Data Transformations & Manipulation with dplyr

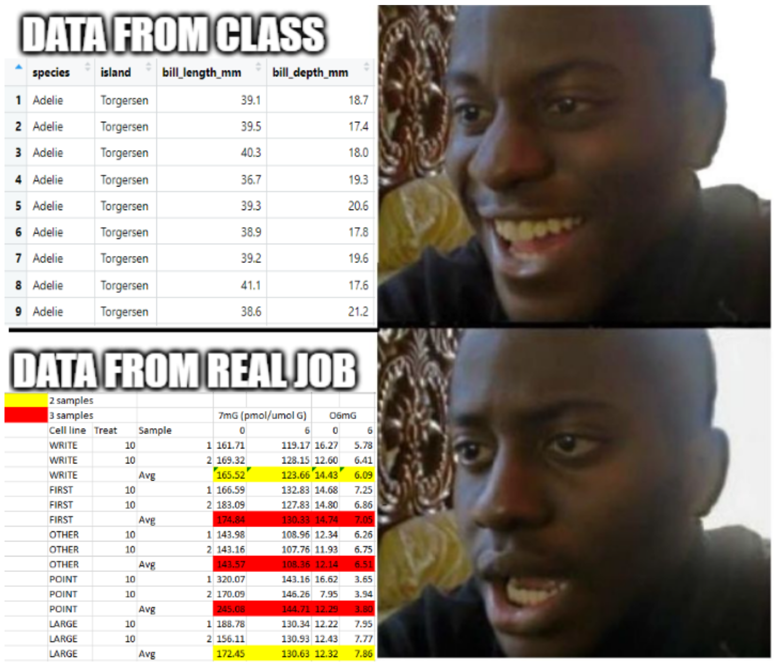
STA 418/518 - Statistical Computing and Graphics with R

Andrew DiLernia

Complete the following activity using R Markdown and then submit your .Rmd file and output Word, PDF, or HTML document via Blackboard.

It varies by domain area, the particular project, and the data scientist conducting an analysis, but it is estimated that about 50-80% of the time to conduct an analysis is spent preparing data to be analyzed as opposed to visualizing, modeling, and actually analyzing the data[1].

Most data we encounter "in the wild" will not be precisely the right form we need for analyzing. Often data will need to be cleaned or tidied up by adding or removing rows & columns, modifying existing columns, creating new columns in our data set, or combining data from multiple sources.



	species	island	bill_length_mm	bill_depth_mm
1	Adelie	Torgersen	39.1	18.7
2	Adelie	Torgersen	39.5	17.4
3	Adelie	Torgersen	40.3	18.0
4	Adelie	Torgersen	36.7	19.3
5	Adelie	Torgersen	39.3	20.6
6	Adelie	Torgersen	38.9	17.8
7	Adelie	Torgersen	39.2	19.6
8	Adelie	Torgersen	41.1	17.6
9	Adelie	Torgersen	38.6	21.2

🟨	2 samples						
🟥	3 samples			7mG (pmol/umol G)		O6mG	
	Cell line	Treat	Sample	0	6	0	6
	WRITE	10	1	161.71	119.17	16.27	5.78
	WRITE	10	2	169.32	128.15	12.60	6.41
	WRITE		Avg	165.52	123.66	14.43	6.09
	FIRST	10	1	166.59	132.83	14.68	7.25
	FIRST	10	2	183.09	127.83	14.80	6.86
	FIRST		Avg	174.84	130.33	14.74	7.05
	OTHER	10	1	143.98	108.96	12.34	6.26
	OTHER	10	2	143.16	107.76	11.93	6.75
	OTHER		Avg	143.57	108.36	12.14	6.51
	POINT	10	1	320.07	143.16	16.62	3.65
	POINT	10	2	170.09	146.26	7.95	3.94
	POINT		Avg	245.08	144.71	12.29	3.80
	LARGE	10	1	188.78	130.34	12.22	7.95
	LARGE	10	2	156.11	130.93	12.43	7.77
	LARGE		Avg	172.45	130.63	12.32	7.86

1

https://diginomica.com/data-science-myths-and-realities-do-data-scientists-really-spend-80-their-time-wrangling-data

In this activity we will explore some key concepts of data cleaning and data wrangling with R using the `dplyr` package.

