Data Management in R

Session 2

July 16 th, 2020

@EcoLaurenY

lauren@mapdatascience.com

Course website

Google Drive

What this course is:

An advanced look at tidyverse functions for data cleaning

An introduction to R Projects for Data Management

Introducing RMarkdown into the Data Management workflow for final reporting

What this course is not:

a deep dive into data visualization (ggplot2) or EDA

a course on statistics, modeling or prediction

Data Management and Cleaning in R

Posted on website and google drive

Please review the Data Management Outline and the Data Management Set up Project

Webinar slides to guide through concepts with embedded R code

Practice exercises with many data to illustrate concepts

- Go over practice exercises in webinar
- Go over it again on your own

Apply what you have learned to new data set with less guidance

At the end, structure your R Project, code and R Markdown for Reporting

R Learning Resources

RStudio

R Studio develops free and open source tools for R. RStudio's mission is to create free and open-source software for data science, scientific research, and technical communication.

Download R Studio

Cheatsheets

R for Data Science

Great free book for all things R https://r4ds.had.co.nz/

R Markdown

https://rmarkdown.rstudio.com/

https://rmarkdown.rstudio.com/lesson-1.html

R Markdown Gallery

Bad Data Handbook

https://www.oreilly.com/library/view/bad-data-handbook/9781449324957/

Data Quality

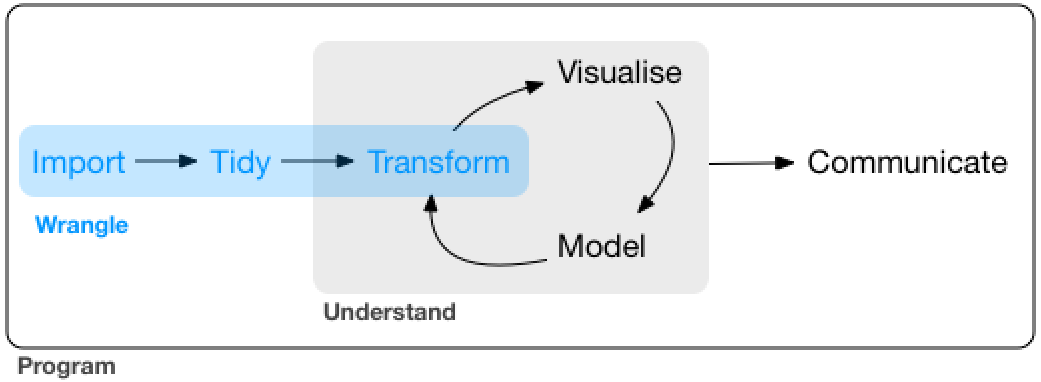
Best Practices For Scientific Computing

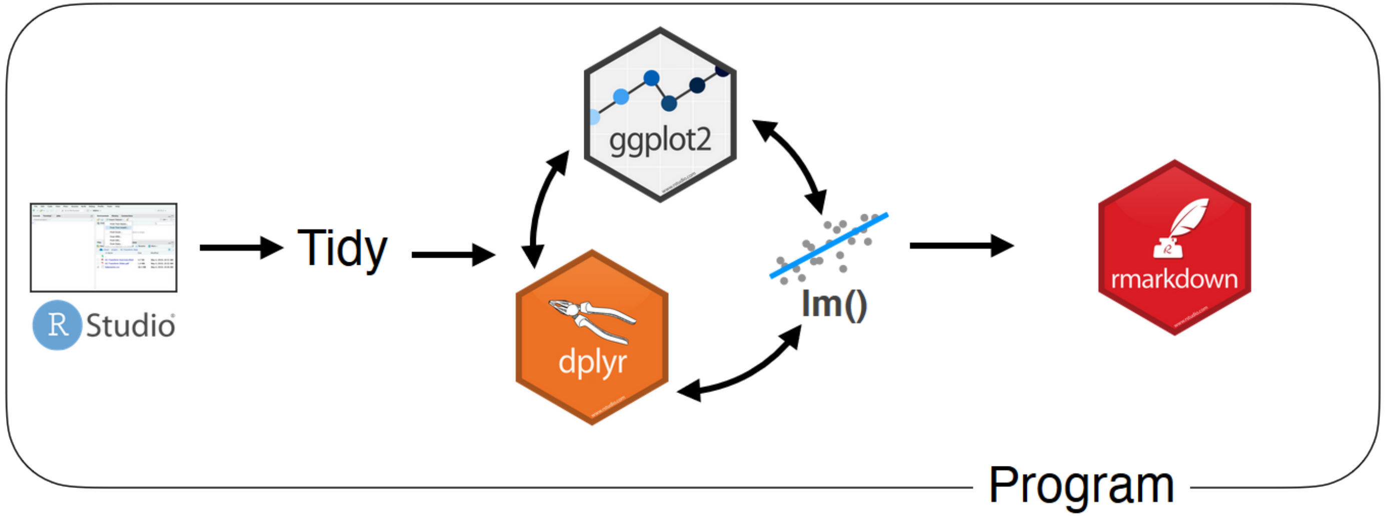
Manitoba Centre for Health Policy - Data Quality Framework for Administrative Data

