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
- Logic class objects only have `TRUE` or `False` (without quotes)
- The repeat `rep()` and `seq()` functions help you create vectors with `to` and `from` arguments (and others)
- `sample()` makes random vectors. Can be used for integers or double depending on what it is sampling from.
- 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 take a positive number (or negative number) by 0.

Finding Missing data

Each missing data type has a function that returns `TRUE` if the data is missing:

- `NA` - `is.na`
- `NaN` - `is.nan`
- `Inf` and `-Inf` - `is.infinite`

Useful checking functions

- `is.na` - is `TRUE` if the data is `FALSE` otherwise
- `!` - negation (NOT)
  - if `is.na(x)` is `TRUE`, then `!is.na(x)` is `FALSE`
- `any` will be `TRUE` if ANY are true
  - `any(is.na(x))` - do we have any `NA`'s in `x`?

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

```
[1] TRUE
```

```
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 tells you:

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

Check if rare values make sense

```
bike <-jhur::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.

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

```
[1] 85.71429
```

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()`

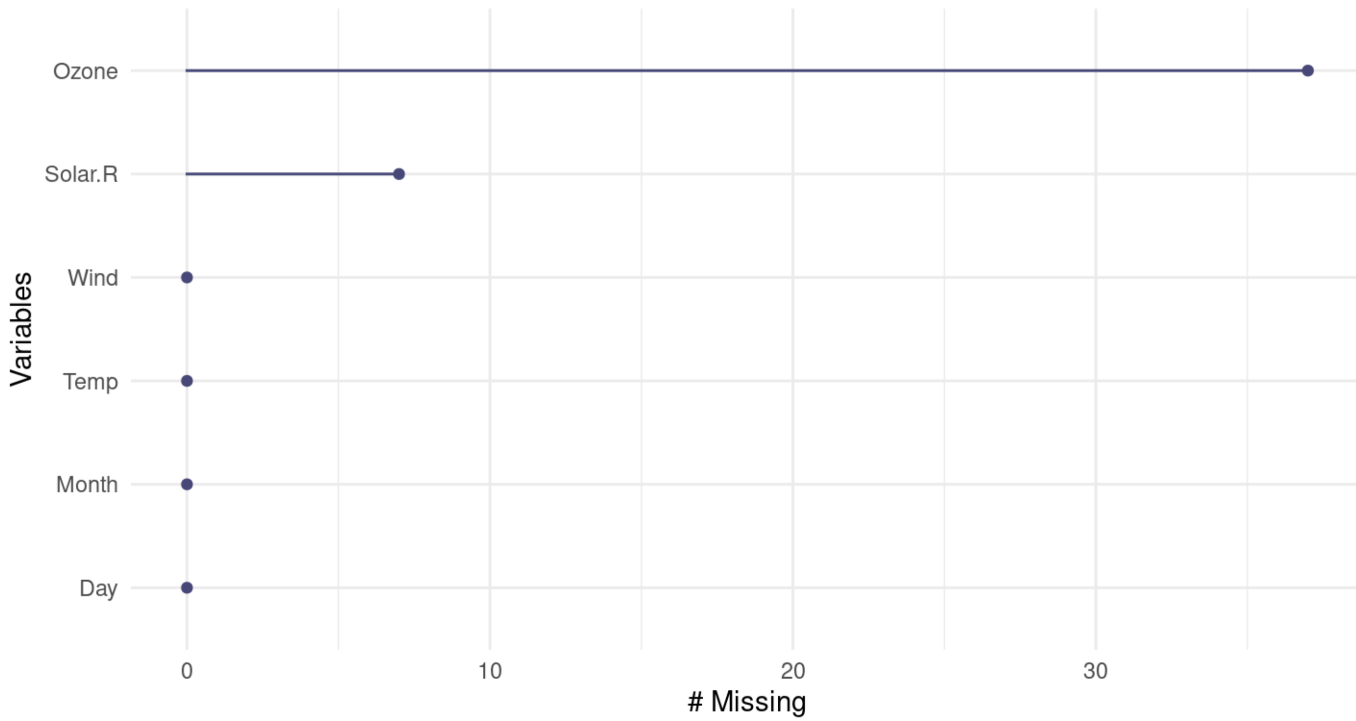
```
pct_complete(airquality)
```

```
[1] 95.20697
```