Learning Objectives

1. Subsetting or selecting rows / observations with `filter()`
2. Sorting or arranging rows of a data set with `arrange()`
3. Rearranging and dropping columns / variables with `select()`
4. Modifying existing columns and creating new columns with `mutate()`
5. Data operations and summary statistics by sub-groups with `group_by()` and `summarize()`

The `dplyr` package

The `dplyr` package, a part of the `tidyverse` set of packages, provides many tools for transforming and manipulating our data.

- **Transforming**: creating new variables, summary columns, modifying existing variables
- **Manipulating**: renaming variables, reordering rows or columns, subsetting by rows or columns, etc.

Data visualization is important, but requires clean data or data in a certain format. Using `dplyr` we can prepare our data for visualization, modeling, and more!

Example data: Michigan flights

For this activity, we will be using the `tidyverse` package and a data set on all flights departing from the four largest Michigan airports between 2019 and 2021: Detroit (DTW), Lansing (LAN), Grand Rapids (GRR), and Flint (FNT) obtained via the `anyflights` R package. This package facilitates obtaining air travel and weather data for airports across the United States.

Image obtained from airport-technology.com.

Variable	Description
origin	Weather station
hour	Hour of recording, UTC
temp	Temperature (Fahrenheit)
dewp	Dewpoint (Fahrenheit)
humid	Relative humidity
wind_dir	Wind direction (degrees)
wind_speed	Wind speed (miles per hour)
wind_gust	Wind gust speed (miles per hour)
precip	Precipitation (inches)
pressure	Sea level pressure (millibars)
visib	Visibility (miles)
time_hour	Date and hour of the recording as a POSIXct date, UTC

Variable	Description
air_time	Amount of time spent in the air (minutes)
distance	Distance between airports (miles)
dep_delay	Departure delay (minutes); negative times represent early departure
arr_delay	Arrival delay (minutes); negative times represent early arrival

➡️ Load packages necessary for this activity using the code below. Note: you may need to install packages beforehand by using the `install.packages()` function and the package name in quotes.

```
library(tidyverse)
library(lubridate)
library(knitr)
library(skimr)
```

➡️ Download the CSV file *miFlights2019-2021.csv* from Blackboard, and import it into R creating an object called `miFlights`.

➡️ Use the `skim()` and `glimpse()` functions to explore characteristics of the data set.

➡️ How many variables and how many observations are in this data set?

➡️ Are there any notable patterns of missing values?

➡️ Create a bar chart showing how many flights departed out of each airport (`origin`) using the `count()` and `geom_col()` functions. Also sort the bars by descending height using the `fct_reorder()` function.

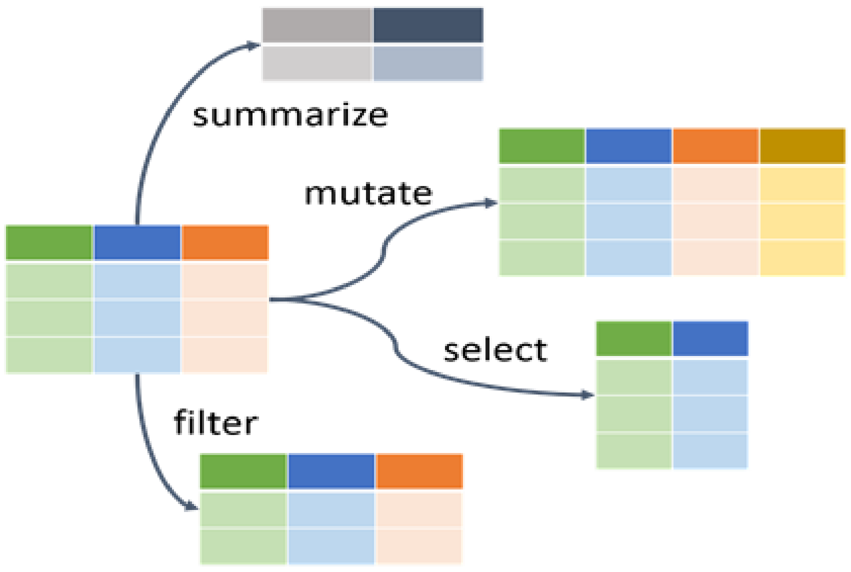
Variable Types

Some variable types may seem new, while others we have encountered before.

- **`int`** stands for integers.
- **`dbl`** stands for doubles, or real numbers.
- **`chr`** stands for character vectors, or strings.
- **`dttm`** stands for date-times (a date + a time).
- **`lgl`** stands for logical, vectors that contain only TRUE or FALSE.

