD, 21218
2 Walters Art Museum          600 N Charles St, Baltimore, MD, 21201
3 American Visionary Art Museum 800 Key Hwy, Baltimore, MD, 21230
```

Summary

- `recode()` can help with simple recoding (not based on condition but simple swap)
- `case_when()` can recode **entire values** based on conditions
  - remember `case_when()` needs `TRUE ~ varaible` to keep values that aren't specified by conditions, otherwise will be `NA`
- `stringr` package has great functions for looking for specific **parts of values** especially `filter()` and `str_detect()` combined
  - also has other useful string manipulation functions like `str_replace()` and more!
  - `separate()` can split columns into additional columns
  - `unite()` can combine columns

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!

```r
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"
```

```r
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:

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

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

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

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

```r
# 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:

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

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

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

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

```r
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.

```r
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
```

Percent calculation without intermediate objects

```
red_blue  %>% pull(col_count) %>%sum()
```

```
[1] 9
```

```
sum(pull(red_blue,col_count))
```

```
[1] 9
```

```
red_blue %>% mutate(percent =
            (col_count/sum(pull(red_blue ,col_count)))*100)
```

```
# A tibble: 3 × 3
  color col_count percent
  <chr>     <int>   <dbl>
1 blue          3    33.3
2 red           3    33.3
3 <NA>          3    33.3
```

```
red_blue %>% drop_na() %>% pull(col_count) %>%sum()
```

```
[1] 6
```

```
sum(pull(drop_na(red_blue),col_count))
```

```
[1] 6
```

```
red_blue %>% mutate(percent =
            (col_count/sum(pull(drop_na(red_blue),col_count)))*100)
```