STT 3850 : Week 3

Fall 2024

Appalachian State University

Section 1

Outline for the week

By the end of the week:

- Data Wrangling
- “Tidy” data

Section 2

Data Wrangling

Data Wrangling

In this chapter, we'll introduce a series of functions from the `dplyr` package for data wrangling. We will be able to take a data frame and **"wrangle" it (transform it)** to suit your needs. Such functions include:

1. `filter()` a data frame's existing rows to only pick out a subset of them.
2. `summarize()` one or more of its columns/variables with a summary statistic.
3. `group_by()` its rows. In other words, assign different rows to be part of the same group.
   - We can then combine `group_by()` with `summarize()` to report summary statistics for each group separately.

Data Wrangling

4. `mutate()` its existing columns/variables to create new ones. For example, convert hourly temperature recordings from degrees Fahrenheit to degrees Celsius.

5. `arrange()` its rows. For example, sort the rows of weather in ascending or descending order of temp.

6. `join()` it with another data frame by matching along a “key” variable. In other words, merge these two data frames together.

An additional benefit from learning to use the `dplyr` package for data wrangling is its similarity to the **SQL** (database querying language).

Needed packages

Let's load all the packages needed for this chapter.

```r
library(nycflights13)
library(ggplot2)
library(dplyr)
```

The pipe operator: |>

Before we start, let's first introduce a nifty tool that gets loaded with the `dplyr` package: **the pipe operator `|>`**.

- The pipe operator allows us to combine multiple operations in R into a single sequential chain of actions.

Let's start with a hypothetical example:

Say you would like to perform a hypothetical sequence of operations on a hypothetical data frame `x` using hypothetical functions $f()$, $g()$, and $h()$:

1. Take `x` then
2. Use `x` as an input to a function $f()$ then
3. Use the output of $f(x)$ as an input to a function $g()$ then
4. Use the output of $g(f(x))$ as an input to a function $h()$

The pipe operator: |>

One way to achieve this sequence of operations is by using nesting parentheses as follows:

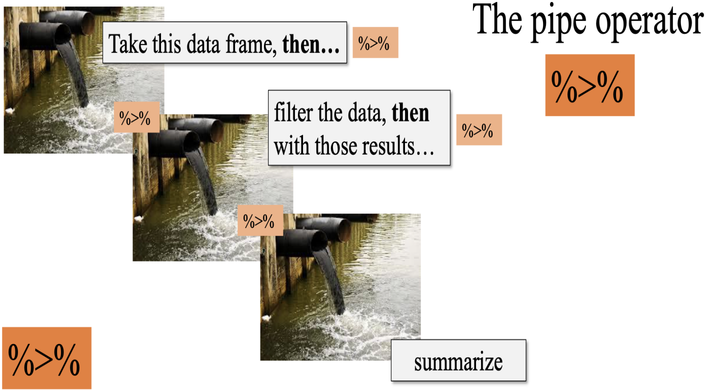
```
h(g(f(x)))
```

You can obtain the same output as the hypothetical sequence of functions as follows:

```
x |>        # take x
  f() |>    # Use this output as the input to f() then
  g() |>    # Use this output as the input to g() then
  h()        # Use this output as the input h()
```

This is much more human-readable because you can clearly read the sequence of operations line-by-line.

The pipe operator: |>



The pipe operator: |>

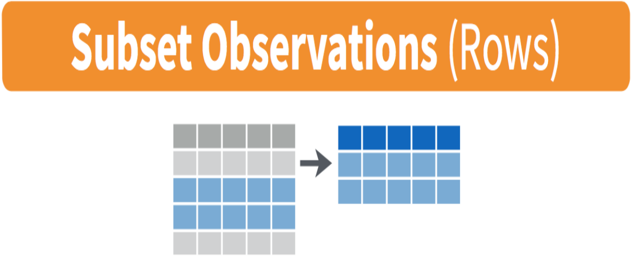
For example:

```
flights |>
  filter(carrier == "AS") |>
  select(year, month, arr_delay, dep_delay) |>
  slice_head(n = 3)
```

```
# A tibble: 3 x 4
   year month arr_delay dep_delay
  <int> <int>     <dbl>     <dbl>
1  2013     1       -10        -1
2  2013     1       -19        -7
3  2013     1       -41        -3
```

Note that the pipe operator |> has to come at the end of lines.

