Data Summarization and Transformation - 1

One should look for what is and not what he thinks should be. (Albert Einstein)

Data Summarization and Transformation: Topic introduction

In this part of the course, we will cover the following concepts:

- Tidy data best practices
- Transform data with tidyr

Warm-up: dplyr review

- Using the six dplyr verbs, how would you transform the following table so that:
  - column A only holds values under 15,
  - column B is in alphabetical order,
  - column C is removed from the dataset,
  - and there's a new column called 'D' which holds the mean of all of the values in column A

A	B	C
5	Adam	Eggs
10	Chelsea	Cheese
15	Bernice	Olives
20	Freddy	Ham
25	Diana	Spinach

Module completion checklist

Topic	Complete
Summarize columns using the summarise and group by functions	
Understand tidy data and its advantages	

Remaining dplyr functions

- With a bit of dplyr experience, we can do a substantial amount of data wrangling
- We can use filter to subset data, arrange to order observations, select to choose specific columns, and mutate to create new variables
- That leaves two functions left to unpack, among the most commonly used in dplyr:
  - `summarise()`, to create a new summary data frame
  - `group_by()`, to take an existing table and convert it to a grouped table so operations can be performed by group

Summarise and group_by

- `summarise()` collapses a data frame down to a single row
- By itself, `summarise` is not very helpful
- We will often use it with `group_by()`
- **Note:** You can also use the synonym `summarize()` (spelled with a "z")

```
# Check for detailed documentation
?dplyr::summarise

# Use cases for `summarise` function
summarise(df,                  #<- data frame
          summary_function1,   #<- summary rule(s)
                               #   for new column
          ...)
```

summarise {dplyr} R Documentation

Summarise each group to fewer rows

Description

`summarise()` creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarising all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

`summarise()` and `summarize()` are synonyms.

Usage

```
summarise(.data, ..., .groups = NULL)

summarize(.data, ..., .groups = NULL)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.

Summarise and group_by

- Grouping doesn't change how data looks apart from listing how it's grouped
- It will change how it acts with the other dplyr verbs
- To remove grouping, use `ungroup()`

```
# Check for detailed documentation
?dplyr::group_by

# Use cases for `group_by` function
group_by(df,          #<- data frame
         variable1, #<- 1st variable to group by
         variable2, #<- 2nd variable to group by
         ...)
```

group_by {dplyr} R Documentation

Group by one or more variables

Description

Most data operations are done on groups defined by variables. `group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

Usage

```
group_by(.data, ..., .add = FALSE, .drop = group_by_drop_default(.data))

ungroup(x, ...)
```

Arguments

`.data`	A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details.
`...`	In `group_by()`, variables or computations to group by. In `ungroup()`, variables to remove from the grouping.
`.add`	When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`. This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions.
`.drop`	When `.drop = TRUE`, empty groups are dropped. See `group_by_drop_default()` for what the default value is for this argument.
`x`	A `tbl()`

Summarise and group_by alone

- Let's see what these functions do independently of one another by looking at the `nycflights13` dataset

```
# Produce a summary
summarise(flights, delay =
mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

```
# Create `by_day` by grouping `flights` by year, month, and day.
by_day = group_by(flights, year, month, day)
by_day
```

```
# A tibble: 336,776 x 19
# Groups:   year, month, day [365]
    year month   day dep_time sched_dep_time dep_delay
arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>
<int>          <int>
 1  2013     1     1      517            515         2
830            819
 2  2013     1     1      533            529         4
850            830
 3  2013     1     1      542            540         2
923            850
 4  2013     1     1      544            545        -1
1004          1022
 5  2013     1     1      554            600        -6
812            837
 6  2013     1     1      554            558        -4
740            728
 7  2013     1     1      555            600        -5
```

Summarise and group_by together

- On their own, `summarise()` and `group_by()` have some usefulness
- But by using `summarise()` and its arguments on grouped data, you can generate new insights

```
# Now use grouped `by_day` data and summarise it to see the average delay by year, month and day.
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# … with 355 more rows
```