Today

- Tidyverse mutate, group_by, summarize
- Joins
- Advanced Tidyverse in dplyr
- Reshaping Data

Review


Visualise
Import
Tidy
Transform
Communicate
Wrangle
Model
Understand
Program


ggplot2
Tidy
R Studio
dplyr
lm()
rmarkdown
Program

Review Reading in Data

Assigning `titanic` as the variable to hold our titanic data. `read_csv` will read in csv data and automatically determine the column types (character, numeric, factor, date, etc.)

```
titanic<-read_csv("./Raw Data/titanic.csv")
```

```
## Parsed with column specification:
## cols(
##   PassengerId = col_double(),
##   Survived = col_double(),
##   Pclass = col_double(),
##   Name = col_character(),
##   Sex = col_character(),
##   Age = col_double(),
##   SibSp = col_double(),
##   Parch = col_double(),
##   Ticket = col_character(),
##   Fare = col_double(),
##   Cabin = col_character(),
##   Embarked = col_character()
## )
```

Review Dplyr

First argument is *always* a data frame/tibble

Subsequent arguments say what to do with that data frame

Pipes %>% represent "and then..."

`count` counts the observations in a group

```
titanic %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1  1309
```

```
titanic %>%
  count(Sex)
```

```
## # A tibble: 2 x 2
##   Sex        n
##   <chr>  <int>
## 1 female   466
## 2 male     843
```

```
titanic %>%
  count(Survived)
```

```
## # A tibble: 3 x 2
##   Survived     n
##      <dbl> <int>
## 1        0   549
## 2        1   342
## 3       NA   418
```

sample_n / sample_frac for a random sample

- sample_n: randomly sample 5 observations

```
titanic_n5 <- titanic %>%
  sample_n(5, replace = FALSE)
dim(titanic_n5)
```

```
## [1]  5 12
```

- sample_frac: randomly sample 20% of observations

```
titanic_perc20 <-titanic %>%
  sample_frac(0.2, replace = FALSE)
dim(titanic_perc20)
```

```
## [1] 262  12
```

`distinct` to filter for unique rows

And `arrange` to order alphabetically

```
titanic %>%
  select(Pclass, Fare) %>%
  distinct() %>%
  arrange(Fare, Pclass)
```

```
## # A tibble: 289 x 2
##    Pclass  Fare
##     <dbl> <dbl>
##  1      1  0
##  2      2  0
##  3      3  0
##  4      3  3.17
##  5      3  4.01
##  6      1  5
##  7      3  6.24
##  8      3  6.44
##  9      3  6.45
## 10      3  6.50
## # ... with 279 more rows
```

`summarise` to reduce variables to values

```
titanic %>%
  summarise(avg_fare = mean(Fare,na.rm=T))
```

```
## # A tibble: 1 x 1
##   avg_fare
##      <dbl>
## 1     33.3
```

`group_by` to do calculations on groups

```
titanic %>%
  group_by(Sex) %>%
  summarize(avg_fare = mean(Fare,na.rm=T))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 2
##   Sex    avg_fare
##   <chr>     <dbl>
## 1 female     46.2
## 2 male       26.2
```

Select

Often we may be given a dataset with many columns and rows that we do not need for our own purposes. To reduce this data, we can take a **subset** of it. In `dplyr` there are two main functions for this purpose *selection* and *filtering*.