- `fctr` stands for factors, which R uses to represent categorical variables with fixed possible values.

Fundamental `dplyr` functions



- **`filter()`**: pick observations by their values
- **`arrange()`**: sort / reorder rows
- **`select()`**: pick variables by their names
- **`mutate()`**: create new variables using functions of existing variables
- **`summarize()`**: collapse many values to a single summary in conjunction with `group_by()`
- **`group_by()`**: specifies the scope of each function from operating on the entire data set to operating on it group-by-group

Most functions from the `dplyr` package have a consistent structure:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The output / result is a new data frame (allows us to chain together `dplyr` functions using the `%>%`, called a "pipe", operator)

Filtering rows with `filter()`

- Using `filter()`, we can subset observations based on their values.
- The first argument of `filter()` is the data frame object, and subsequent arguments separated by commas are the conditions to filter by

➡️ Selecting all flights on January 1st in the data set, create a new object called `janFlights`.

➡️ Suppose we want to create a data set called `dec25` that contains flight data from December 25th. What code would we need using the `filter()` function to create `dec25`?

In general, the double equals sign is used to check equality ==, while a single equals sign = is used for specifying arguments of functions, but not for checking equality.

Logical comparisons within `filter()`

A key aspect of using the `filter()` function is to create a logical / boolean (`TRUE` or `FALSE` value) vector indicating whether or not each row should be retained in the data set. Here are some useful tools for constructing logical expressions:

1. | serves as an 'or'
2. & serves as an 'and'
3. == is used to check equality
4. > is used to check if a numeric variable is *greater than* a certain value
5. < is used to check if a numeric variable is *less than* a certain value
6. >= is used to check if a numeric variable is *greater than or equal to* a certain value
7. <= is used to check if a numeric variable is *less than or equal to* a certain value

➡️ Find all flights that departed in November or December, creating an object called `novDec`.

If there are more than just two categories (November and December here), this code could become very long and cumbersome. The `%in%` operator makes this easier.

➡️ Find all flights that departed in November or December using the `%in%` operator, creating an object called `novDec`.

We can also use negation to filter by certain rows using `!`. When doing so, it is important to be mindful of De Morgan's laws. For two events, say A and B, De Morgan's laws state:

$$!\,(A \mid B) \;= (!\,A)\ \&\ (!\,B),\ !\,(A\ \&\ B) \;= (!\,A) \mid (!\,B),$$

where:

- ! A is the negation of A,
- & is the intersection operator (AND),
- | is the union operator (OR).

➡️ Select all flights except those in the months of November and December using `!`.

➡️ Knowing that `arr_delay` and `dep_delay` represent the arrival and departure delays in minutes respectively, what data set is produced using the code below?

```
dplyr::filter(miFlights, !(arr_delay > 120 | dep_delay > 120))
dplyr::filter(miFlights, arr_delay <= 120, dep_delay <= 120)
```

Missing values

Missing values in R are represented using `NA`, and are an important consideration when using `filter()`.

➡️ What does running the code below produce?

```
NA > 5
10 == NA
NA + 10
NA / 2
```

➡️ How about the following code:

```
NA == NA
```

Hadley gave a helpful line of reasoning for why this does not work as expected

```
# Let x be Mary's age. We don't know how old she is.
x <- NA

# Let y be John's age. We don't know how old he is.
y <- NA

# Are John and Mary the same age?
x == y
# We don't know!
```

If you want to determine if a value is missing, use the `is.na()` function:

```
x <- NA
is.na(x)
```

Using `is.na()` in tandem with `filter()` we can subset to only missing values, or where there are no missing values.

➡️ Create a new object called `miFlightsComplete` where all departure times are non-missing, and `miFlightsMiss` where all departure times are missing

Arrange rows with `arrange()`

The `arrange()` function works similarly to `filter()` except that instead of selecting rows, it changes their order.

➡️ Sort `miFlights` by the day of the flight (smallest to largest), and print the first 4 columns and 5 rows of the resulting data set using the `slice_head()` function.

By default, `arrange()` sorts from smallest to largest, but we can sort the data from largest to smallest using `desc()`.

➡️ Sort `miFlights` by the day of the flight (largest to smallest), and print the first 4 columns and 5 rows of the resulting data set using the `slice_head()` function.

