Hertie School/SCRIPTS Data Science Workshop Series

Session 2: Modern data management with R

Therese Anders[*]
Allison Koh[†]

January 17, 2020

Data wrangling using `dplyr`

Introduction

Transforming and combining data from different sources into formats suitable for statistical analysis and visualization lies at the heart of the data science workflow. In many data science applications, an estimated 50% to 80% of a project is spent on data collection, cleaning, and wrangling. Today, we will be introducing the `dplyr`, `tidyr`, and `lubridate` packages. Together with `stringr` (string operations using regular expressions), these packages offer functionality for the majority of data cleaning and reshaping task in `R`.

RStudio has a number of great cheat sheets available online: https://rstudio.com/resources/cheatsheets/. Check out the "Data transformation with `dplyr` cheat sheet!

The data

For this exercise, we will be using data from Five Thirty Eight's Presidential Endorsement Tracker, which records endorsements of American presidential primary candidates from current/former government officials and high-ranked political party members (i.e. members of the Democratic National Convention). For this dataset, an endorsement is defined as a "public display or pronouncement of support that articulates or strongly implies that a candidate is an endorser's current No. 1 choice for president". This means that there is only one endorsement linked to each endorser, and that they can change over the course of the primary season. In the case of ambiguities, Five Thirty Eight reaches out to offices of the candidates and endorsers for a clarification.

In this model, endorsements are "ranked" by their perceived relative value based on their current/former position. For the 2020 Democratic Presidential Primaries, it comprises of the following:

1. Current and former presidents and vice presidents (10 points).
2. Current party leaders: Nancy Pelosi (House speaker), Steny Hoyer (House majority leader), James Clyburn (House majority whip), Chuck Schumer (Senate minority leader), Dick Durbin (Senate minority whip) and Tom Perez (Democratic National Committee chair) (10 points).
3. Current governors, including governor equivalents from the U.S. territories and Washington, D.C.1 (8 points).
4. Current U.S. Senators (6 points).
5. Past presidential and vice presidential nominees (5 points).
6. Former party leaders2 (5 points).
7. Former 2020 presidential candidates who appeared in at least one debate and have since dropped out (5 points).
8. Current U.S. representatives, including non-voting delegates from U.S. territories (3 points).
9. Mayors of cities with at least 300,000 people (3 points).
10. Officials holding statewide or territory-wide elected office, excluding positions (e.g. commissioners) that are held by multiple people at once (2 points).
11. State and territorial legislatures' majority and minority leaders (2 points).3

[*]Instructor, Hertie School/SCRIPTS, anders@hertie-school.org.
[†]Teaching Assistant, Hertie School, kohallison3@gmail.com.

12. Other DNC members (1 point).

The dataset contains the following variables:

- `date`: date endorsement took place
- `endorser`: individual who is endorsing a presidential candidate
- `position`: endorser's position in government
- `state`: origin state of endorser
- `endorsee`: presidential candidate endorsed
- `endorser.party`: political party of the endorser
- `source`: link to endorsement (if online)
- `points`: # of "endorsement points" attributed to the endorser (see above for more details)
- `drop`: Has the candidate dropped out of the race? (1 = yes)
- `drop_date`: date that the candidate dropped out (This variable is not part of 'fivethirtyeight. Allison coded this variable for us. Thank you!).

Below, notice the additional parameters we give `read.csv()`. We covered `stringsAsFactors` last week: it prevents string variables from being imported as factors and instead defaults to character type.

```r
library(foreign)
raw <- read.csv("538_presidential_endorsements.csv",
               stringsAsFactors = FALSE)
```

Functions in `dplyr`

`dplyr` uses a strategy called "Split - Apply - Combine". Some of the key functions include:

- `select()`: Subset columns.
- `filter()`: Subset rows.
- `arrange()`: Reorders rows.
- `mutate()`: Add columns to existing data.
- `summarise()` or `summarize()`: Summarizing the data set.

First, lets download the package and call it using the `library()` function.

```r
# install.packages("dplyr")
library(dplyr)
```

Using `select()` and introducing the Piping Operator `%>%`

Using the so-called **piping operator** will make the `R` code faster and more legible, because we are not saving every output in a separate data frame, but passing it on to a new function. First, let's use only a subsample of variables in the data frame.

Notice a couple of things in the code below:

- We can assign the output to a new data set.
- We use the piping operator to connect commands and create a single flow of operations.
- We can use the select function to rename variables.
- Instead of typing each variable, we can select sequences of variables.
- Note that the `everything()` command inside `select()` will select all variables.

NOTE: The `::` operator specifies that we want to use the *object* `select()` from the *package* `dplyr`. This explicit programming is useful when functions or objects are contained in multiple packages to avoid confusion. In practice, I now always write `dplyr::select()` because I have come across a few instances where code broke because `select` was part of another package.

Selecting variables with `select()`

```r
# Explicitly selecting variables one-by-one
df_sub1 <- raw %>%
  dplyr::select(position,
                endorsee)

# Selecting multiple variables at once
df_sub2 <- raw %>%
  dplyr::select(position,
                endorsee,
                source:drop_date) #Selecting a number of columns.
names(df_sub2)
```

```
## [1] "position"  "endorsee"  "source"    "category"  "points"    "drop"
## [7] "drop_date"
```

```r
# Later we will also learn a few additional tricks using helper functions
```

Explicitly dropping variables with `select()`

Suppose, we didn't really want to select the `source` variable. We can use `select()` to drop variables.

```r
df_sub3 <- raw %>%
  dplyr::select(-source)
```

Reordering variables with `select()`

```r
df_sub3 <- raw %>%
  dplyr::select(date,
                endorsee,
                endorser,
                points)
```

Renaming variables with `select()`

We can also use `select()` to rename variables. Below, we need to use the `everything()` function if we don't want to drop the remaining variables in the process

```r
names(raw)
```

```
##  [1] "date"           "position"       "state"          "endorser"
##  [5] "endorsee"       "endorser.party" "source"         "category"
##  [9] "points"         "drop"           "drop_date"
```

```r
df <- raw %>%
  dplyr::select(endorse_date = date,
                endorser_party = endorser.party,
                everything())
```

Helper functions

`dplyr` has a number of helper functions — and that is where the magic lies. These can be used with either `select()` or `filter()`. Here are some useful functions:

- `starts_with()`
- `ends_with()`

- `contains()`
- `matches()`: More general than `contains()` because it allows for regular expressions. We won't do an example here because we will go over regular expressions in another session.