`filter` rows



The `filter()` function allows you to specify criteria about the values of a variable in your dataset and then filters out only the rows that match that criteria.

filter rows

- We begin by focusing only on flights from New York City to Portland, Oregon.
  - The dest destination code (or airport code) for Portland, Oregon is "PDX".
  - Run the following and look at the results in RStudio's spreadsheet viewer to ensure that only flights heading to Portland are chosen.

```
portland_flights <- flights |>
  filter(dest == "PDX")
# View(portland_flights)
```

We test for equality using the double equal sign == and not a single equal sign =.

`filter` rows

- You can use other operators beyond just the == operator that tests for equality:
  - > corresponds to "greater than"
  - < corresponds to "less than"
  - >= corresponds to "greater than or equal to"
  - <= corresponds to "less than or equal to"
  - != corresponds to "not equal to." The `!` is used in many programming languages to indicate "not".
- Furthermore, you can combine multiple criteria using operators that make comparisons:
  - | corresponds to "or"
  - & corresponds to "and"

filter rows

- We filter flights for all rows that
  - departed from JFK and
  - were heading to Burlington, Vermont ("BTV") or Seattle, Washington ("SEA") and
  - departed in the months of October, November, or December.

```
btv_sea_flights_fall <- flights |>
  filter(origin == "JFK" & (dest == "BTV" | dest == "SEA") &
         month >= 10)
# View(btv_sea_flights_fall)
```

One may use commas in place of &

```
btv_sea_flights_fall <- flights |>
  filter(origin == "JFK", (dest == "BTV" | dest == "SEA"),
         month >= 10)
# View(btv_sea_flights_fall)
```

filter rows

Lets filter rows corresponding to flights that didn't go to Burlington, VT or Seattle, WA.

```
not_BTV_SEA <- flights |>
  filter(!(dest == "BTV" | dest == "SEA"))
# View(not_BTV_SEA)
```

Note note the careful use of parentheses. The code below will produce different results.

```
flights |>
  filter(!dest == "BTV" | dest == "SEA")
```

filter rows

Say we have a larger number of airports we want to filter for. We could continue to use the | (or) operator:

```
many_airports <- flights |>
  filter(dest == "SEA" | dest == "SFO" | dest == "PDX" |
           dest == "BTV" | dest == "BDL")
```

A shorter approach will be to use `%in%` operator along with the `c()` function.

```
many_airports <- flights |>
  filter(dest %in% c("SEA", "SFO", "PDX", "BTV", "BDL"))
# View(many_airports)
```

The `%in%` operator is useful for looking for matches commonly in one vector/variable compared to another.

summarize variables

The next common task when working with data frames is to compute **summary statistics**. Summary statistics are single numerical values that summarize a large number of values.

- Commonly known examples of summary statistics include
  - the mean (also called the average) ,
  - the median (the middle value),
  - the sum,
  - the smallest value also called the minimum,
  - the largest value also called the maximum, and
  - the standard deviation.

summarize variables

Let's calculate two summary statistics (mean and standard deviation) of the `temp` temperature variable in the `weather` data frame from `nycflights13` package.

```
summary_temp <- weather |>
  summarize(mean = mean(temp), std_dev = sd(temp))
summary_temp
```

```
# A tibble: 1 x 2
   mean std_dev
  <dbl>   <dbl>
1    NA      NA
```

- NAs appear as the answers since `temp` has `NA` values.

summarize variables

- If you want to ignore the NA values:
  - Set the na.rm argument to TRUE.
  - rm is short for "remove"; this will ignore any NA missing values and only return the summary value for all non-missing values.

```
summary_temp <- weather |>
  summarize(mean_temp = mean(temp, na.rm = TRUE),
            sd_temp = sd(temp, na.rm = TRUE))
summary_temp
```

```
# A tibble: 1 x 2
  mean_temp sd_temp
      <dbl>   <dbl>
1      55.3    17.8
```

