Methods Camp

UT Austin, Department of Government

Andrés Cruz Meiying Xu

August 2024

Table of contents

Class schedule

WIP

These are materials for the ongoing version of Methods Camp (2024). Click here to see last year's materials.

Date	Time	Location
Fri, Aug. 16	9:00 AM - 4:00 PM	RLP 2.606
Sat, Aug. 17	No class	-
Sun, Aug. 18	No class	-
Mon, Aug. 19	9:00 AM - 4:00 PM	BAT 5.108
Tue, Aug. 20	9:00 AM - 4:00 PM	RLP 2.606
Wed, Aug. 21	9:00 AM - 4:00 PM	BAT 5.108
Thu, Aug. 22	9:00 AM - 4:00 PM	RLP 2.606

On class days, we will have a lunch break from 12:00-1:00 PM. We'll also take short breaks periodically during the morning and afternoon sessions as needed.

Description

Welcome to Introduction to Methods for Political Science, aka "Methods Camp"! Methods Camp is designed to give everyone a chance to brush up on some skills in preparation for the introductory Statistics and Formal Theory courses. The other goal of Methods Camp is to allow you to get to know your cohort. We hope that matrix algebra and the chain rule will still prove to be good bonding exercises!

As you can see from the above schedule, we'll be meeting on Friday, August 16th as well as from Monday, August 19th through Thursday, August 22nd. Classes at UT begin the start of the following week on Monday, August 26th. Below is a tentative schedule outlining what will be covered in the class, although we may rearrange things if we find we're going too slowly or too quickly through the material.

Course outline

1 Friday morning: Intro to R

- Introductions
- R and RStudio: basics
- Objects (vectors, matrices, data frames, etc.)
- Basic functions (`mean()`, `length()`, etc.)
- Packages: installation and loading (including the tidyverse)

2 Friday afternoon: Tidy data analysis I

- Tidy data
- Data wrangling with dplyr
- Data visualization basics with `ggplot2`

3 Monday morning: Functions

- Definitions
- Functions in R
- Common types of functions
- Logarithms and exponents
- Composite functions

4 Monday afternoon: Calculus

- Derivatives
- Optimization
- Integrals

5 Tuesday morning: Matrices

- Matrices
- Systems of linear equations
- Matrix operations (multiplication, transpose, inverse, determinant)
- Solving systems of linear equations in matrix form (and why that's cool)
- Introduction to OLS

6 Tuesday afternoon: Tidy data analysis II

- Loading data in different formats (.csv, R, Excel, Stata, SPSS)
- Recoding values (`if_else()`, `case_when()`)
- Handling missing values
- Pivoting data
- Merging data

- Plotting extensions (trend graphs, facets, customization)

7 Wednesday morning: Probability

- Probability: basic concepts
- Random variables, probability distributions, and their properties
- Common probability distributions

8 Wednesday afternoon: Statistics and simulations

- Statistics: basic concepts
- Random sampling and loops in R
- Simulation example: bootstrapping

9 Thursday morning: Coding with AI

- Visualization tools
- Statistical testing and simulation
- Text analysis examples

10 Thursday afternoon: Wrap-up

- Project management fundamentals
- Self-study resources and materials
- Other software (Overleaf, Zotero, etc.)
- Methods resources at UT

Contact info

If you have any questions during or outside of methods camp, you can contact us via email. Or if you are curious about our research, you can also check out our respective websites and Twitter accounts (or should we say X…):

- Andrés Cruz: andres.cruz@utexas.edu [Website] [Twitter]
- Meiying Xu: xu.meiying@utexas.edu [Website] [Twitter]

Acknowledgements

We thank previous Methods Camp instructors for their accumulated experience and materials, which we have based ours upon. UT Gov Prof. Max Goplerud gave us amazing feedback for this iteration of Methods Camp (2024). All errors remain our own (and will hopefully be fixed with your help!).

Materials from previous editions

- 2023: co-taught by Andrés Cruz and Matt Martin.