For example, let's create a data frame with all variables that contain the word "endorse".

```r
df_endorse <- df %>%
  dplyr::select(contains("endorse"))
names(df_endorse)
```

```
## [1] "endorse_date"   "endorser_party" "endorser"       "endorsee"
```

Ot we could select all variables that start with the letter "p".

```r
df_startsp <- df %>%
  dplyr::select(starts_with("p"))
names(df_startsp)
```

```
## [1] "position" "points"
```

Introducting `filter()`

While `select()` operates on columns, `filter()` operates on observations (rows). We always use `filter()` in combination with logical statements. First, let's briefly review the most commonly used logical statements from last week.

- `x < y`
- `x <= y`
- `x == y`
- `x != y`
- `x >= y`
- `x > y`
- `x %in% c(a, b, c)` is `TRUE` if `x` is in the vector `c(a, b, c)`.

There are two more important logical statements we frequently use in data wrangling. * `is.na()` * `!is.na()`

First, let's take a look at the `endorser_party` variable we created above using the `table()` function. Unsurprisingly, most endorsements come from Democrats.

Who are the non-Democrat endorsers? We can use `filter()` to do so.

```r
table(df$endorser_party)
```

```
##
##   D   I NPP   R
## 992   3   2   1
```

```r
df_nondem <- df %>%
  filter(endorser_party != "D")
```

Ok, so there are a number of rows for which the `endorsee` variable is `NA`. That is weird. Lets check out how many cases we have in which the `endorsee` variable is `NA`.

```r
table(is.na(df$endorsee))
```

```
##
## FALSE  TRUE
##   223   782
```

That is most of our observations! So `fivethirtyeight` appears to have already listed people it expects to announce an endorsement before they even did so! We can use filter to create a new data frame that only contains observations for which an endorsement actually happened using `is.na()` (or more specifically the negation `!is.na()`).

```
df_nona <- df %>%
  filter(!is.na(endorsee))
```

Filtering for multiple values at once

Suppose, we wanted to filter all the endorsements that come from states that will take part in Super Tuesday. Registered Democrats from these states will vote for the Democratic nominee, who will be on the ballot for the 2020 Presidential Election.

The states that will hold primary elections on this year's Super Tuesday (March 3, 2020) are Alabama (AL), Arkansas (AR), California (CA), Colorado (CO), Maine (ME), Massachussets (MA), Minnesota (MN), North Carolina (NC), Oklahoma (OK), Tennessee (TN), Texas (TX), Utah (UT), Vermont (VT), and Virginia (VA).

```
super_tuesday <- c("AL", "AR", "CA", "CO", "ME", "MA", "MN", "NC", "OK", "TN", "TX", "UT", "VT", "VA")
df_super <- df %>%
  filter(state %in% super_tuesday)
```

Wait, what do we need `%in%` for? Couldn't we just use `==`? Lets use a smaller example to illustrate why this isn't the case. Below, lets select only the following states: California (CA) and Texas (TX).

```
head(df$state)
```

```
## [1] "MD" "NY" "CA" "DE" "TX" "CA"
```

```
test1 <- df %>%
  filter(state %in% c("CA", "TX"))
nrow(test1)
```

```
## [1] 132
```

```
head(test1$state)
```

```
## [1] "CA" "TX" "CA" "CA" "CA" "CA"
```

```
test2 <- df %>%
  filter(state == c("CA", "TX"))
```

```
## Warning in state == c("CA", "TX"): longer object length is not a multiple
## of shorter object length
```

```
nrow(test2)
```

```
## [1] 61
```

```
head(test2$state)
```

```
## [1] "CA" "CA" "CA" "CA" "CA" "CA"
```

What does `R` do? Well it does a vector matching operation. From the variable `df$state` (`"MD" "NY" "CA" "DE" "TX" "CA"`) it test is the first element "CA", is the second element "TX", is the third element "CA", the fourth element "TX", and so on. Only if the return for each of these checks is `TRUE` does it select the respective row (observation).

So yes, if you want to filter for multiple values at once, you need to use `%in%`!

Introducting `mutate()`

We can use the `mutate()` function to create new variables or change existing ones.

Suppose we want to rescale the `points` variable to run from 0 to 1. Since the original variable is scaled from 1 (least valuable endorsement) to 10 (most valuable endorsement; former presidents and vice presidents; current party leaders), we can create a new variable `points_rescaled` by dividing the original variable by 10.

```r
table(df$points)
```

```
##
##   1   2   3   5   6   8  10
## 404 209 278  33  41  29  11
```

```r
df_rescaled <- df %>%
  dplyr::mutate(points_rescaled = points/10)
```

Exercise Quick question to check our understanding: What is the difference in the output of the following two lines of code?

`mutate(us_state = state)`

`select(us_state = state)`

Suppose, rather than simply dropping all observations that are part of the Super Tuesday group, I wanted to add a binary variable that is `1` if the state has a primary on Super Tuesday and 0 if it doesn't.

Here, we use the `ifelse()` function that is tremendously helpful in data wrangling exercises. The syntax of `ifelse()` is as follows:

`ifelse(condition, value if TRUE, value if FALSE).`

```r
df_super2 <- df %>%
  mutate(super = ifelse(state %in% super_tuesday, 1, 0))
```

Recoding variables

We can also use `mutate()` to recode variables. Suppose I only wanted to look at endorsements for candidates who are still in the race, but it is not updated to include the candidate who most recently dropped out of the race (Cory Booker). We can use the `replace()` function inside `mutate()` to make this change.