naniar plots

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

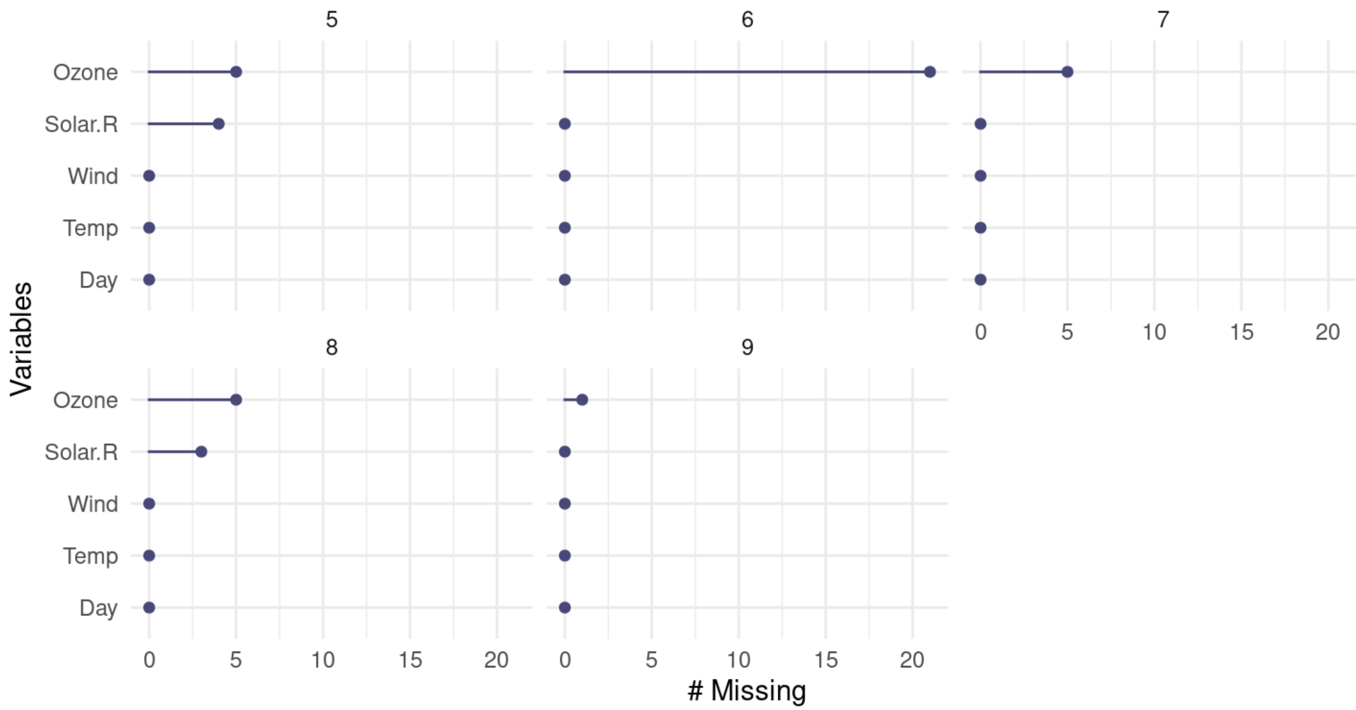
```
naniar::gg_miss_var(airqual)
```



`naniar` plots

We can use the `facet` argument to make more plots about a specific variable.

```
naniar::gg_miss_var(airqual, facet = Month)
```



Missing Data Issues

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

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

```
[1] NA
```

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

```
[1] NA
```

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

```
[1] NA
```

Missing Data Issues

This is also true for logicals. The NA data could be `> 2` or not, we don't know,so `R` says there is no `TRUE` or `FALSE`, so that is missing.

```
x = c(0, NA, 2, 3, 4, -0.5, 0.2)
x > 2
```

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

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.

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
```

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

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

to remove rows with NAs for a variable use `drop_na()`

Avoid using filter for NA values. Instead use `drop_na()`

```
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
```

tidyr::drop_na

This function will drop rows with **any** missing data in **any** column when used on a df.

```
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
```

```
drop_na(df)
```

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

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

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

Drop **columns** with any missing values

```
df<-df %>% mutate(test =c(1,2,3,4,5,6))
miss_var_which(df) # which columns have missing values
```

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

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

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

Removing columns with threshold of percent missing row values

```
is.na(df)
```

```
       Dog   Cat  test
[1,] FALSE  TRUE FALSE
[2,]  TRUE FALSE FALSE
[3,] FALSE FALSE FALSE
[4,] FALSE  TRUE FALSE
[5,] FALSE FALSE FALSE
[6,] FALSE  TRUE FALSE
```

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

```
      Dog       Cat      test 
0.1666667 0.5000000 0.0000000 
```

```
df %>% select(which(colMeans(is.na(df)) < 0.2))
```

```
# A tibble: 6 × 2
    Dog  test
  <dbl> <dbl>
1     0     1
2    NA     2
3     2     3
4     3     4
5     1     5
6     1     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 %>% select(Dog) %>% na_if(0)
```

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

```
df %>% mutate(Dog = na_if(Dog,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`

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 this and also if there was a different issue?