Setup

Installing R and RStudio

R is a programming language optimized for statistics and data analysis. Most people use R from RStudio, a graphical user interface (GUI) that includes a file pane, a graphics pane, and other goodies. Both R and RStudio are open source, i.e., free as in beer and free as in freedom!

Your first steps should be to install R and RStudio, in that order (if you have installed these programs before, make sure that your versions are up-to-date—if they are not, simply follow the instructions below to re-install them):

1. Download and install R from the official website, CRAN. Click on "Download R for <Windows/MacOS>" and follow the instructions. If you have a Mac, make sure to select the version appropriate for your system (Apple Silicon for newer M1/M2/M3 Macs and Intel for older Macs).
2. Download and install RStudio from the official website. Scroll down and select the installer for your operating system (most likely the .exe for Windows 10/11 or the .dmg for macOS 12+).

After these two steps, you can open RStudio in your system, as you would with any program. You should see something like this:

> **Note for Windows users**
>
> While the installation steps above should be enough for most tasks, we also suggest that Windows users install RTools (click on the "Rtools44 installer" link at the middle of the package to get the .exe file). Rtools is needed on Windows to install some advanced packages, so it is a good idea to have it on your system.

That's it for the installation! We also *strongly* recommend that you change a couple of RStudio's default settings.[1] You can change settings by clicking on `Tools > Global Options` in the menubar. Here are our recommendations:

[1] The idea behind these settings (or at least the first two) is to force R to start from scratch with each new session. No lingering objects from previous coding sessions avoids misunderstandings and helps with reproducibility!

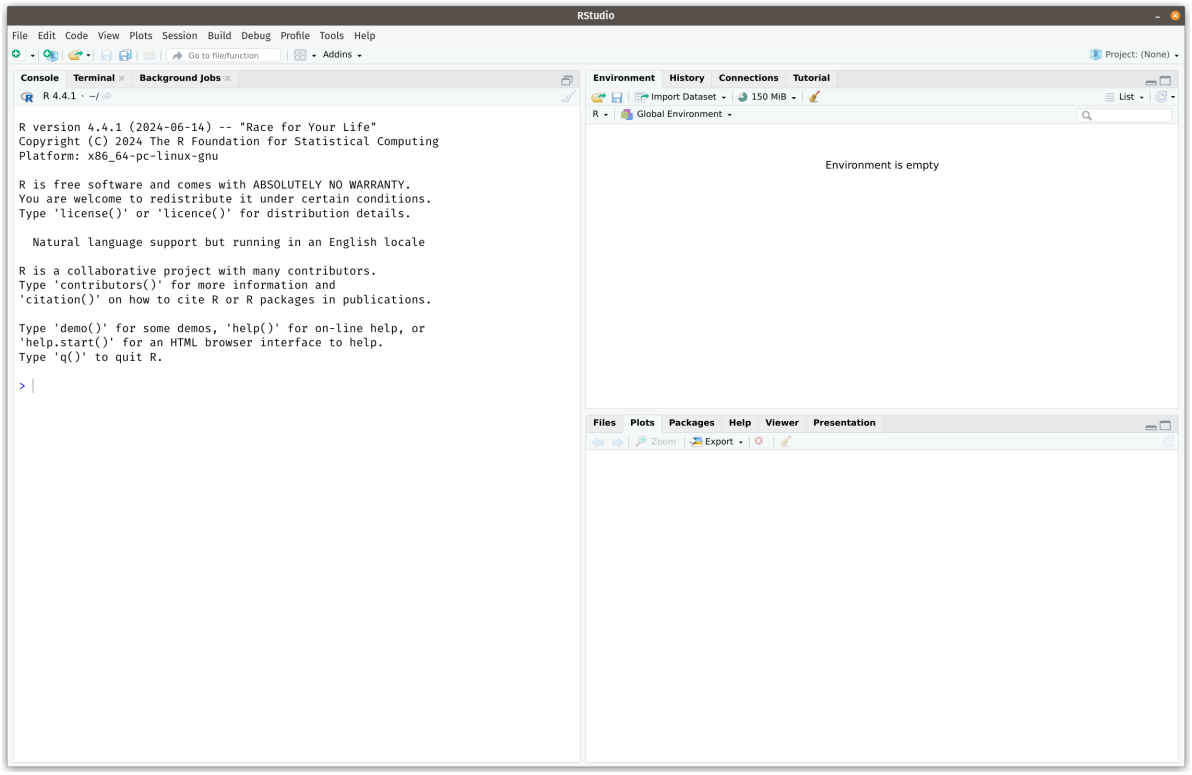


Figure 1: How RStudio looks after a clean installation.