We can also arrange our data set by multiple variables simultaneously. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

➡️ Sort `miFlights` by the year, month, and day of the flight.

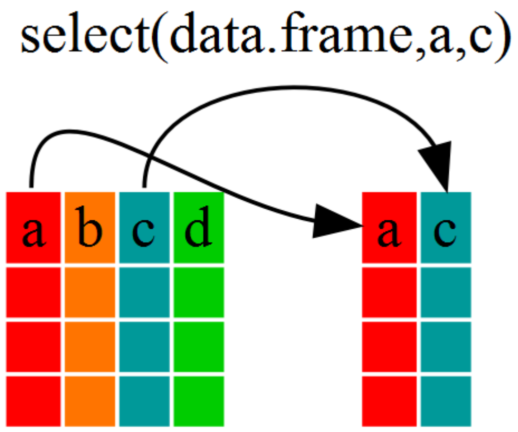
You try

1. Sort `miFlights` to find the 3 most delayed flights (`arr_delay`), and the 3 that left the earliest relative to their scheduled departure (`dep_delay`).

2. Sort `miFlights` to find the 3 fastest (highest speed) flights.

3. For flights coming out of GRR, find the 3 flights that traveled the farthest (`distance`) and that arrived the earliest in the morning (`arr_time`) simultaneously.

Select columns with `select()`



The `select()` function allows us to hone in on a useful subset using operations based on the names or positions of the variables.

We can list the names of columns out separated by commas, or use a colon to select sets of contiguous columns as well:

We can also drop columns by using the minus sign inside of `select()`.

➡️ Drop the `year` and `month` columns from `miFlights` creating a new data set called `miDropped`.

We can also drop sets of contiguous or touching columns:

➡️ Drop all variables between `year` and `day` columns (inclusive) from `miFlights` creating a new data set called `miDropped2`.

There are several "helper" functions for use with `select()`:

1. `starts_with("abc")`: matches names that begin with "abc".
2. `ends_with("xyz")`: matches names that end with "xyz".
3. `contains("ijk")`: matches names that contain "ijk".
4. `num_range("var", 1:3)`: matches `var1`, `var2` and `var3`.

Another option is to use `select()` in tandem with the `everything()` helper.

```
dplyr::select(miFlights, time_hour, air_time, everything())
```

We can use `select()` and `everything()` to rearrange columns and still drop columns too:

```
dplyr::select(miFlights, time_hour, air_time, everything(), -day)
```

You try

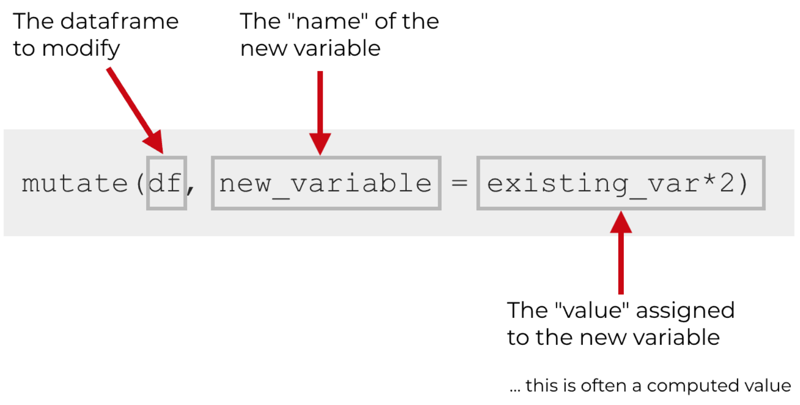
1. Create a subset of the `miFlights` data set called `timeFlights` that only contains variables that end with the word "time".

2. Create a new data frame called `departureInfo` that only has variables that start with "dep"

3. Create a new data frame call `newFlights` by rearranging the columns of the full `miFlights` data set so that flight number (`flight`), origin (`origin`), and destination (`dest`) are provided first, then all other columns except the tail number (`tailnum`).

Add new variables with `mutate()`

The `mutate()` function allows us to add new columns to a data set using functions of existing columns. Note that `mutate()` always appends columns to the end of the data set.