The syntax for `replace()` is similar to that of `ifelse()`:

`mutate(variable = replace(variable, condition, value if TRUE))`

```r
df_drop <- df %>%
  mutate(drop = replace(drop, endorsee == "Cory Booker", 1))
```

Super-helper `case_when()`

We can use the `case_when()` function to use complex logical queries to create new variables. Think of `case_when()` as a supercharged `ifelse()`. To get used to the syntax, let's use it as an alternative to `ifelse()`.

```r
df_super_alt <- df %>%
  mutate(newvar = case_when(
    state %in% super_tuesday ~ 1, #Read: "If state is part of super_tuesday"
    T ~ 0 #Read: "In all other cases"
  ))
```

OK, that seems a lot more difficult that just using `ifelse()`. But not so fast, `case_when()` can do *a lot* more. Suppose, we wanted to re-group (i.e. simplify) the `position` variable, because it has a ton of different categories.

```r
table(df$position)
```

```
## 
##                                   attorney general 
##                                                 24 
##                                    auditor general 
##                                                  1 
##                                auditor of accounts 
##                                                  1 
##                                           chairman 
##                                                  1 
## commissioner of agriculture and consumer services 
##                                                  1 
##                          commissioner of insurance 
##                                                  1 
##              commissioner of labor and industries 
##                                                  1 
##                       commissioner of public lands 
##                                                  2 
##                                           delegate 
##                                                  4 
##                        Democratic caucus chairman 
##                                                  2 
##                                          DNC chair 
##                                                  1 
##                                         DNC member 
##                                                404 
##                          Executive Board chairman 
##                                                  1 
##                            Executive Board member 
##                                                  1 
##                             Former 2020 candidate 
##                                                  7 
##                                   former DNC chair 
##                                                 15 
##                former U.S. House Democratic whip 
##                                                  2 
##                former U.S. House majority leader 
##                                                  1 
##               former U.S. Senate majority leader 
##                                                  3 
##                                  general treasurer 
##                                                  1 
##                                           governor 
##                                                 28 
##                             insurance commissioner 
##                                                  2 
##                                lieutenant governor 
##                                                 24 
##                                    majority leader 
##                                                  1 
##                                              mayor 
##                                                 48 
##                              minority floor leader
##                                               2
##                                  minority leader
##                                              51
##                                   past president
##                                               3
##                        past presidential nominee
##                                               3
##                              past vice president
##                                               2
##                   past vice-presidential nominee
##                                               2
##                                        president
##                                              14
##                                president pro tem
##                                               7
##                                   public auditor
##                                               1
##                                   representative
##                                             227
##                               secretary of state
##                                              12
##                   secretary of the commonwealth
##                                               1
##                                          senator
##                                              41
##                                          speaker
##                                              23
##                                  speaker pro tem
##                                               2
##                                    state auditor
##                                               8
##                                state comptroller
##                                               4
##                                 state controller
##                                               2
##                                  state treasurer
##                                              13
##            state treasurer and receiver-general
##                                               1
##            superintendent of public instruction
##                                               4
##                       U.S. House Democratic whip
##                                               1
##                       U.S. House majority leader
##                                               1
##                               U.S. House speaker
##                                               1
##                      U.S. Senate Democratic whip
##                                               1
##                      U.S. Senate minority leader
##                                               1
```

```r
df_regroup <- df %>%
  mutate(position_redux = case_when(
    position %in% c("past president",
                    "past vice president")  ~ "Former government",
    position == "governor" ~ "Governor",
    position %in% c("senator", "representative") ~ "Member of Congress",
    T ~ "Other"
  ))
table(df_regroup$position_redux)
```

```
##
##  Former government           Governor Member of Congress
##                  5                 28                268
##              Other
##                704
```

Putting it all together.

Ok, it is super tedious to always create a new data frame. The power of piping is that we can chain multiple operations together. This makes our live easier and the code a lot easier to read. You can also use comments to remind yourself what you are doing and inform others.

```
df_new <- raw %>%

  # Renaming endorser_party variable
  # selecting remainding variables minus source variable
  dplyr::select(endorse_date = date,
                endorser_party = endorser.party,
                everything(),
                -source) %>%

  # Creating Super Tuesday variable
  mutate(super = ifelse(state %in% super_tuesday, 1, 0)) %>%

  # recoding drop variable
  mutate(drop = replace(drop, endorsee == "Cory Booker", 1)) %>%

  # Coding alternative position variable
  mutate(position_redux = case_when(
    position %in% c("past president",
                    "past vice president")  ~ "Former government",
    position == "governor" ~ "Governor",
    position %in% c("senator", "representative") ~ "Member of Congress",
    T ~ "Other"
  ))
```

Introducting `summarize()`

One of the most powerful `dplyr` features is the `summarize()` function,[1] especially in combination with `group_by()`.

Let's compute the average and total number of endorsement points that each candidate got. Note that if the `points` variable had missing values, we would need to specify `na.rm = TRUE` within the `mean()` and `sum()` functions.

[1]Note that both the British and American English versions of `summarise()` and `summarize()` work!

```r
by_endorsee <- df_new %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T))
```

What is going on with that last row that shows up as `NA` for the `endorsee` variable? Remember that there are a number of observations that are `NA` because no endorsement has been made (as of 13 January 2020). Therefore, let's first drop the values that are `NA` for the `endorsee` variable before creating the summarized data frame.[2]

```r
by_endorsee <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T))
```

Suppose, instead of looking at the points, I wanted to know how many endorsements each candidate has. We can use the `n()` function to get a count.

```r
by_endorsee <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T),
            count_endorsement = n()) #counts the number of observations
```

Grouping by more than one variable

Do endorsements vary between Super Tuesday states and the rest? Let's find out! To do so, we can simply group the data by `endorsee` and by the variable `super` that we created.

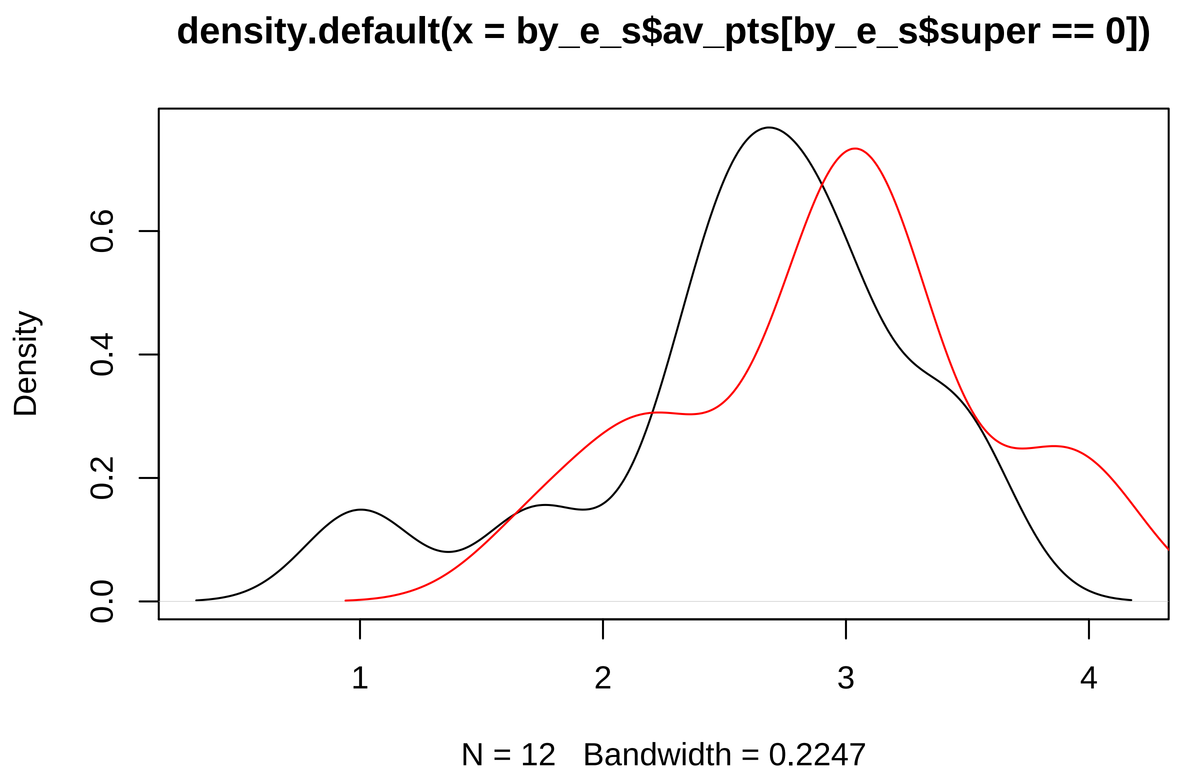
```r
by_e_s <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee, super) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T),
            count_endorsement = n()) #counts the number of observations
```