Think about `NA`

If it is something more like a zero then you might want it included in your data like a zero.

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.

```
red_blue
```

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

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

```
# A tibble: 3 × 3
  color col_count percent
  <chr>     <int>   <dbl>
1 blue          3   0.333
2 red           3   0.333
3 <NA>          3   0.333
```

Word of caution

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

```
# A tibble: 3 × 3
  color col_count percent
  <chr>     <int>   <dbl>
1 blue          3     0.5
2 red           3     0.5
3 <NA>          3     0.5
```

```
# Should you be dividing by 9 or 6? It depends on your data
# Pay attention to your data and your NAs!
```

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 - avoid filtering specifically for `NA` values
- `NA` values can change your calculation results
- think about what `NA` values represent

Lab Part 1

- Class Website
- Lab

Recoding Variables

Example of Recoding

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            195             9
 2 B     m               169             6
 3 B     Other           226            -5
 4 A     F               171            11
 5 B     Female          147            15
 6 B     M               159            18
 7 A     f               135            10
 8 B     O               218            14
 9 B     Man             118             2
10 A     f               236            19
11 B     F               225            -9
12 B     O               162            16
```

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, would be a matter of filtering and changing all by hand or using if statements. This can 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} = {Variable_fixing, {old_value} = {new_value},
                                       {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)
```

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}
              condition (==x, >2 ,etc.) ~ {value_for_con})
```

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

Use of **case_when()**

```
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>            195             9
 2 B     <NA>            169             6
 3 B     <NA>            226            -5
 4 A     <NA>            171            11
 5 B     <NA>            147            15
 6 B     Male            159            18
 7 A     <NA>            135            10
 8 B     <NA>            218            14
 9 B     <NA>            118             2
10 A     <NA>            236            19
11 B     <NA>            225            -9
12 B     <NA>            162            16
```

```
# General Format - this is not code!
{data_input} %>%
  mutate({variable_to_fix} = case_when({Variable_fixing}
           condition (==x, >2 ,etc.) ~ {value_for_con},
            TRUE ~ {value_for_not_meeting_condition})
```

Use of **case_when()**

```
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            195             9
 2 B     m               169             6
 3 B     Other           226            -5
 4 A     F               171            11
 5 B     Female          147            15
 6 B     Male            159            18
 7 A     f               135            10
 8 B     O               218            14
 9 B     Man             118             2
10 A     f               236            19
11 B     F               225            -9
12 B     O               162            16
```

More complicated case_when()

```
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            195             9
 2 B     Male            169             6
 3 B     Other           226            -5
 4 A     Female          171            11
 5 B     Female          147            15
 6 B     Male            159            18
 7 A     Female          135            10
 8 B     Other           218            14
 9 B     Male            118             2
10 A     Female          236            19
11 B     Female          225            -9
12 B     Other           162            16
```

Another reason for **case_when()**

`case_when` can do very sophisticated comparisons

```
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            195             9 Increase
2 B     m               169             6 Increase
3 B     Other           226            -5 Decrease
4 A     F               171            11 Increase
5 B     Female          147            15 Increase
6 B     M               159            18 Increase
```

```
# A tibble: 3 × 3
  Diet  Effect       n
  <chr> <chr>    <int>
1 A     Increase     4
2 B     Decrease     2
3 B     Increase     6
```

What if our data looked like this?

```
diet_comb
```

```
# A tibble: 3 × 2
  change          n
  <chr>       <int>
1 A_Increase      4
2 B_Decrease      2
3 B_Increase      6
```

Separating columns based on a separator

- From `tidyr`, you can split a data set into multiple columns:

```
diet_comb %>%
  separate(change, into = c("Diet", "Change"))
```

```
# A tibble: 3 × 3
  Diet  Change       n
  <chr> <chr>    <int>
1 A     Increase     4
2 B     Decrease     2
3 B     Increase     6
```

Separating columns based on a separator

You can specify the separator with `sep`.

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

```
# A tibble: 3 × 3
  Diet   Change       n
  <chr>  <chr>    <int>
1 A_diet Increase     4
2 B_diet Decrease     2
3 B_diet Increase     6
```

Uniting columns based on a separator

- From `tidyr`, you can unite:

```
df = tibble(id = rep(1:5, 3), visit = rep(1:3, each = 5))
head(df, 4)
```

```
# 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
```

Strings functions

Splitting/Find/Replace and Regular Expressions

- 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`

Download Salary FY2014 Data

From https://data.baltimorecity.gov/City-Government/Baltimore-City-Employee-Salaries-FY2015/nsfe-bg53, from https://data.baltimorecity.gov/api/views/nsfe-bg53/rows.csv

Read the CSV into R `Sal`:

```
Sal = jhur::read_salaries() # or
```

```
head(Sal)
```

```
# A tibble: 6 × 7
  name                   JobTitle   AgencyID Agency HireDate AnnualSalary GrossPay
  <chr>                  <chr>      <chr>    <chr>  <chr>    <chr>        <chr>
1 Aaron,Patricia G       Facilitie… A03031   OED-E… 10/24/1… $55314.00    $53626.…
2 Aaron,Petra L          ASSISTANT… A29045   State… 09/25/2… $74000.00    $73000.…
3 Abaineh,Yohannes T     EPIDEMIOL… A65026   HLTH-… 07/23/2… $64500.00    $64403.…
4 Abbene,Anthony M       POLICE OF… A99005   Polic… 07/24/2… $46309.00    $59620.…
5 Abbey,Emmanuel         CONTRACT … A40001   M-R I… 05/01/2… $60060.00    $54059.…
6 Abbott-Cole,Michelle CONTRACT … A90005   TRANS… 11/28/2… $42702.00    $20250.…
```

‘Find’ **`str_detect()`** function: finding values: **`stringr`**

```
Sal %>% filter(str_detect(name, "Rawlings"))
```

```
# A tibble: 3 × 7
  name                      JobTitle AgencyID Agency HireDate AnnualSalary GrossPay
  <chr>                     <chr>    <chr>    <chr>  <chr>    <chr>        <chr>
1 Rawlings,Kellye A         EMERGEN… A40302   M-R I… 01/06/2… $48940.00    $73356.…
2 Rawlings,Paula M          COMMUNI… A04015   R&P-R… 12/10/2… $19802.00    $10443.…
3 Rawlings-Blake,Stepha… MAYOR    A01001   Mayor… 12/07/1… $167449.00   $165249…
```

Showing difference in `str_replace` and `str_replace_all`

`str_replace` replaces only the first instance.

```
head(pull(Sal, JobTitle))
```

```
[1] "Facilities/Office Services II" "ASSISTANT STATE'S ATTORNEY"
[3] "EPIDEMIOLOGIST"                "POLICE OFFICER"
[5] "CONTRACT SERV SPEC II"         "CONTRACT SERV SPEC II"
```

```
head(str_replace(pull(Sal, JobTitle), "II", "2"))
```

```
[1] "Facilities/Office Services 2" "ASSISTANT STATE'S ATTORNEY"
[3] "EPIDEMIOLOGIST"               "POLICE OFFICER"
[5] "CONTRACT SERV SPEC 2"         "CONTRACT SERV SPEC 2"
```

`str_replace` replaces all instances.

```
head(str_replace_all(pull(Sal, name), "a", "j"), 2)
```

```
[1] "Ajron,Pjtricij G" "Ajron,Petrj L"
```

Summary

- `recode()` can help with simple recoding (not based on condition but simple swap)
- `case_when()` can recode 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()` or `str_extract()`

Lab Part 2

Class Website Lab

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"
```

!– # Before Cleaning - Subsetting with Brackets –>

–>

–> –> –>

Using Regular Expressions

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

```
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"
```

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`

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

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

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

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

str_extract

`str_extract` extracts matched strings - `\\d` searches for DIGITS/numbers

```
head(Sal$AgencyID)
```

```
[1] "A03031" "A29045" "A65026" "A99005" "A40001" "A90005"
```

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

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

‘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)`

‘Find’ functions: Finding Indices

These are the indices where the pattern match occurs:

```
grep("Rawlings",Sal$Name)
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
integer(0)
```

```
which(grepl("Rawlings", Sal$Name))
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
integer(0)
```

```
which(str_detect(Sal$Name, "Rawlings"))
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
integer(0)
```

‘Find’ functions: Finding Logicals

These are the indices where the pattern match occurs:

```
head(grepl("Rawlings",Sal$Name))
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
logical(0)
```

```
head(str_detect(Sal$Name, "Rawlings"))
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
logical(0)
```

‘Find’ functions: finding values, base R

```
grep("Rawlings",Sal$Name,value=TRUE)
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
character(0)
```

```
Sal[grep("Rawlings",Sal$Name),]
```

```
Warning: Unknown or uninitialised column: `Name`.
```

```
# A tibble: 0 × 7
# … with 7 variables: name <chr>, JobTitle <chr>, AgencyID <chr>, Agency <chr>,
#   HireDate <chr>, AnnualSalary <chr>, GrossPay <chr>
```

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
```

Website

[Website]

Extra slides

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

```r
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%
```