The dataframe to modify
The "name" of the new variable
mutate(df, new_variable = existing_var*2)
The "value" assigned to the new variable
... this is often a computed value

We can create a new variable, `gain`, that is a function of existing variables in `miFlights`:

```
flights_sml <- miFlights %>% dplyr::select(ends_with("delay"), distance,
air_time)

flights_sml %>% mutate(gain = dep_delay - arr_delay) %>%
  slice_head(n = 5)
```

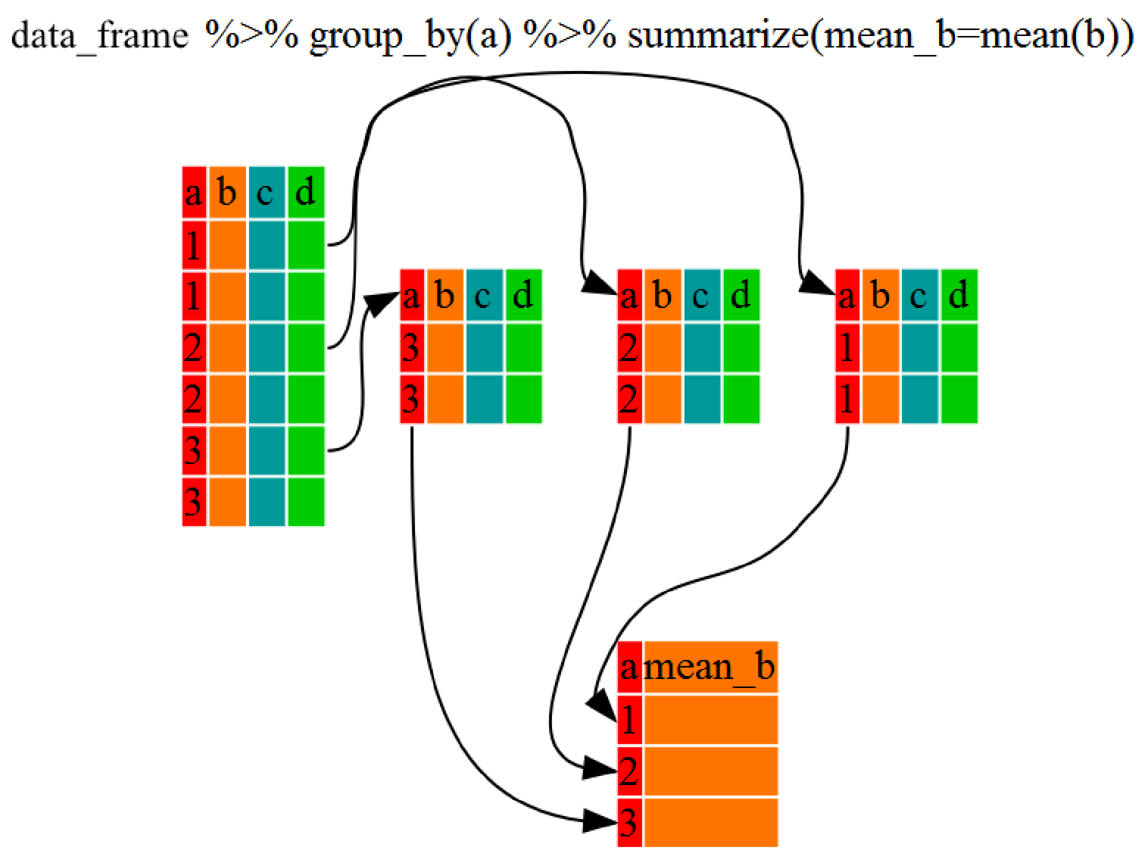
➡️ Extending the code provided with a single call to `mutate()`, create a new variable, `speed`, that is equal to `distance` divided by `air_time`, producing a new data set called `flightSpeeds`.

➡️ Create a plot showing the relationship between the speed and time gain of each flight, adding appropriate axis and title labels.

How can we better visualize the points since there are so many observations?

➡️ Add color to the plot to display the distance the flight traveled as well. Is there a noticeable pattern?