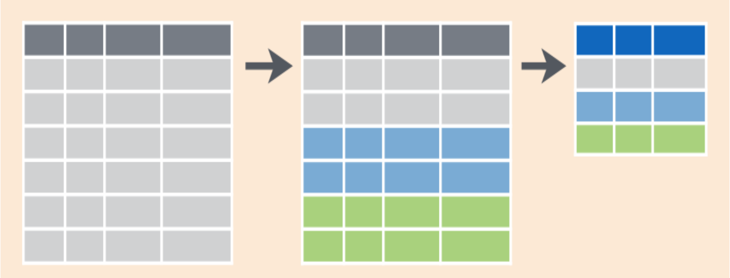
`summarize` variables

Other summary functions we can use inside the `summarize()`:

- `mean()`: the average
- `sd()`: the standard deviation, which is a measure of spread
- `min()` and `max()`: the minimum and maximum values, respectively
- `IQR()`: interquartile range
- `sum()`: the total amount when adding multiple numbers
- `n()`: a count of the number of rows in each group

group_by

- Say instead of a single mean temperature for the whole year, you would like 12 mean temperatures, one for each of the 12 months separately.
  - We would like to compute the mean temperature split by month.
  - We can do this by "grouping" temperature observations by the values of another variable, in this case by the 12 values of the variable `month`.



group_by

```
summary_monthly_temp <- weather |>
  group_by(month) |>
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE),
            count = n())
summary_monthly_temp |>
  slice_head(n = 3)
```

```
# A tibble: 3 x 4
  month  mean std_dev count
  <int> <dbl>   <dbl> <int>
1     1  35.6   10.2   2226
2     2  34.3    6.98  2010
3     3  39.9    6.25  2227
```

Grouping by more than one variable

We can also group by more than one variable

```
by_origin_monthly <- flights |>
  group_by(origin, month) |>
  summarize(count = n())
dim(by_origin_monthly)
```

```
[1] 36  3
```

```
by_origin_monthly |> head(n = 2)
```

```
# A tibble: 2 x 3
# Groups:   origin [1]
  origin month count
  <chr>  <int> <int>
1 EWR        1  9893
2 EWR        2  9107
```

Observe that there are 36 rows to `by_origin_monthly` because there are 12 months for 3 airports (`EWR`, `JFK`, and `LGA`).

Grouping by more than one variable

Why do we `group_by(origin, month)` and not `group_by(origin)` and then `group_by(month)`? Let's investigate:

```
by_origin_monthly_incorrect <- flights |>
  group_by(origin) |>
  group_by(month) |>
  summarize(count = n())
dim(by_origin_monthly_incorrect)
```

```
[1] 12  2
```

```
by_origin_monthly_incorrect |> head(n = 2)
```

```
# A tibble: 2 x 2
  month count
  <int> <int>
1     1 27004
2     2 24951
```

The second `group_by(month)` overwrote `group_by(origin)`.

`mutate` existing variables

Another common transformation of data is to create/compute new variables based on existing ones.

For example, we can create a new variable by converting temperatures from °F to °C using the formula

$$\text{temp in C} = \frac{\text{temp in F} - 32}{1.8}$$

mutate existing variables

We can apply this formula to the temp variable using the mutate() function from the dplyr package.

```
weather |>
  mutate(temp_in_C = (temp - 32) / 1.8,
         monthT = month.name[month]) -> weather
weather |> select(month, monthT, temp_in_C, temp) |>
  head(n = 2)
```

```
# A tibble: 2 x 4
  month monthT  temp_in_C  temp
  <int> <chr>       <dbl> <dbl>
1     1 January       3.9  39.0
2     1 January       3.9  39.0
```

`mutate` existing variables

- In this code:
  - we `mutate()` the `weather` data frame by creating a new variable `temp_in_C = (temp - 32) / 1.8`,
  - create a new variable `monthT` that has the name of the months,
  - then overwrite the original weather data frame.

mutate existing variables

Let's compute monthly average temperatures in both °F and °C.

```
summary_monthly_temp <- weather |>
  group_by(month, monthT) |>
  summarize(mean_temp_in_F = mean(temp, na.rm = TRUE),
            mean_temp_in_C = mean(temp_in_C, na.rm = TRUE))
summary_monthly_temp |>
  head(n = 4)
```

```
# A tibble: 4 x 4
# Groups:   month [4]
  month monthT   mean_temp_in_F mean_temp_in_C
  <int> <chr>             <dbl>          <dbl>
1     1 January            35.6           2.02
2     2 February           34.3           1.26
3     3 March              39.9           4.38
4     4 April              51.7          11.0
```

mutate existing variables

Let's consider another example.