Hm, this is pretty hard to see. Let's visualize it.[3]

```r
{plot(density(by_e_s$av_pts[by_e_s$super == 0]))
lines(density(by_e_s$av_pts[by_e_s$super == 1]),
      col = "red")}
```

[2]Of course, we could also drop this variable retroactively.

[3]This would be a lot easier using `ggplot2` by the way. We will cover data visualization with `ggplot2` in the next workshop. Stay tuned!



Quick note on `group_by()` + `summarize()` vs. `group_by()` + `mutate()`

Above, we create grouped data summaries. Notice that only the variables we explicitly specify are retained when first grouping the data and then running `summarize()`.

Sometimes, we want to create grouped variables but retain the remainder of the dataframe. To do so , we can also use `group_by()` with `mutate()`. However, this is not appropriate in all situations. For example, if we created a `count_endorsement` variable for each row, someone looking at our dataframe could interpret this as each candidate having received that many endorsements from a particular endorsee.[4]

```
pleasedont <- df_new %>%
  filter(!is.na(endorsee)) %>%

  group_by(endorsee) %>%
  mutate(count_endorsement = n())
```

Sorting with `arrange()`

To get an idea of where each candidate stands, let's reorder the output using `arrange()`.

```
by_endorsee <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T),
            count_endorsement = n()) %>%

  # ordering output
  arrange(count_endorsement)
```

[4]Not the best example, but you get the point. . .

To re-order in descending order use `-` before the variable you want the data frame to be ordered by or `desc()`. Let's find out who is the candidate with the most average endorsement points

```
by_endorsee <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T),
            count_endorsement = n()) %>%

  # ordering output
  arrange(desc(av_pts)) # - works too but I find it confusing
```

Among the endorsees, Joe Biden received the most endorsements and has the highest number of "total endorsement points" per the scale of each endorsement's relative value generated by Five Thirty Eight. However, the candidate with the highest value for "average endorsement points" is Michael Bennet, followed by Amy Klobuchar.

Tie breakers

A number of candidates have the same number of endorsements. We can add a tie breaker by specifying more than one variable inside `arrange()`.

```
by_endorsee <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T),
            count_endorsement = n()) %>%

  # ordering output
  arrange(desc(count_endorsement), desc(av_pts))
```

Selecting rows using `slice()`

Suppose, we only wanted to know who the top 10 candidates were in terms of endorsements. We can use the `slice()` function for this.

```
by_endorsee_top10 <- df_new %>%
  filter(!is.na(endorsee)) %>%
  group_by(endorsee) %>%
  summarize(av_pts = mean(points, na.rm = T),
            tot_pts = sum(points, na.rm = T),
            count_endorsement = n()) %>%

  # ordering output
  arrange(desc(count_endorsement), desc(av_pts)) %>%

  # only the top ten
  slice(1:10)
```

Saving files

Finally, lets save the output of our code. This time, we would like to save it in the .dta format. Remember to load the `foreign()` library before saving.

```
# write.dta(df_new, "highest_endorsements.dta")
```

Working with dates in `lubridate`

Below, I give you a very *brief* primer on the `lubriate` package that makes working with dates much easier.

```
#install.packages("lubridate")
library(lubridate)
```

```
##
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':
##
##     date
```

Lubridate can consolidate different date formats. Take the following as an example. `ymd` stands for "the input is of the following order: year, month, day."

```
dates <- c("2020-01-17", "2020.01.17", "2020-1-17")
ymd(dates)
```

```
## [1] "2020-01-17" "2020-01-17" "2020-01-17"
```

Of course, lubridate understands other formats as well and translates them into a common `ymd` format.

```
dates2 <- c("01-17-2020", "01.17.2020", "1-17-2020")
mdy(dates2)
```

```
## [1] "2020-01-17" "2020-01-17" "2020-01-17"
```

In our dataframe, we have a column for when a candidate dropped out of the race. Lets translate it into a lubridate date variable (even though it technically is already cleaned; for learning purposes). The items that failed to parse are `NA` values.

```
table(df_new$drop_date)
```

```
##
## 2019-08-15 2019-08-21 2019-11-01 2019-12-03 2020-01-02 2020-01-13
##          1          4          1         32          4         22
```

```
head(table(df_new$endorse_date))
```

```
##
## 01.01.20 01.02.19 01.03.19 01.08.19 01.09.19 02.01.19
##        1        2        2        2        1        1
```

```
df_new <- df_new %>%
  mutate(endorse_date = dmy(endorse_date),
         drop_date = ymd(drop_date))
```

Now we can extract other variables from the date.

```
df_new <- df_new %>%
  mutate(endorse_year = year(endorse_date),
         endorse_month = month(endorse_date))
```

Lets compute the endorsements per month that candidates dropped out of the race.

```r
df_drop_endorse <- df_new %>%
  filter(drop == 1) %>%
  group_by(endorse_year, endorse_month) %>%
  summarise(count = n())
```