Selections allow you to choose the names of columns to retain or discard whereas

Filters specify some values within rows that you want to keep or discard

Filtering

Numeric Values

Filter

In most cases you may want a subset of data

You can filter numeric variables based on their values. The most used operators for this are

`>, >=, <, <=, == and !=`

For filtering between two numeric values we can use:

`filter(x>=6, x <=10)` or

`filter(x>=6 & x <=10)`

`filter(between(x,6,10))`

Another handy tool is `near` where you can specify the `tol` or tolerance between numbers :

`filter(near(x, 5, tol = 0.5))`

for instance will return any rows where x is between 5.5 and 4.5

Recall:

```
titanic %>%
  filter(Age == 35, Sex=="female")%>%
  select(Name, Sex, Age, Fare)
```

```
## # A tibble: 11 x 4
##    Name                                                 S
##    <chr>                                                <
##  1 Futrelle, Mrs. Jacques Heath (Lily May Peel)        f
##  2 Cameron, Miss. Clear Annie                          f
##  3 Harris, Mrs. Henry Birkhardt (Irene Wallach)        f
##  4 Ward, Miss. Anna                                    f
##  5 Bissette, Miss. Amelia                              f
##  6 Abbott, Mrs. Stanton (Rosa Hunt)                    f
##  7 Holverson, Mrs. Alexander Oskar (Mary Aline Towner) f
##  8 Hoyt, Mrs. Frederick Maxfield (Jane Anne Forby)     f
##  9 Geiger, Miss. Amalie                                f
## 10 Schabert, Mrs. Paul (Emma Mock)                     f
## 11 McGowan, Miss. Katherine                            f
```

Filtering

Across Multiple Columns

Filtering

- `filter_all()` will filter all columns based on your criteria
- `filter_if()` requires a function that returns a boolean to indicate which columns to filter on. If true, the filter will be applied to those columns.
- `filter_at()` requires you to specify columns inside a `vars()` argument for which the filtering will be done.

These functions have been *superseded* by `dplyr`. Which means you may see them in the documentation or older tutorials but there is a newer, better way called `across`.

Column-wise Operations

Across

`across(.cols, .fns)` **where**:

.cols are the columns you want to operate on using a tidy selection

.fns is a function or list of functions to apply to each column

Look across the columns that contain character values, and give us the number of unique values in that column

```
titanic %>%
  summarise(across(where(is.character), ~ length(unique(.x))))
```

```
## # A tibble: 1 x 5
##    Name   Sex Ticket Cabin Embarked
##   <int> <int>  <int> <int>    <int>
## 1  1307     2    929   187        4
```

Mutate

Mutate is a function that defines and inserts new variables into a tibble. You can refer to existing variables by name.

Most often when you define a new variable with mutate you'll also want to save the resulting data frame, often by writing over the original data frame.

Create a family size variable by combining SibSp and Parch (including the passenger themselves)

```
titanic <- titanic %>%
    mutate(family_size = SibSp + Parch + 1)
```

```
## # A tibble: 1,309 x 3
##    SibSp Parch family_size
##    <dbl> <dbl>       <dbl>
##  1     1     0           2
##  2     1     0           2
##  3     0     0           1
##  4     1     0           2
```

We can recode the 0 and 1 values contained in Survived to "Survived" or "Died"

```
titanic %>%
  mutate(Survived = ifelse(Survived==1, "Survived",
   ifelse(Survived==0,"Died", NA)
  ))
```

```
## # A tibble: 1,309 x 13
##    PassengerId Survived Pclass Name
##          <dbl> <chr>     <dbl> <chr>
##  1           1 Died          3 Braund, Mr. Owen Harris
##  2           2 Survived      1 Cumings, Mrs. John Bradle
##  3           3 Survived      3 Heikkinen, Miss. Laina
##  4           4 Survived      1 Futrelle, Mrs. Jacques He
##  5           5 Died          3 Allen, Mr. William Henry
##  6           6 Died          3 Moran, Mr. James
##  7           7 Died          1 McCarthy, Mr. Timothy J
##  8           8 Died          3 Palsson, Master. Gosta Le
##  9           9 Survived      3 Johnson, Mrs. Oscar W (El
## 10          10 Survived      2 Nasser, Mrs. Nicholas (Ac
## # ... with 1,299 more rows
```

Joins

There are three families of verbs designed to work with relational data:

Mutating joins, which add new variables to one data frame from matching observations in another (like `mutate()`).