Group-wise operations and statistics with `group_by()` & `summarize()`



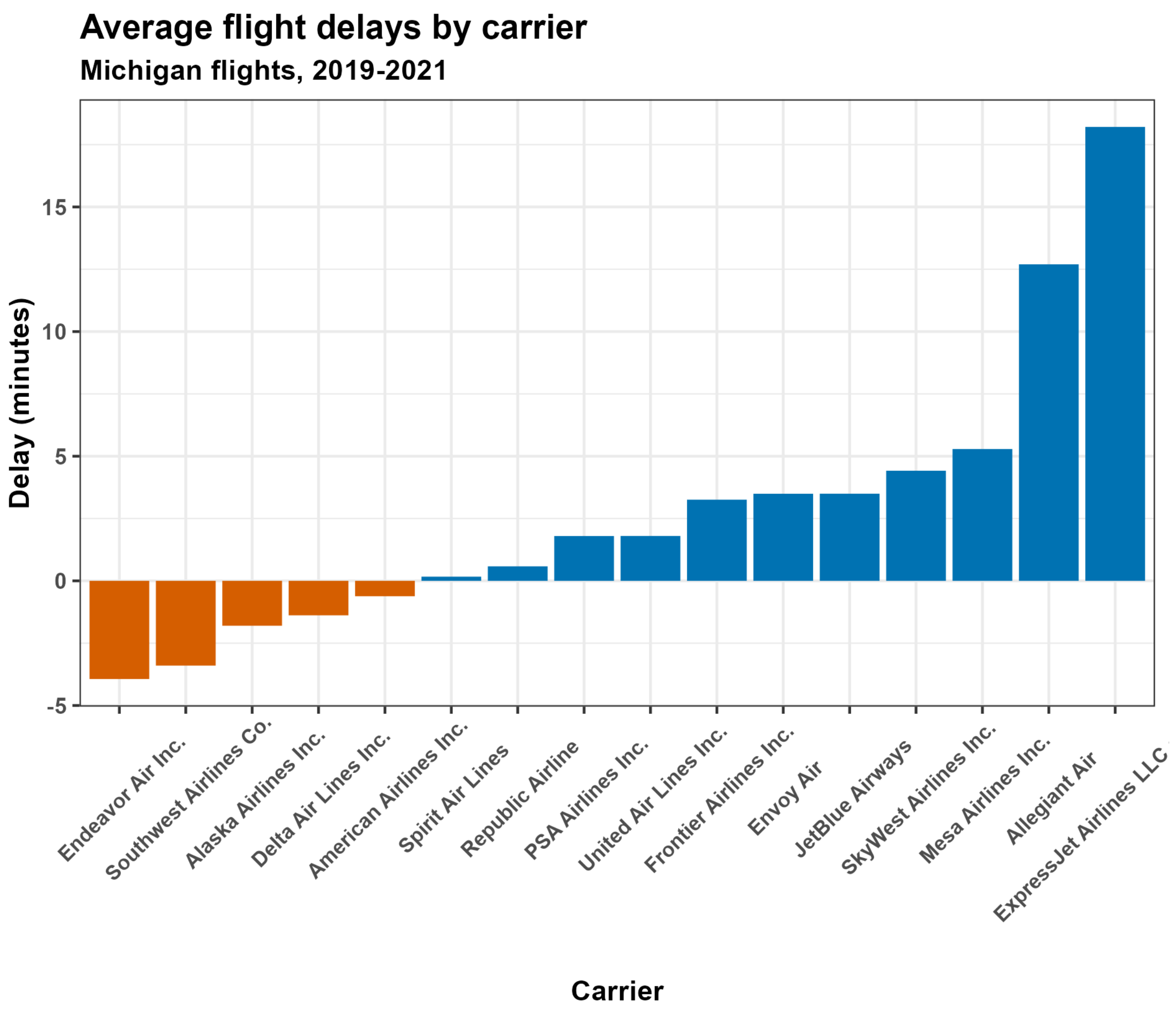
Together `group_by()` and `summarise()` provide useful tools: grouped data operations and summaries.

```
miFlights %>% group_by(year, month, day) %>%
summarize(delay = mean(dep_delay, na.rm = TRUE)) %>%
  slice_head(n = 5)
```

Let's see which airlines tend to have the worst delays. The code below creates a summary table containing the average flight delay in minutes for each carrier.

```
delaySummary <- miFlights %>% group_by(carrier_name) %>%
  summarize(Delay = mean(arr_delay, na.rm = T))
```

➡️ Reproduce the waterfall plot below using this summary table.



You try

1. Create a data frame summarizing the *median* flight delay (`arr_delay`) by month. Which month has the worst delays? In which month are flights most early / on-time?

2. Which type of plot would be most useful for displaying the typical delay each month? Creating and viewing this plot, are there any apparent trends?

3. Extend the plot in 2. by faceting by where the flight departed from (`origin`). You will need to use `group_by()` again to do this. What are your observations?

4. **Bonus (optional)**: Create a line chart showing the average daily flight delay across time for each of the major airports