- Passengers are often frustrated when their flight departs late, but aren't as annoyed if, in the end, pilots can make up some time during the flight.
- This is known in the airline industry as `gain`, and we will create this variable using the `mutate()` function:

```
flights <- flights |>
  mutate(gain = dep_delay - arr_delay)
flights  |>
  select(dep_delay, arr_delay, gain) -> flights_gain
```

mutate existing variables

```
flights_gain
```

```
# A tibble: 336,776 x 3
   dep_delay arr_delay  gain
       <dbl>     <dbl> <dbl>
 1         2        11    -9
 2         4        20   -16
 3         2        33   -31
 4        -1       -18    17
 5        -6       -25    19
 6        -4        12   -16
 7        -5        19   -24
 8        -3       -14    11
 9        -3        -8     5
10        -2         8   -10
# i 336,766 more rows
```

mutate existing variables

Let's look at some summary statistics of the gain variable

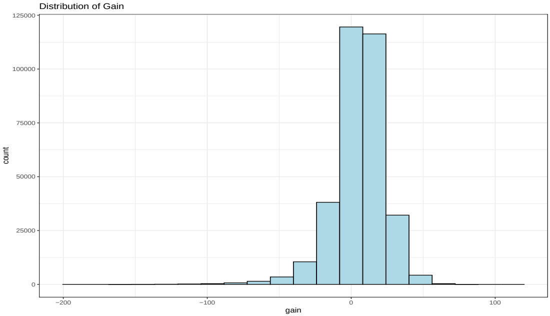
```
gain_summary <- flights |>
  summarize(
    Min = min(gain, na.rm = TRUE),
    Q1 = quantile(gain, 0.25, na.rm = TRUE),
    Median = quantile(gain, 0.5, na.rm = TRUE),
    Q3 = quantile(gain, 0.75, na.rm = TRUE),
    Max = max(gain, na.rm = TRUE),
    Mean = mean(gain, na.rm = TRUE),
    SD = sd(gain, na.rm = TRUE),
    Missing = sum(is.na(gain))
  )
gain_summary
```

```
# A tibble: 1 x 8
    Min    Q1 Median    Q3   Max  Mean    SD Missing
  <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>   <int>
1  -196    -3      7    17   109  5.66  18.0    9430
```

mutate existing variables

Since `gain` is a numerical variable, we can visualize its distribution using a histogram.

```
ggplot(data = flights, mapping = aes(x = gain)) +
  geom_histogram(color = "black", fill = "lightblue",
                 bins = 20) +
  theme_bw() +
  labs(title = "Distribution of Gain")
```



`arrange` and sort rows

- One of the most commonly performed data wrangling tasks is to sort a data frame's rows in alphanumeric order by one of the variables in the data frame/tibble.
- The `dplyr` package's `arrange()` function allows us to sort/reorder a data frame's rows according to the values of a specified variable.

arrange and sort rows

Suppose we are interested in determining the most frequent destination airports for all domestic flights departing from New York City in 2013.

```
freq_dest <- flights |>
  group_by(dest) |>
  summarize(num_flights = n())
freq_dest |> head(n = 3)
```

```
# A tibble: 3 x 2
  dest  num_flights
  <chr>       <int>
1 ABQ           254
2 ACK           265
3 ALB           439
```

Observe that by default the rows of the resulting `freq_dest` data frame are sorted in alphabetical order of destination.

arrange and sort rows

Say instead we would like to see the same data, but sorted from the most to the least number of flights (`num_flights`) instead

```
freq_dest |>
  arrange(num_flights) |>
  head(n = 5)
```

```
# A tibble: 5 x 2
  dest  num_flights
  <chr>       <int>
1 LEX             1
2 LGA             1
3 ANC             8
4 SBN            10
5 HDN            15
```

This is, however, the opposite of what we want. The rows are sorted with the least frequent destination airports displayed first.

arrange and sort rows

To switch the ordering to be in "descending" order instead, we use the `desc()` function.

```
freq_dest |>
  arrange(desc(num_flights)) |>
  head(n = 5)
```