Filtering joins, which filter observations from one data frame based on whether or not they match an observation in the other table (like `filter()`).

Set operations, which treat observations as if they were set elements.

Joins

We have six join options in R. Each of these join functions take at least three arguments: `x`, `y`, and `by`.

- `x` and `y` are data frames to join
- `by` is the variable(s) to join by

Four of these join functions combine variables from the two data frames:

- `inner_join()`: return all rows from `x` where there are matching values in `y`, and all columns from `x` and `y`.

- `left_join()`: return all rows from `x`, and all columns from `x` and `y`. Rows in x with no match in y will have NA values in the new columns.

- `right_join()`: return all rows from `y`, and all columns from `x` and `y`. Rows in y with no match in x will have NA values in the new columns.

- `full_join()`: return all rows and all columns from both `x` and `y`. Where there are not matching values, returns NA for the one missing.

And the other two join functions only keep cases from the left-hand data frame, and are called **filtering joins**. We'll learn about these another time but you can find out more about the join functions in the help files for any one of them, e.g. `?full_join`.

Join Example

Create a band

```
band <- tribble(
   ~name,     ~band,
  "Mick",  "Stones",
  "John", "Beatles",
  "Paul", "Beatles"
)
```

name	band
Mick	Stones
John	Beatles
Paul	Beatles

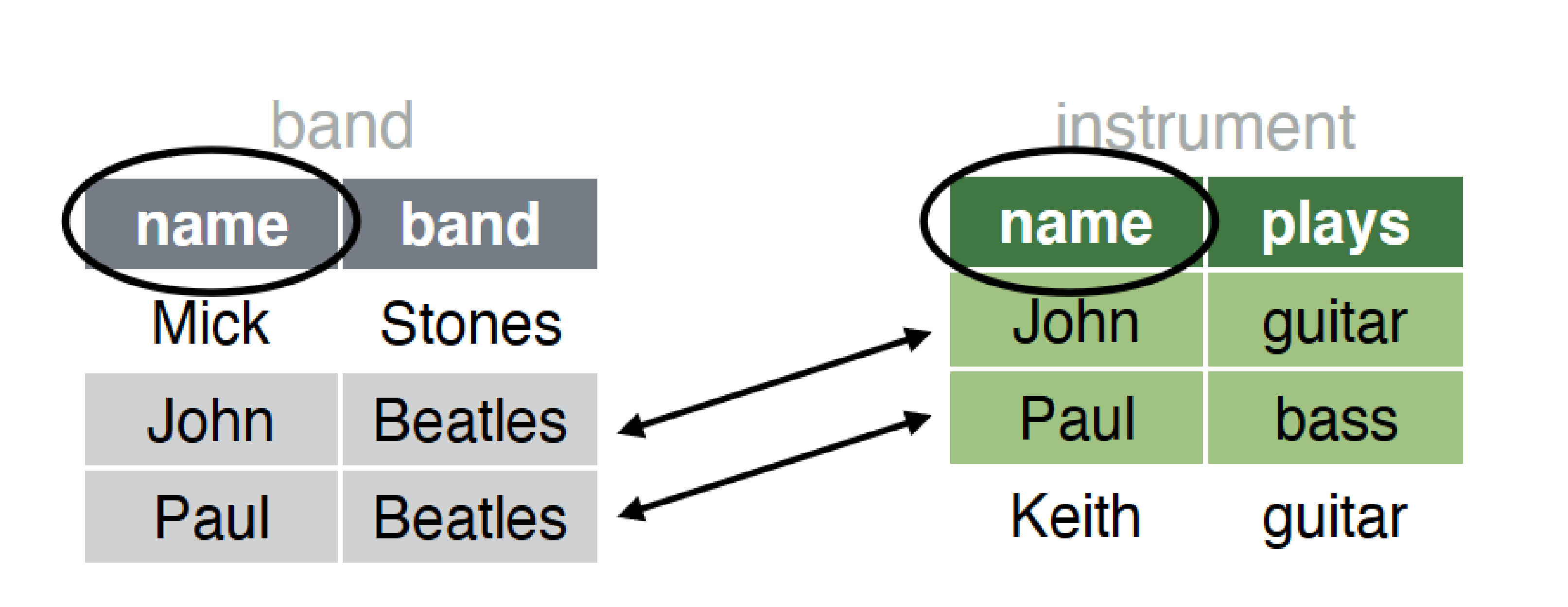
Create an instrument table

```
instrument <- tribble(
    ~name,   ~plays,
   "John", "guitar",
   "Paul",   "bass",
  "Keith", "guitar"
)
```

name	plays
John	guitar
Paul	bass
Keith	guitar

Joins

What is common between `bands` and `instruments`?