- `General > Uncheck "Restore .RData into workspace at startup"`
- `General > Save workspace to .RData on Exit > Select "Never"`
- `Code > Check "Use native pipe operator"`
- `Tools > Global Options > Appearance` to change to a dark theme, if you want! Pros: better for night sessions, hacker vibes...

Setting up for Methods Camp

All materials for Methods Camp are both on this website and available as RStudio projects for you to execute locally. An RStudio project is simply a folder where one keeps scripts, datasets, and other files needed for a data analysis project.

Below are RStudio projects for you to download, available as .zip compressed files. On MacOS, the file will be uncompressed automatically. On Windows, you should do `Right click > Extract all`.

- Download Part 1 of the class materials.
- (Additional projects will be available in the following days).

> **Warning**
>
> Make sure to properly unzip the materials. Double-clicking the .zip file on most Windows systems *will not* unzip the folder—you must do `Right click > Extract all`.

You should now have a folder called `methodscamp_part1/` on your computer. Navigate to the `methodscamp_part1.Rproj` file within it and open it. RStudio should open the project right away. You should see `methodscamp_part1` on the top-right of RStudio—this indicates that you are working in our RStudio project.

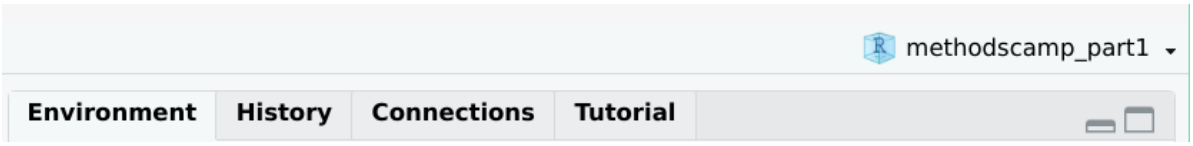


Figure 2: How the bottom-right corner of RStudio looks after opening our project.

That's all for setup! We can now start coding. After opening our RStudio project, we'll begin by opening the `01_r_intro.qmd` file from the "Files" panel, in the bottom-right portion of RStudio. This is a Quarto document,[2] which contains both code and explanations (you can also read the materials in the next chapter of this website).

[2] Perhaps you have used R Markdown before. Quarto is the next iteration of R Markdown, and is both more flexible and more powerful!

1 Intro to R

In Quarto documents like this one, we can write comments by just using plain text. In contrast, code needs to be within *code blocks*, like the one below. To execute a code block, you can click on the little "Play" button or press `Cmd/Ctrl + Shift + Enter` when your keyboard is hovering the code block.

```
2 + 2
```

```
[1] 4
```

That was our first R command, a simple math operation. Of course, we can also do more complex arithmetic:

```
12345 ^ 2  / (200 + 25 - 6 * 2) # this is an inline comment, see the leading "#"
```

```
[1] 715488.4
```

In order to *create* a code block, you can press `Cmd/Ctrl + Alt + i` or click on the little green "+C" icon on top of the script.

> **Exercise**
>
> Create your own code block below and run a math operation.