We can also compute the duration in days between the last endorsement and the candidate dropping out of the race using the `interval()` function. Notice the use of `ungroup()` here.

```r
df_drop_duration <- df_new %>%
  filter(drop == 1) %>%
  group_by(endorsee) %>%
  filter(endorse_date == max(endorse_date)) %>%
  ungroup() %>%
  mutate(dur = interval(endorse_date, drop_date),
         dur_days = dur/ddays(1)) %>%
  arrange(desc(dur_days))
```

Exercise How long have candidates that didn't drop out not received an endorsement today (17 January 2020)?

```
## # A tibble: 9 x 2
##   endorsee          dur_days
##   <chr>                <dbl>
## 1 John Delaney           324
## 2 Michael Bennet          40
## 3 Amy Klobuchar           31
## 4 Tom Steyer              14
## 5 Elizabeth Warren         3
## 6 Joe Biden                3
## 7 Pete Buttigieg           2
## 8 Bernie Sanders           1
## 9 Michael Bloomberg        1
```

Exercise How many days are you old? `lubridate` can tell you!

```
## [1] 12051
```

What day of the week were you born?

```r
wday(dob, label = T)
```

```
## [1] Mon
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Merging data

The data

In `R`, the package `haven` allows us to access read and write various data formats. Currently, it supports Stata, SPSS and SAS files. Value labels from imported data are translated into a new `labelled()` class, which preserves the original semantics. In addition to some of the functions that can be used to explore the dimensions of any data frame, we can also explore the substantive descriptions of variables.

For this exercise, we will be working with data from the Pew Research Center's *American Trends Panel* (ATP), which is a nationally representative panel of randomly selected U.S. adults. First, we will load the data from the 38th and 39th wave of the panel. The data has been cleaned after downloading from the original sources and subsetted to only include U.S. citizens and respondents who participated in both surveys.

The topics covered in each wave are as follows:

- Wave 38: Pre-election poll, generations research
- Wave 39: Post-election poll

The Wave 38 data contains the following variables:

- `POL1DT_W38`: Do you approve or disapprove of the way Donald Trump is handling his job as president?
- `VTPLAN`: Do you plan to vote in the elections this November?
- `F_PARTY_FINAL`: Party affiliation

The Wave 39 data contains the following variables:

- `POL1DT_W39`: Do you approve or disapprove of the way Donald Trump is handling his job as president?
- `VTHPPYUS_W39`: All in all, are you happy or unhappy with the results of the recent elections that took place across the United States?
- `VOTED_ATP`: Which of the following statements best describes you?
- `F_PARTY_FINAL`: Party affiliation

First, let's load the data from both waves.

```r
library(haven)
# load datasets
w38.df <- haven::read_dta("ATP_W38.dta")
w39.df <- haven::read_dta("ATP_W39.dta")
```

Let us take a look at the data sets. Notice that we can see the question below the variable name.

```r
# View(w38.df)
```

Where does this information come from? We can use the `attributes()` function to get a better idea.[5]

```r
attributes(w38.df$VTPLAN_W38)
```

```
## $label
## [1] "VTPLAN. Do you plan to vote in the elections this November?"
##
## $format.stata
## [1] "%10.0g"
##
## $class
## [1] "haven_labelled"
##
## $labels
##     Yes, I plan to vote ON Election Day
##                                       1
## Yes, I plan to vote BEFORE Election Day
##                                       2
##                  No, I do not plan to vote
##                                       3
##                                Not sure
##                                       4
##                                 Refused
##                                      99
```

```r
attributes(w39.df$VOTED_ATP_W39)
```

```
## $label
```

[5]The "%10.0g" is the state encoding for different variable types, see https://www.stata.com/manuals13/dformat.pdf.

```
## [1] "VOTED_ATP. Which of the following statements best describes you?"
##
## $format.stata
## [1] "%10.0g"
##
## $class
## [1] "haven_labelled"
##
## $labels
##     I did not vote in the 2018 congressional elections
##                                                       1
##                   I planned to vote but wasn't able to
##                                                       2
## I definitely voted in the 2018 congressional elections
##                                                       3
##                                                 Refused
##                                                      99
```

```
# attributes(w38.df$F_PARTY_FINAL)
# attributes(w38.df$F_IDEO)
```

Exercise 1 Suppose we would like to investigate differences between Democrats and Republicans in the Wave 39 data. Add a new variable called `party_cat` that is coded `dem` if the respondent is affiliated with the Democrats, `rep` if the respondent is affiliated with the Republicans, `ind` if they are independent, and `other` for the remaining cases. Check that you get the same numbers as below by using the `table()` function.

Bonus points if you figure out how to express this is percentages of the total number of respondents!

```
##
##   dem   ind other   rep
##  3923  2547  1059  2844
```

```
##
##      dem      ind    other      rep
## 37.81934 24.55413 10.20920 27.41733
```

Using `dplyr` for merging

Luckily, the `dplyr` package in `R` contains a number of functions that allow us to merge data in "smarter" ways.

Suppose, we have two data frames: `x` (here w38.df) and `y` (here w39.df). The basic syntax for data merging with `dplyr` is the following:

`output <- whichjoin(A, B, by = "variable")`

We will focus on the following four join functions:

- `full_join()`: Join data from `x` and `y` upon retaining all rows and values. This is the maximum join possible. Neither `x` nor `y` is the "master."
- `inner_join()`: Join only those rows that appear in both `x` and `y`. Neither `x` nor `y` is the "master."
- `left_join()`: Join only those rows from `y` that appear in `x`, retaining all data in `x`. Here, `x` is the "master."
- `right_join()`: Join only those rows from `x` that appear in `y`, retaining all data in `y`. Here, `y` is the "master."

`full_join()`

Suppose we wanted to keep the maximum amount of data in our new merged data set. We would use the `full_join()` function to merge the Wave 38 and Wave 39 data.

WARNING Do not try to run this on RStudio! The data frame is too big and you will crash your session!!!