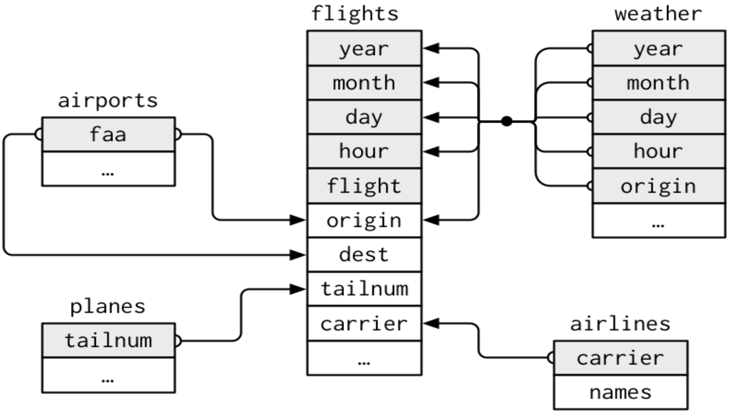
```
# A tibble: 5 x 2
  dest  num_flights
  <chr>       <int>
1 ORD         17283
2 ATL         17215
3 LAX         16174
4 BOS         15508
5 MCO         14082
```

`join` data frames

Another common data transformation task is "joining" or "merging" two different datasets.

- For example, in the `flights` data frame, the variable carrier lists the carrier code for the different flights.
- While the corresponding airline names for "UA" and "AA" might be somewhat easy to guess (United and American Airlines), what airlines have codes "VX", "HA", and "B6"?
- This information is provided in a separate data frame `airlines`.

Lets see the data relationships from the `nycflights13` package.



Matching "key" variable names

In both the `flights` and `airlines` data frames, the key variable we want to join/merge/match the rows by has the same name: **carrier**.

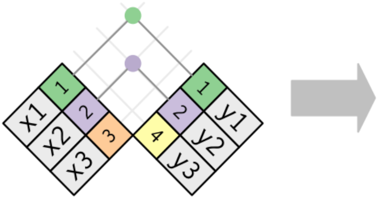
- Let's use the `inner_join()` function to join the two data frames, where the rows will be matched by the variable carrier, and then compare the resulting data frames:

```
flights_joined <- flights |>
  inner_join(airlines, by = "carrier")
# View(flights)
# View(flights_joined)
```

Observe that the `flights` and `flights_joined` data frames are identical except that `flights_joined` has an additional variable `name`.

Matching "key" variable names

A visual representation of the inner_join() is shown below



key	val_x	val_y
1	x1	y1
2	x2	y2

- There are other types of joins available,
    - such as left_join(), right_join(), outer_join(), and anti_join(),
    - but the inner_join() will solve nearly all of the problems you'll encounter in this class.

Different “key” variable names

- Say instead you are interested in the destinations of all domestic flights departing NYC in 2013,
  - and you ask yourself questions like: “What cities are these airports in?”, or “Is `ORD` Orlando?”, or “Where is `FLL`?”
- The `airports` data frame contains the airport codes for each airport:

```
# View(airports)
```

- However, if you look at both the `airports` and `flights` data frames, you’ll find that the airport codes are in variables that have different names.
  - In `airports` the airport code is in `faa`.
  - whereas in `flights` the destination airport codes are in `dest`.
  - This fact is further highlighted in the visual representation of the relationships between these data frames.

Different “key” variable names

In order to join these two data frames by airport code, our `inner_join()` operation will use the `by = c("dest" = "faa")` argument:

```
flights_with_airport_names <- flights |>
  inner_join(airports, by = c("dest" = "faa"))
# View(flights_with_airport_names)
```

Different “key” variable names

Let's construct the chain of pipe operators |> that computes the number of flights from NYC to each destination, but also includes information about each destination airport:

```
named_dests <- flights |>
  group_by(dest) |>
  summarize(num_flights = n()) |>
  arrange(desc(num_flights)) |>
  inner_join(airports, by = c("dest" = "faa")) |>
  rename(airport_name = name)
named_dests[1:2, 1:4]
```

```
# A tibble: 2 x 4
  dest  num_flights airport_name                    lat
  <chr>       <int> <chr>                         <dbl>
1 ORD         17283 Chicago Ohare Intl             42.0
2 ATL         17215 Hartsfield Jackson Atlanta Intl  33.6
```