Left Join

There are a few different syntaxes for joins in R. You can use the pipe approach where you call one `tribble` and then `pipe` into a join.

```
band %>% left_join(instrument, by = "name")

left_join(band,instrument,by="name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass

Right Join

```
band %>% right_join(instrument, by = "name")

right_join(band,instrument,by="name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

Full Join

```
band %>% full_join(instrument, by = "name")

full_join(band,instrument,by="name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

Review types of Joins

```
band %>% left_join(instrument, by = "name")
band %>% right_join(instrument, by = "name")
band %>% full_join(instrument, by = "name")
band %>% inner_join(instrument, by = "name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
## 3 Keith <NA>    guitar
```

```
## # A tibble: 4 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
```

Filtering Joins

Semi Join returns those in band that have a match in `instrument`

```
band %>% semi_join(instrument, by = "name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band
John	Beatles
Paul	Beatles

Anti Join returns the rows in band that do not have a match `instruments`

```
band %>% anti_join(instrument, by = "name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band
Mick	Stones

Joins continued

Let's use a similar example but here we define a new variable `instrument2`.

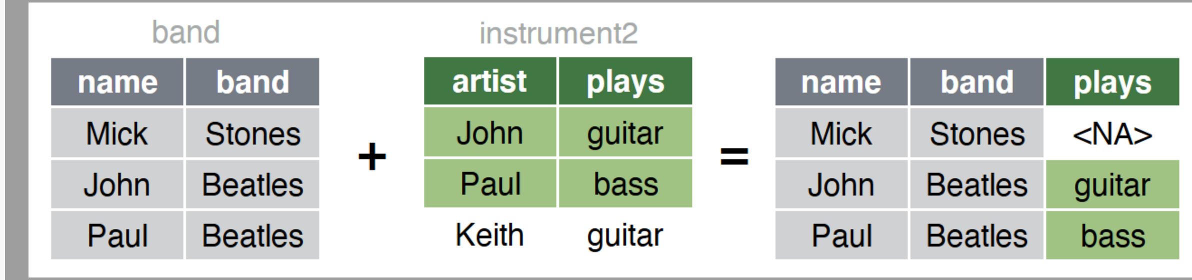
What if the variable names do not match?

```r
band <- tribble(
   ~name,     ~band,
  "Mick",  "Stones",
  "John", "Beatles",
  "Paul", "Beatles"
)

instrument2 <- tribble(
    ~artist,   ~plays,
   "John", "guitar",
   "Paul",   "bass",
  "Keith", "guitar"
)
```

We can use the following syntax to match `name` in `band` with the key `artist` in `instrument2`

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```



Join Problems

In the real world, joins can be a real pain when they go wrong…

Here are a few things that you should do with your own data to make your joins go smoothly.

Identifying the variables that form the primary key in each table. You should usually do this based on your understanding of the data, not empirically by looking for a combination of variables that give a unique identifier. If you just look for variables without thinking about what they mean, you might get (un)lucky and find a combination that's unique in your current data but the relationship might not be true in general.

Check that none of the variables in the primary key are missing. If a value is missing then it can't identify an observation!

Check that your foreign keys match primary keys in another table. The best way to do this is with an `anti_join()`. It's common for keys not to match because of data entry errors. Fixing these is often a lot of work.

If you do have missing keys, you'll need to be thoughtful about your use of inner vs. outer joins, carefully considering whether or not you want to drop rows that don't have a match.

⚠️Be aware that simply checking the number of rows before and after the join is not sufficient to ensure that your join has gone smoothly. If you have an inner join with duplicate keys in both tables, you might get unlucky as the number of dropped rows might exactly equal the number of duplicated rows!

Take aways

- `left_join()` retains all cases in *left* data set
- `right_join()` retains all cases in *right* data set
- `full_join()` retains all cases in *either* data set
- `inner_join()` retains *only* cases in *both* data sets

- `semi_join()` extracts cases that *have* a match
- `anti_join()` extracts cases that *do not have* a match

Tidyr

With the data in tidy form, it's natural to get a computer to do further summarization or to make a figure.

Untidy Data may lead to issues with plotting and visualizations - check that your data is tidy!

See: https://tidyr.tidyverse.org/

Reshaping Data with Tidyr

pivot_longer

"lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is pivot_wider()

formerly known as `spread`

Consider an example data set `religion and income`