Dplyr and the pipe

- Suppose we wanted to understand how flight distance affects delays
- We could chain together the outputs of a series of dplyr functions:
  - **Group** flights by destination
  - **Summarize** to compute distance, average delay, and number of flights
  - **Filter** to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport

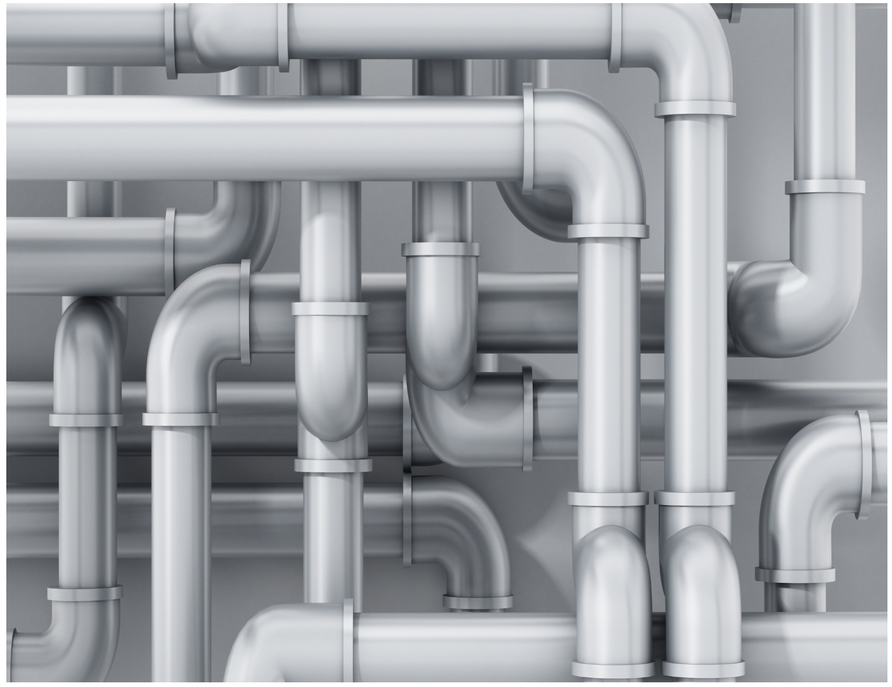
```
# Step 1: Create a new grouped data
frame `by_dest`.
by_dest = group_by(flights, dest)

# Step 2: Create a summary of `by_dest` and save as `delay`.
delay = summarise(by_dest,
                  count = n(),
                  dist = mean(distance, na.rm = TRUE),
                  delay = mean(arr_delay, na.rm = TRUE))

# Step 3: Filter `delay` by their count and destination.
delay = filter(delay, count > 20, dest != "HNL")
```

Introducing the pipe operator

- We can use the **pipe operator** (%>%)
- Pipes take the **output** of one function and feed it in as **input** to another function without the need for intermediate steps
- Read the %>% as **“and then”** when you see it in code
- Pipes also make your code more readable and easier to debug



Utilizing the pipe operator

- Here's the same example on the previous slide, this time incorporating pipes!

```
delays = flights %>%                                        #<- take flights data
  group_by(dest) %>%                                        #<- group it by destination
  summarise(count = n(),                                    #<- then summarize by creating count
variable
            dist = mean(distance, na.rm = TRUE),            #<- and computing mean distance
            delay = mean(arr_delay, na.rm = TRUE)) %>%      #<- and mean arrival delay
  filter(count > 20, dest != "HNL")                         #<- then filter it
```

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

```
delays
```

```
# A tibble: 96 x 4
   dest  count  dist delay
   <chr> <int> <dbl> <dbl>
 1 ABQ     254 1826   4.38
 2 ACK     265  199   4.85
 3 ALB     439  143  14.4
 4 ATL   17215  757. 11.3
 5 AUS    2439 1514.  6.02
 6 AVL     275  584.  8.00
 7 BDL     443  116   7.05
 8 BGR     375  378   8.03
 9 BHM     297  866. 16.9
10 BNA    6333  758. 11.8
# … with 86 more rows
```