Multiple "key" variables

- Say instead we want to join two data frames by multiple key variables.
- For example, from the visual representation of the relationships between the data frame:
  - we see that in order to join the `flights` and `weather` data frames, we need more than one key variable: `year`, `month`, `day`, `hour`, and `origin`.
  - This is because the combination of these 5 variables act to uniquely identify each observational unit in the weather data frame: hourly weather recordings at each of the 3 NYC airports.
- We achieve this by specifying a vector of key variables to join by using the `c()` function.
  - Recall, this function is short for "combine" or "concatenate."

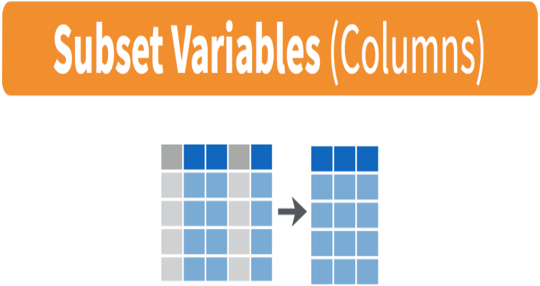
Multiple “key” variables

```r
flights_weather_joined <- flights |>
  inner_join(weather, by = c("year", "month", "day",
                             "hour", "origin"))
flights_weather_joined[1:5, 1:6]
```

```
# A tibble: 5 x 6
   year month   day dep_time sched_dep_time dep_delay
  <int> <int> <int>    <int>          <int>     <dbl>
1  2013     1     1      517            515         2
2  2013     1     1      533            529         4
3  2013     1     1      542            540         2
4  2013     1     1      544            545        -1
5  2013     1     1      554            600        -6
```

```r
# View(flights_weather_joined)
```

select variables



We've seen that the `flights` data frame in the `nycflights13` package contains 19 different variables.

```r
#glimpse(flights)
```

However, say you only need two of these 19 variables, say `carrier` and `flight`. You can `select()` these two variables:

```
flights_sub <-flights |>
  select(carrier, flight)
```

Let's say instead you want to drop, or de-select, certain variables. For example, lets say we want to remove the `year` in the `flights` data frame. We can deselect year by using the - sign:

```
flights_no_year <- flights |> select(-year)
```

select variables

Another way of selecting columns/variables is by specifying a range of columns:

```r
names(flights)
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_t
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"      "gain"
```

```r
flight_arr_times <- flights |>
  select(month:day, arr_time:sched_arr_time)
#flight_arr_times
```

This will `select()` all columns between `month` and `day`, as well as between `arr_time` and `sched_arr_time`, and drop the rest.

select variables

The `select()` function can also be used to reorder columns when used with the `everything()` helper function.

- For example, suppose we want the `hour`, `minute`, and `time_hour` variables to appear immediately after the `year`, `month`, and `day` variables, while not discarding the rest of the variables.
- In the following code, `everything()` will pick up all remaining variables:

```
flights_reorder <- flights |>
  select(year, month, day, hour, minute, time_hour,
         everything())
# glimpse(flights_reorder)
```

select variables

Lastly, the helper functions `starts_with()`, `ends_with()`, and `contains()` can be used to select variables/columns that match those conditions. As examples,

```
flights_sub1 <- flights |> select(starts_with("a"))
flights_sub2 <- flights |> select(ends_with("delay"))
flights_sub3 <- flights |> select(contains("time"))
```

Summary table

Verb	Data wrangling operation
`filter()`	Pick out a subset of rows
`summarize()`	Summarize many values to one using a summary statistic function like `mean()`, `median()`, etc.
`group_by()`	Add grouping structure to rows in data frame. Note this does not change values in data frame, rather only the meta-data
`mutate()`	Create new variables by mutating existing ones
`arrange()`	Arrange rows of a data variable in ascending (default) or `desc`ending order
`inner_join()`	Join/merge two data frames, matching rows by a key variable