```
# full_wrong <- full_join(w38.df, w39.df, by = "F_PARTY_FINAL")
# nrow(full_wrong)
```

A couple of things to notice:

1. There are a number of duplicate column names in our data. `dplyr` automatically added a suffix to those instances; x for the Wave 38 data (the first data set we passed to the function) and y for the Wave 39 data (the second data set passed to the function). We will rename these variables later on to keep track from which data set they originate.
2. We have over 30 million observations even though our original datasets only have approximately 10,000 observations each. Why do you think this is the case? Do you notice something weird about the `QKEY` variable?

The reason for the high number of observations is that we asked `dplyr` to join the data sets based on the `F_PARTY_FINAL` variable only. This matches each observation in dataset x to each observation in dataset y as long as they have the same party affiliation. However, the rows in our data are not uniquely identified by the `F_PARTY_FINAL` variable. Since we are working with panel data sets, each observation is uniquely identified by the variable `QKEY` to track which respondents participated in multiple waves. We need to pass this information to the `full_join()` function to correctly match up the observations in our data.

`full_join()` and the other join functions from dplyr can identify which columns to match on based on duplicate column names. However, in our particular case, we run into problems because we have more duplicate column names that unique identifiers for the data; see the message that R outputs when running the command without specifying by which variables to merge. Ideology and/or party affiliation per respondent could have changed across the two waves! We do not want to join based on them!

```
full_alsowrong <- full_join(w38.df, w39.df)
```

```
## Joining, by = c("QKEY", "F_PARTY_FINAL", "F_IDEO")
```

```
nrow(full_alsowrong)
```

```
## [1] 11508
```

Before we do the correct `full_join()`, let's change the name of variables that are not unique identifiers in each dataset. To do this, we can use the `select()` function.

```
names(w38.df)
```

```
## [1] "QKEY"          "POL1DT_W38"    "VTPLAN_W38"    "F_PARTY_FINAL"
## [5] "F_IDEO"
```

```
w38.df <- w38.df %>%
  dplyr::select(QKEY:VTPLAN_W38,
                F_PARTY_FINAL_W38 = F_PARTY_FINAL,
                F_IDEO_W38 = F_IDEO)
names(w38.df)
```

```
## [1] "QKEY"              "POL1DT_W38"        "VTPLAN_W38"
## [4] "F_PARTY_FINAL_W38" "F_IDEO_W38"
```

Notice that below, we silently drop the `party_cat` variable by not including it within out `select()` statement.

```
w39.df <- w39.df %>%
  dplyr::select(QKEY:VOTED_ATP_W39,
                F_PARTY_FINAL_W39 = F_PARTY_FINAL,
                F_IDEO_W39 = F_IDEO)
names(w39.df)
```

```
## [1] "QKEY"              "POL1DT_W39"        "VTHPPYUS_W39"
## [4] "VOTED_ATP_W39"     "F_PARTY_FINAL_W39" "F_IDEO_W39"
```

Now we are ready to join!

```
merged_full <- full_join(w38.df, w39.df, by = "QKEY")
names(merged_full)
```

```
##  [1] "QKEY"              "POL1DT_W38"        "VTPLAN_W38"
##  [4] "F_PARTY_FINAL_W38" "F_IDEO_W38"        "POL1DT_W39"
##  [7] "VTHPPYUS_W39"      "VOTED_ATP_W39"     "F_PARTY_FINAL_W39"
## [10] "F_IDEO_W39"
```

```
nrow(merged_full)
```

```
## [1] 11487
```

`inner_join()`

Suppose, instead of retaining as much data as possible, we only wanted to keep data for which we have *observations* in both data sets. Note that this merging is only done based on the variables that we specify to uniquely identify each observation. Please refer to the *Data Wrangling Cheat Sheet* for methods that will select based on matching values beyond the unique identifiers (see for example `intersect()` and `setdiff()`).

```
merged_inner <- inner_join(w38.df, w39.df, by = "QKEY")
head(merged_inner)
```

```
## # A tibble: 6 x 10
##     QKEY POL1DT_W38 VTPLAN_W38 F_PARTY_FINAL_W~ F_IDEO_W38 POL1DT_W39
##    <dbl>  <dbl+lbl>  <dbl+lbl>        <dbl+lbl>  <dbl+lbl>  <dbl+lbl>
## 1 100197 2 [Disapp~ 2 [Yes, I~   2 [Democrat]   4 [Libera~ 2 [Disapp~
## 2 100260 1 [Approv~ 2 [Yes, I~   1 [Republican] 1 [Very c~ 1 [Approv~
## 3 100314 1 [Approv~ 1 [Yes, I~   1 [Republican] 3 [Modera~ 1 [Approv~
## 4 100363 1 [Approv~ 1 [Yes, I~   2 [Democrat]   4 [Libera~ 1 [Approv~
## 5 100446 2 [Disapp~ 1 [Yes, I~   2 [Democrat]   4 [Libera~ 2 [Disapp~
## 6 100588 2 [Disapp~ 1 [Yes, I~   2 [Democrat]   3 [Modera~ 2 [Disapp~
## # ... with 4 more variables: VTHPPYUS_W39 <dbl+lbl>,
## #   VOTED_ATP_W39 <dbl+lbl>, F_PARTY_FINAL_W39 <dbl+lbl>,
## #   F_IDEO_W39 <dbl+lbl>
```

```
nrow(merged_inner)
```

```
## [1] 9298
```

`left_join()` and `right_join()`

Suppose we had an existing master data set and wanted to add data to this master without adding new rows—just new columns. This could for example be the case if we had a specific time frame that we wanted to study, and only want to merge data which matches this time frame.

Here, suppose we wanted to make the Wave 38 data our master data set and merge data from the Wave 39 data onto this master. The Wave 39 data took place after the 2018 midterm elections, while Wave 38 contains pre-election data. We are interested in keeping all of the observations from pre-election, and add on data for pre-election respondents who also participated in Wave 39. We can show that the Wave 38 data and the merged data frame have the same number of observations (rows).

```
merged_left <- left_join(w38.df, w39.df, by = c("QKEY"))
nrow(w38.df) == nrow(merged_left)
```

```
## [1] TRUE
```

Exercise 2 How would you achieve the exact same result using the `right_join()` function?