Summarise and handling NAs

- We do NOT address NAs

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# … with 355 more rows
```

- We address NAs

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay,
                        na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# … with 355 more rows
```

- If we do not address NAs, the aggregation functions will return NAs for each item if there is just one NA in the input
- Be intentional about how you address NAs when summarizing

A few more useful summary functions

- Apart from `mean()`, there are many other summary functions describing data from various aspects:

Summary Functions	Explanation
`n()`	will count the number of entries that come from a summarise
`min(x), quantile(x, 0.25), max(x)`	measures of rank and distribution can be used
`first(x), nth(x, 2), last(x)`	measures of position and order
`n_distinct`	will count the number of distinct values

Summarise with n()

- `n()` will count the number of entries that come from a `summarise()` function
- **Note:** `n()` can **only** be used within `summarise()`, `mutate()` and `filter()`

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE),
            n = n()) #<- add a column with summary counts
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day  mean     n
   <int> <int> <int> <dbl> <int>
 1  2013     1     1 11.5    842
 2  2013     1     2 13.9    943
 3  2013     1     3 11.0    914
 4  2013     1     4  8.95   915
 5  2013     1     5  5.73   720
 6  2013     1     6  7.15   832
 7  2013     1     7  5.42   933
 8  2013     1     8  2.55   899
 9  2013     1     9  2.28   902
10  2013     1    10  2.84   932
# … with 355 more rows
```

Count instances without summarise()

- `count()` is a simple count function that **does not** require the summary function
- `count()` is the rough equivalent of `summarise(n = n())`

```
flights %>%
  count(day) #<- count number of instances of entry in `day` column
```

```
# A tibble: 31 x 2
     day     n
   <int> <int>
 1     1 11036
 2     2 10808
 3     3 11211
 4     4 11059
 5     5 10858
 6     6 11059
 7     7 10985
 8     8 11271
 9     9 10857
10    10 11227
# … with 21 more rows
```

Summarise using min and max

- Suppose you were to summarize the data to obtain departure information on the flights that were not canceled on a particular day
- You can use `summarise()` to return the `min(x)`, `quantile(x, 0.25)`, and `max(x)` from the grouped data to display the departure times of the first and last flights

```
# 1. Build a subset of all flights that were not canceled.
not_cancelled = flights %>%
  filter(!is.na(dep_time))  #<- filter flights where `dep_time` was not `NA`

# 2. Group and summarize all flights that were not canceled to get desired results.
not_cancelled  %>%
  group_by(year, month, day) %>%   #<- group the not canceled flights
  summarise(first = min(dep_time), #<- then summarize them by calculating the first
            last = max(dep_time))  #<- and last flights in the `dep_time` in each group
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day first  last
   <int> <int> <int> <int> <int>
 1  2013     1     1   517  2356
 2  2013     1     2    42  2354
 3  2013     1     3    32  2349
 4  2013     1     4    25  2358
 5  2013     1     5    14  2357
 6  2013     1     6    16  2355
 7  2013     1     7    49  2359
```

Summarise using first and last

- You could also choose to obtain the departure information of the flights using the `first()` and `last()` functions

```
# 1. Group and summarize all flights that were not canceled to get desired results using first and
last functions.
not_cancelled  %>%
  group_by(year, month, day) %>%   #<- group the not canceled flights
  summarise(first = first(dep_time), #<- then summarize them by calculating the first
            last = last(dep_time))  #<- and last flights in the `dep_time` in each group
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day first  last
   <int> <int> <int> <int> <int>
 1  2013     1     1   517  2356
 2  2013     1     2    42  2354
 3  2013     1     3    32  2349
 4  2013     1     4    25  2358
 5  2013     1     5    14  2357
 6  2013     1     6    16  2355
 7  2013     1     7    49  2359
 8  2013     1     8   454  2351
 9  2013     1     9     2  2252
10  2013     1    10     3  2320
# … with 355 more rows
```

Summarise distinct values

- `n_distinct(x)` will count the number of distinct values