Section 3

"Tidy" data

"Tidy" data

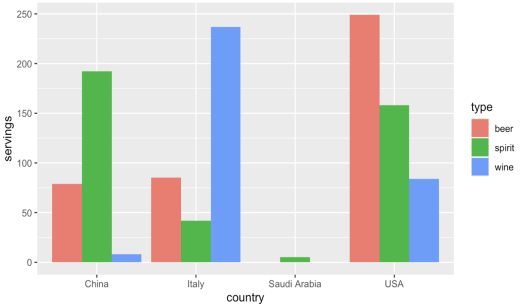
Let's now learn about the concept of **"tidy"** data format.

```
library(fivethirtyeight)
drinks_smaller <- drinks |>
  filter(country %in% c("USA", "China", "Italy",
                        "Saudi Arabia")) |>
  select(-total_litres_of_pure_alcohol) |>
  rename(beer = beer_servings, spirit = spirit_servings,
         wine = wine_servings)
drinks_smaller
```

```
# A tibble: 4 x 4
  country       beer spirit  wine
  <chr>        <int>  <int> <int>
1 China           79    192     8
2 Italy           85     42   237
3 Saudi Arabia     0      5     0
4 USA            249    158    84
```

“Tidy” data

The `drinks_smaller` data frame, cannot be used to create the side-by-side barplot show below.



why?

“Tidy” data

- Let’s break down the grammar of graphics we introduced earlier:
  - The categorical variable country with four levels (China, Italy, Saudi Arabia, USA) would have to be mapped to the x-position of the bars.
  - The numerical variable servings would have to be mapped to the y-position of the bars (the height of the bars).
  - The categorical variable type with three levels (beer, spirit, wine) would have to be mapped to the fill color of the bars.
- To recreate the barplot above, our data frame should be in the **“tidy”** format.

"Tidy" data: Definition

The word "tidy" in data science means that your data follows a standardized format.

- "Tidy" data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.
- In tidy data:
  - Each variable forms a column.
  - Each observation forms a row.
  - Each type of observational unit forms a table.

"Tidy" data: Definition

Stock prices (non-tidy format):

Date	Boeing stock price	Amazon stock price	Google stock price
2009-01-01	$173.55	$174.90	$174.34
2009-01-02	$172.61	$171.42	$170.04

Stock prices (tidy format):

Date	Stock Name	Stock Price
2009-01-01	Boeing	$173.55
2009-01-01	Amazon	$174.90
2009-01-01	Google	$174.34
2009-01-02	Boeing	$172.61
2009-01-02	Amazon	$171.42
2009-01-02	Google	$170.04

"Tidy" data: Definition

- Observe that:
  - The non-tidy format of the stock prices is what's known as "wide" format, whereas - the tidy format is known as "long/narrow" format.
- In the context of doing data science, **long/narrow** format is also known as **"tidy"** format.
- In order to use the ggplot2 and dplyr packages for data visualization and data wrangling, your input data frames must be in "tidy" format.
- Thus, all non-"tidy" data must be converted to "tidy" format first.

Coverting to "Tidy" format

We convert the `drinks_smaller` data to "tidy" format by using the `pivot_longer()` function from the `tidyr` package:

```
library(tidyr)
drinks_smaller_tidy <- drinks_smaller |>
  pivot_longer(names_to = "type",
               values_to = "servings",
               cols = -country)
```

Coverting to "Tidy" format

```
drinks_smaller_tidy
```

```
# A tibble: 12 x 3
   country      type    servings
   <chr>        <chr>      <int>
 1 China        beer          79
 2 China        spirit       192
 3 China        wine           8
 4 Italy        beer          85
 5 Italy        spirit        42
 6 Italy        wine         237
 7 Saudi Arabia beer           0
 8 Saudi Arabia spirit         5
 9 Saudi Arabia wine           0
10 USA          beer         249
11 USA          spirit       158
12 USA          wine          84
```

Coverting to “Tidy” format

Note that the two ways to specify the vector argument to `cols` produce the same output.

```
library(tidyr)
drinks_smaller |>
  pivot_longer(names_to = "type",
               values_to = "servings",
               cols = c(beer, spirit, wine))
# Or
drinks_smaller |>
  pivot_longer(names_to = "type",
               values_to = "servings",
               cols = beer:wine)
```

Coverting to "Tidy" format

```
library(tidyr)
ggplot(drinks_smaller_tidy, aes(x = country, y = servings,
                                fill = type)) +
  geom_col(position = "dodge") +
  theme_bw()
```