```
## [1] TRUE
```

Data re-shaping with `tidyr`

Another important task in data management is data re-shaping. Often, data does not come in the format that we need for data merging, data visualization, statistical analysis, or vectorized programming. In general, we want data in the following format:

1. Each variable forms a column.
2. Each observation forms a row[6].
3. For panel data, the unit (e.g. country) and time (e.g. year) identifier form columns.

The `tidyr` package offers two main functions for data reshaping:

- `pivot_longer()`: Shaping data from wide to long.
- `pivot_wider()`: Shaping data from long to wide.

Wide versus long data

For **wide** data formats, each unit's responses are in a single row. For example:

Country	Area	Pop1990	Pop1991
A	300	56	58
B	150	40	45

For **long** data formats, each row denotes the observation of a unit at a given point in time. For example:

Country	Year	Area	Pop
A	1990	300	56
A	1991	300	58
B	1990	150	40
B	1991	150	45

`pivot_longer()`

We use the `pivot_longer()` function to reshape data from wide to long. In general, the syntax of the data is as follows:

`new_df <- pivot_longer(old_df, columns to transform, names_to = "name", values_to = "value")`[7]

[6]Hadley Wickham (2014, "Tidy Data" in *Journal of Statistical Analysis*) adds another condition, namely that "Each type of observational unit forms a table." We will not go into this here, but I can highly recommend you read Wickham's piece if you want to dive deeper into tidying data.

[7]If we use the function in a pipe, we do not need to specify the `old_df` parameter. `tidyr` automatically knows to use the

We will be working with the `merged_inner` data set. Suppose instead of having a separate column for the mutual variables for each wave, we wanted to create a new column called `wave` that codes the wave of the survey.

Let's start with a simple example and first only work with a subset of the `merged_inner` data frame that contains the `QKEY`, `POL1DT_W38`, and `POL1DT_W39` variables. Below, note that since `tidyr` and `dplyr` are sibling packages from the "tidyverse," we can use them seamlessly in the same pipe (here using `group_by()`, `summarize()`, `mutate()`, and `filter()` from `dplyr`).

```
library(tidyr)
names(merged_inner)
```

```
##  [1] "QKEY"              "POL1DT_W38"        "VTPLAN_W38"
##  [4] "F_PARTY_FINAL_W38" "F_IDEO_W38"        "POL1DT_W39"
##  [7] "VTHPPYUS_W39"      "VOTED_ATP_W39"     "F_PARTY_FINAL_W39"
## [10] "F_IDEO_W39"
```

```
df_long_small <- merged_inner %>%

  dplyr::select(QKEY,
                contains("POL1DT")) %>%

  pivot_longer(names_to = "wave",
               values_to = "trump_approval",
               cols = -QKEY) # Everything except the QKEY

# We could have also written cols = c("POL1DT_W38", "POL1DT_W39")

head(df_long_small)
```

```
## # A tibble: 6 x 3
##     QKEY wave       trump_approval
##    <dbl> <chr>           <dbl+lbl>
## 1 100197 POL1DT_W38 2 [Disapprove]
## 2 100197 POL1DT_W39 2 [Disapprove]
## 3 100260 POL1DT_W38 1 [Approve]
## 4 100260 POL1DT_W39 1 [Approve]
## 5 100314 POL1DT_W38 1 [Approve]
## 6 100314 POL1DT_W39 1 [Approve]
```

Ok, now that we know how this works, let's try this with all variables. Unfortunately, `R` throws an error that essentially complains because the factor levels (i.e. `haven` labels) don't have the same values across variables. This is a know issue (see https://github.com/tidyverse/haven/issues/477) that will hopefully be fixed by the `haven` developing team soon.

```
# df_long_all <- merged_inner %>%
#
#   pivot_longer(names_to = "variable",
#                values_to = "value",
#                cols = -QKEY)
```

For now, a workaround would be to change the variable types to factor using `haven`'s `as_factor()` function. Note that you can also use the `as_factor()` function on a single variable, rather than on an entire dataframe.

```
df_long_all <- as_factor(merged_inner) %>%
```

lastest version of the data frame in the pipe.

```r
  pivot_longer(names_to = "variable",
               values_to = "value",
               cols = -QKEY)
```

Ok, that is cool, but it doesn't really help us all that much because the original variable names aren't terribly informative. Below, we try this again, selecting only the variables that are common across both waves. First, we rename a few variables to make our life easier later on.[8]

```r
names(merged_inner)
```

```
##  [1] "QKEY"              "POL1DT_W38"        "VTPLAN_W38"       
##  [4] "F_PARTY_FINAL_W38" "F_IDEO_W38"        "POL1DT_W39"       
##  [7] "VTHPPYUS_W39"      "VOTED_ATP_W39"     "F_PARTY_FINAL_W39"
## [10] "F_IDEO_W39"
```

```r
df_long <- as_factor(merged_inner) %>%

  dplyr::select(QKEY,
                trump_w38 = POL1DT_W38,
                trump_w39 = POL1DT_W39,
                ideo_w38 = F_IDEO_W38,
                ideo_w39 = F_IDEO_W39,
                party_w38 = F_PARTY_FINAL_W38,
                party_w39 = F_PARTY_FINAL_W39) %>%

  pivot_longer(names_to = "variable",
               values_to = "value",
               cols = contains("trump"))

## Please inspect the output to reason about what happened above!

# Lets select all variables using a combination of multiple contains.
# Note that this is for instructional purposes only,
# in reality we would probably again use -QKEY
df_long <- as_factor(merged_inner) %>%

  dplyr::select(QKEY,
                trump_w38 = POL1DT_W38,
                trump_w39 = POL1DT_W39,
                ideo_w38 = F_IDEO_W38,
                ideo_w39 = F_IDEO_W39,
                party_w38 = F_PARTY_FINAL_W38,
                party_w39 = F_PARTY_FINAL_W39) %>%

  pivot_longer(names_to = "variable",
               values_to = "value",
               cols = c(contains("trump"),
                        contains("ideo"),
                        contains("par")))
```

Introducing `separate()`

Why did renaming the variables make our life easier? Well, the `separate()` function is awsome for data wrangling tasks, because it allows us to separate the `variable` column into two, that way we have a separate

[8]Once you learn regular expressions and string operations in a later session, this will be easier.

time-related column.

```
df_long <- as_factor(merged_inner) %>%

  dplyr::select(QKEY,
                trump_38 = POL1DT_W38,
                trump_39 = POL1DT_W39,
                ideo_38 = F_IDEO_W38,
                ideo_39 = F_IDEO_W39,
                party_38 = F_PARTY_FINAL_W38,
                party_39 = F_PARTY_FINAL_W39) %>%

  pivot_longer(names_to = "variable",
               values_to = "value",
               cols = -QKEY) %>%

  separate(variable, c("var", "wave"))
```

It would be interesting to find out to what extent people changed their stated values after the election? Let's drop the people who refused to answer. Also, we need to change the `wave` variable to numeric. We then use the `lag()` function to create a lagged variable from `value`.