```
# Number of flights that take off, by day.
not_cancelled  %>%
  group_by(year, month, day) %>%
  summarise(flights_that_take_off = n_distinct(dep_time)) #<- calculate distinct departure times
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day flights_that_take_off
   <int> <int> <int>                 <int>
 1  2013     1     1                   552
 2  2013     1     2                   583
 3  2013     1     3                   589
 4  2013     1     4                   589
 5  2013     1     5                   495
 6  2013     1     6                   564
 7  2013     1     7                   572
 8  2013     1     8                   573
 9  2013     1     9                   580
10  2013     1    10                   572
# … with 355 more rows
```

Remember to ungroup before you regroup

- `group_by()` adds metadata to the dataframe that impacts its usability downstream
- Because of this, it is recommended that you use `ungroup()` after every `group_by()`

```
# Take the same `not_canceled` data, but now group by month instead of by day.
not_cancelled %>%                                      #<- set data frame
  ungroup() %>%                                        #<- first ungroup it
  group_by(year, month) %>%                            #<- then group by year and month
  summarise(flights_by_year = n_distinct(dep_time))#<- then do the rest ...
```

```
# A tibble: 12 x 3
# Groups:   year [1]
    year month flights_by_year
   <int> <int>           <int>
 1  2013     1            1165
 2  2013     2            1171
 3  2013     3            1199
 4  2013     4            1216
 5  2013     5            1186
 6  2013     6            1220
 7  2013     7            1242
 8  2013     8            1204
 9  2013     9            1156
10  2013    10            1139
11  2013    11            1135
12  2013    12            1191
```

Module completion checklist

Topic	Complete
Summarize columns using the summarise and group by functions	✔
Understand tidy data and its advantages	

Introducing tidyr

- We've discussed several kinds of activities that fall under the category of data wrangling, including profiling, restructuring, and cleaning
- **tidyr**, a package within tidyverse, allows us to get our data into a **tidy** format
- When data is tidy, it is easier to merge with other data sources, to validate for usability and correctness, and to share with others
- For further reading and understanding of tidy data and where it originated, check out this paper, ***Tidy Data (opens PDF)***, by Hadley Wickham, Chief Scientist at Posit, PBC

Ease of analysis with tidy datasets

- **tidyr** has some in-built datasets derived from the World Health Organization's tuberculosis data
- Each of the following tables relates to the number of cases and the total population for each country for the years 1999 and 2000

- How would you describe the organization of this table?
- Is it easy or difficult to work with?

```
table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

Ease of analysis with tidy datasets contd

- What about this table?

```
table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

- And this one?

```
table3
```

```
# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

Characteristics of ‘tidy’ data

- These three interrelated rules make a dataset tidy:
  - Each **variable** must have its own **column**
  - Each **observation** must have its own **row**
  - Each **value** must have its own **cell**
- `tidy_country`, the example below, follows all three rules.

```
table1
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

- Do you think the examples on the previous slide follow these rules?

Advantages of tidy data

- Storing data in a **consistent** way:
  - It's easier to learn the tools that work with it because of the underlying uniformity
- Making use of R's **internal vectorization**:
  - Most built-in R functions work with vectors of values
- Making use of **`pivot_longer`** and **`pivot_wider`**:
  - The functions of `tidyr` that help transform `messy` data to `tidy` data
  - We'll talk more about these functions in the next module!

Knowledge check

Module completion checklist

Topic	Complete
Summarize columns using the summarise and group by functions	✔
Understand tidy data and its advantages	✔

Congratulations on completing this module!

You are now ready to try Tasks 1-3 in the Exercise for this topic