```
## # A tibble: 18 x 11
##    religion                `<$10k` `$10-20k` `$20-30k` `
##    <chr>                     <dbl>     <dbl>     <dbl>
##  1 Agnostic                     27        34        60
##  2 Atheist                      12        27        37
##  3 Buddhist                     27        21        30
##  4 Catholic                    418       617       732
##  5 Don’t know/refused           15        14        15
##  6 Evangelical Prot            575       869      1064
##  7 Hindu                         1         9         7
##  8 Historically Black Prot     228       244       236
##  9 Jehovah's Witness            20        27        24
## 10 Jewish                       19        19        25
## 11 Mainline Prot               289       495       619
```

```
relig_income %>%
  pivot_longer(-religion, names_to = "income", values_to =
```

```
## # A tibble: 180 x 3
##    religion income             count
##    <chr>    <chr>              <dbl>
##  1 Agnostic <$10k                 27
##  2 Agnostic $10-20k               34
##  3 Agnostic $20-30k               60
##  4 Agnostic $30-40k               81
##  5 Agnostic $40-50k               76
##  6 Agnostic $50-75k              137
##  7 Agnostic $75-100k             122
##  8 Agnostic $100-150k            109
##  9 Agnostic >150k                 84
## 10 Agnostic Don't know/refused    96
## # ... with 170 more rows
```

Reshaping Data with Tidyr

pivot_wider

"widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is `pivot_longer()`.

formerly known as `gather`

```
## # A tibble: 114 x 3
##    fish  station  seen
##    <fct> <fct>   <int>
##  1 4842  Release     1
##  2 4842  I80_1       1
##  3 4842  Lisbon      1
##  4 4842  Rstr        1
##  5 4842  Base_TD     1
##  6 4842  BCE         1
##  7 4842  BCW         1
##  8 4842  BCE2        1
##  9 4842  BCW2        1
## 10 4842  MAE         1
## # ... with 104 more rows
```

```
fish_encounters %>%
  pivot_wider(names_from = station, values_from = seen)
```

```
## # A tibble: 19 x 12
##    fish  Release I80_1 Lisbon  Rstr Base_TD   BCE   BCW <
##    <fct>   <int> <int>  <int> <int>   <int> <int> <int> <
##  1 4842        1     1      1     1       1     1     1
##  2 4843        1     1      1     1       1     1     1
##  3 4844        1     1      1     1       1     1     1
##  4 4845        1     1      1     1       1    NA    NA
##  5 4847        1     1      1    NA      NA    NA    NA
##  6 4848        1     1      1     1      NA    NA    NA
##  7 4849        1     1     NA    NA      NA    NA    NA
##  8 4850        1     1     NA     1       1     1     1
##  9 4851        1     1     NA    NA      NA    NA    NA
## 10 4854        1     1     NA    NA      NA    NA    NA
## 11 4855        1     1      1     1       1    NA    NA
## 12 4857        1     1      1     1       1     1     1
## 13 4858        1     1      1     1       1     1     1
## 14 4859        1     1      1     1       1    NA    NA
## 15 4861        1     1      1     1       1     1     1
## 16 4862        1     1      1     1       1     1     1
## 17 4863        1     1     NA    NA      NA    NA    NA
## 18 4864        1     1     NA    NA      NA    NA    NA
## 19 4865        1     1      1    NA      NA    NA    NA
```

Data Management in R

Session 2

July 16 th, 2020

@EcoLaurenY

lauren@mapdatascience.com

Course website

Google Drive

Review: Week_1_Practice_01.Rmd and Week_1_Practice_02.rmd

References

Portions of this material are derived from:

RStudio's 'Learning Tidyverse'

Data Carpentry datasciencebox.org

Estrellado, R. A., Bovee, E. A., Motsipak, J., Rosenberg, J. M., & Velásquez, I. C. (in press). Data science in education using R. London, England: Routledge. Nb. All authors contributed equally

https://stat545.com/

https://r4ds.had.co.nz/