```
df_change <- df_long %>%
  mutate(wave = as.numeric(wave)) %>%
  group_by(QKEY, var) %>%
  mutate(value_lag = lag(value)) %>%
  ungroup() %>%
  mutate(change = ifelse(value != value_lag, 1, 0))

# Only 0.06% of answers changed
sum(df_change$change, na.rm = T)/length(unique(df_change$QKEY))
```

```
## [1] 0.06065821
```

`pivot_wider()`

Suppose we wanted to revert our operation (or generall shape data from a long to a wide format), we can use `tidyr`'s `pivot_wider()` function. The syntax is similar to `pivot_longer()`.

`new_df <- spread(old_df, names_from = key, values_from = value)`,

where `key` refers to the colum which contains the values that are to be converted to column names and `value` specifies the column that contains the values which is to be stored in the newly created columns.

Below, we semi-revert the creation of a single column containing the variable types and a single column containing our variable values.

```
names(df_long)
```

```
## [1] "QKEY"  "var"   "wave"  "value"
```

```
df_wide <- df_long %>%
  pivot_wider(names_from = var,
              values_from = value) %>%

  #Cleaning issues w/ levels
  droplevels() # drops empty factor levels
```

We can use this format to compute comparisons of Trump approval across waves. Notice the use of `ungroup()` below.

```r
df_approval1 <- df_wide %>%
  filter(trump != "Refused") %>%
  group_by(wave, trump) %>%
  summarise(count = n()) %>%

  # Creating percentage
  ungroup() %>%
  group_by(wave) %>%
  mutate(perc = 100/sum(count)*count)
df_approval1
```

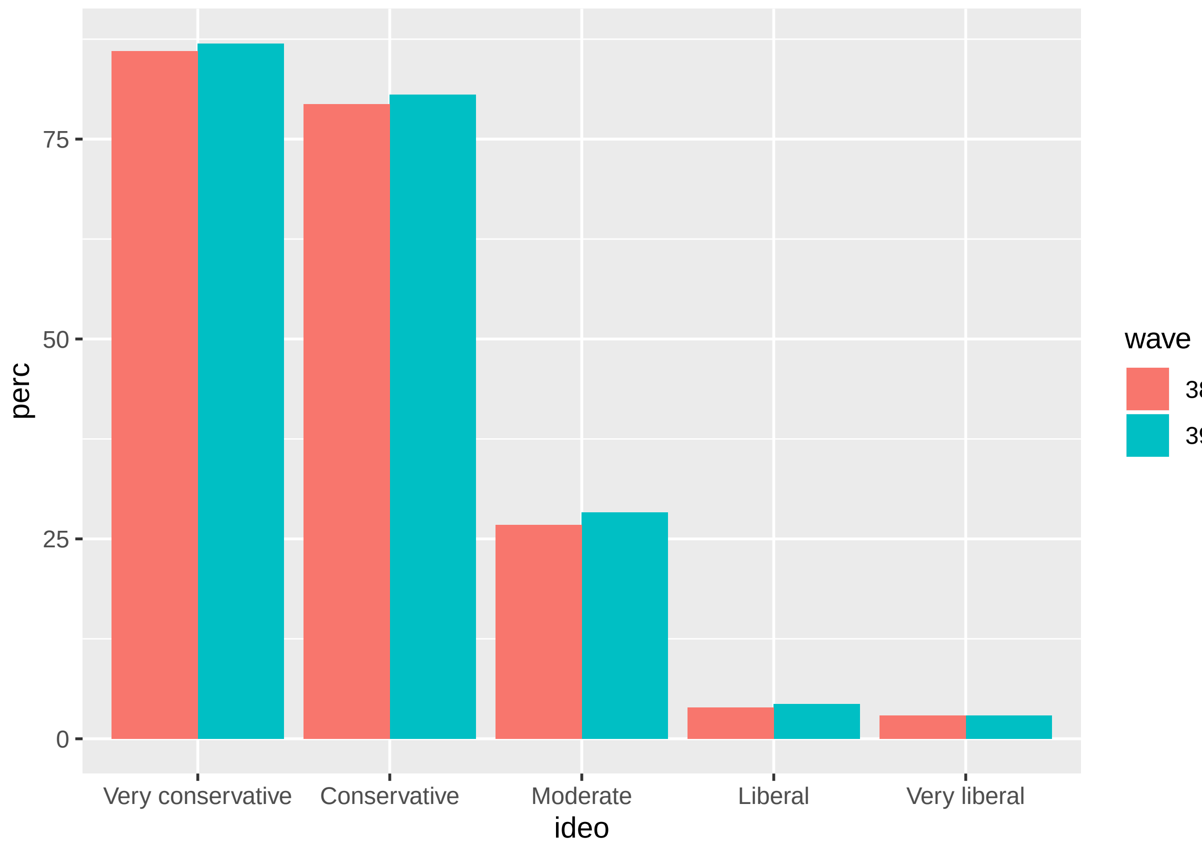
```
## # A tibble: 4 x 4
## # Groups:   wave [2]
##   wave  trump      count  perc
##   <chr> <fct>      <int> <dbl>
## 1 38    Approve     3310  36.1
## 2 38    Disapprove  5871  63.9
## 3 39    Approve     3423  37.1
## 4 39    Disapprove  5797  62.9
```

Bonus We can also use this format to compare pre- and post-election approval rates about Donald Trump by political ideology among the panel respondents (Note: without applying survey weights, this is *not* a representative estimate). Approval ratings have gone up across the ideological spectrum.

```r
df_approval2 <- df_wide %>%
  filter(trump != "Refused",
         ideo != "Refused") %>%
  group_by(wave, trump, ideo) %>%
  summarise(count = n()) %>%

  # Creating percentage
  # Here, the baseline is all people in a certain ideology group per wave
  ungroup() %>%
  group_by(wave, ideo) %>%
  mutate(perc = 100/sum(count)*count) %>%
  filter(trump == "Approve")

library(ggplot2)
ggplot(df_approval2,
       aes(x = ideo, y = perc, fill = wave)) +
  geom_bar(stat = "identity", position = position_dodge())
```



We can also fully revert the operation by specifying both the `wave` and the the `value` colum in the `names_from` parameter.

```r
names(df_long)
```

```
## [1] "QKEY"  "var"   "wave"  "value"
```

```r
df_wider <- df_long %>%
  pivot_wider(names_from = c(var, wave),
              values_from = value)
```

Sources

Bacoffe, A. and Silver, N., 2019. *The 2020 Endorsement Primary*. FiveThirtyEight. Retrieved January 13, 2020.

Pew Research Center. 2015. Building Pew Research Center's American Trends Panel, Technical Report, Washington D.C. Available at http://pewrsr.ch/1Jo4nKE.