1.1 Objects

A huge part of R is working with *objects*. Let's see how they work:

```
my_object <- 10 # opt/alt + minus sign will make the arrow
```

```
my_object # to print the value of an object, just call its name
```

```
[1] 10
```

We can now use this object in our operations:

```
2 ^ my_object
```

```
[1] 1024
```

Or even create another object out of it:

```
my_object2 <- my_object * 2
```

```
my_object2
```

```
[1] 20
```

You can delete objects with the `rm()` function (for "remove"):

```
rm(my_object2)
```

1.2 Vectors and functions

Objects can be of different types. One of the most useful ones is the *vector*, which holds a series of values. To create one manually, we can use the `c()` function (for "combine"):

```
my_vector <- c(6, -11, my_object, 0, 20)
```

```
my_vector
```

```
[1]   6 -11  10   0  20
```

One can also define vectors by sequences:

```r
3:10
```

```
[1]  3  4  5  6  7  8  9 10
```

We can use square brackets to retrieve parts of vectors:

```r
my_vector[4] # fourth element
```

```
[1] 0
```

```r
my_vector[1:2] # first two elements
```

```
[1]   6 -11
```

Let's check out some basic functions we can use with numbers and numeric vectors:

```r
sqrt(my_object) # squared root
```

```
[1] 3.162278
```

```r
log(my_object) # logarithm (natural by default)
```

```
[1] 2.302585
```

```r
abs(-5) # absolute value
```

```
[1] 5
```

```r
mean(my_vector)
```

```
[1] 5
```

```r
median(my_vector)
```

```
[1] 6
```

```r
sd(my_vector) # standard deviation
```

```
[1] 11.53256
```

```r
sum(my_vector)
```

```
[1] 25
```

```r
min(my_vector) # minimum value
```

```
[1] -11
```

```r
max(my_vector) # maximum value
```

```
[1] 20
```

```r
length(my_vector) # length (number of elements)
```

```
[1] 5
```

Notice that if we wanted to save any of these results for later, we would need to *assign* them:

```r
my_mean <- mean(my_vector)
```

```r
my_mean
```

```
[1] 5
```

These functions are quite simple: they take one object and do one operation. A lot of functions are a bit more complex—they take multiple objects or take options. For example, see the `sort()` function, which by default sorts a vector *increasingly*:

```r
sort(my_vector)
```

```
[1] -11   0   6  10  20
```

If we instead want to sort our vector *decreasingly*, we can use the `decreasing = TRUE` argument (`T` also works as an abbreviation for `TRUE`).

```
sort(my_vector, decreasing = TRUE)
```

```
[1]  20  10   6   0 -11
```

> **Tip**
>
> If you use the argument values in order, you can avoid writing the argument names (see below). This is sometimes useful, but can also lead to confusing code—use it with caution.
>
> ```
> sort(my_vector, T)
> ```
>
> ```
> [1]  20  10   6   0 -11
> ```

A useful function to create vectors in sequence is `seq()`. Notice its arguments:

```
seq(from = 30, to = 100, by = 5)
```

```
 [1]  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

To check the arguments of a function, you can examine its help file: look the function up on the "Help" panel on RStudio or use a command like the following: `?sort`.

> **Exercise**
>
> Examine the help file of the `log()` function. How can we compute the the base-10 logarithm of `my_object`? Your code:

Other than numeric vectors, character vectors are also useful:

```
my_character_vector <- c("Apple", "Orange", "Watermelon", "Banana")
```

```
my_character_vector[3]
```

```
[1] "Watermelon"
```

```
nchar(my_character_vector) # count number of characters
```

```
[1]  5  6 10  6
```

1.3 Data frames and lists

Another useful object type is the *data frame*. Data frames can store multiple vectors in a tabular format. We can manually create one with the `data.frame()` function:

```
my_data_frame <- data.frame(fruit = my_character_vector,
                            calories_per_100g = c(52, 47, 30, 89),
                            water_per_100g = c(85.6, 86.8, 91.4, 74.9))
```

```
my_data_frame
```

```
       fruit calories_per_100g water_per_100g
1      Apple                52           85.6
2     Orange                47           86.8
3 Watermelon                30           91.4
4     Banana                89           74.9
```

Now we have a little 4x3 data frame of fruits with their calorie counts and water composition. We gathered the nutritional information from the USDA (2019).

We can use the `data_frame$column` construct to access the vectors within the data frame:

```
mean(my_data_frame$calories_per_100g)
```

```
[1] 54.5
```

> **Exercise**
>
> Obtain the maximum value of water content per 100g in the data. Your code:

Some useful commands to learn attributes of our data frame:

```
dim(my_data_frame)
```

```
[1] 4 3
```

```
nrow(my_data_frame)
```

```
[1] 4
```

```
names(my_data_frame) # column names
```

```
[1] "fruit"             "calories_per_100g" "water_per_100g"
```

We will learn much more about data frames in our next module on data analysis.

After talking about vectors and data frames, the last object type that we will cover is the *list*. Lists are super flexible objects that can contain just about anything:

```
my_list <- list(my_object, my_vector, my_data_frame)
```

```
my_list
```

```
[[1]]
[1] 10

[[2]]
[1]   6 -11  10   0  20

[[3]]
       fruit calories_per_100g water_per_100g
1      Apple                52           85.6
2     Orange                47           86.8
3 Watermelon                30           91.4
4     Banana                89           74.9
```

To retrieve the elements of a list, we need to use double square brackets:

```
my_list[[1]]
```

```
[1] 10
```

Lists are sometimes useful due to their flexibility, but are much less common in routine data analysis compared to vectors or data frames.

1.4 Packages

The R community has developed thousands of *packages*, which are specialized collections of functions, datasets, and other resources. To install one, you should use the `install.packages()` command. Below we will install the `tidyverse` package, a suite for data analysis that we will use in the next modules. You just need to install packages once, and then they will be available system-wide.

```
install.packages("tidyverse") # this can take a couple of minutes
```

If you want to use an installed package in your script, you must load it with the `library()` function. Some packages, as shown below, will print descriptive messages once loaded.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

> **Warning**
>
> Remember that `install.packages("package")` needs to be executed just once, while `library(package)` needs to be in each script in which you plan to use the package. In general, never include `install.packages("package")` as part of your scripts or Quarto documents!

2 Tidy data analysis I

The tidyverse is a suite of packages that streamline data analysis in R. After installing the `tidyverse` with `install.packages("tidyverse")` (see the previous module), you can load it with:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

> **Tip**
>
> Upon loading, the `tidyverse` prints a message like the one above. Notice that multiple packages (the constituent elements of the "suite") are actually loaded. For instance, `dplyr` and `tidyr` help with data wrangling and transformation, while `ggplot2` allows us to draw plots. In most cases, one just loads the `tidyverse` and forgets about these details, as the constituent packages work together nicely.

Throughout this module, we will use `tidyverse` functions to load, wrangle, and visualize real data.

2.1 Loading data

Throughout this module we will work with a dataset of senators during the Trump presidency, which was adapted from FiveThirtyEight (2021).

We have stored the dataset in .csv format under the `data/` subfolder. Loading it into R is simple (notice that we need to assign it to an object):

```
trump_scores <- read_csv("data/trump_scores_538.csv")
```

```
Rows: 122 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): bioguide, last_name, state, party
dbl (4): num_votes, agree, agree_pred, margin_trump

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
trump_scores
```

```
# A tibble: 122 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 A000360  Alexander  TN    R           118 0.890      0.856       26.0
 2 B000575  Blunt      MO    R           128 0.906      0.787       18.6
 3 B000944  Brown      OH    D           128 0.258      0.642        8.13
 4 B001135  Burr       NC    R           121 0.893      0.560        3.66
 5 B001230  Baldwin    WI    D           128 0.227      0.510        0.764
 6 B001236  Boozman    AR    R           129 0.915      0.851       26.9
 7 B001243  Blackburn  TN    R           131 0.885      0.889       26.0
 8 B001261  Barrasso   WY    R           129 0.891      0.895       46.3
 9 B001267  Bennet     CO    D           121 0.273      0.417       -4.91
10 B001277  Blumenthal CT    D           128 0.203      0.294      -13.6
# i 112 more rows
```

Let's review the dataset's columns:

- `bioguide`: A unique ID for each politician, from the Congress Bioguide.
- `last_name`
- `state`
- `party`
- `num_votes`: Number of votes for which data was available.
- `agree`: Proportion (0-1) of votes in which the senator voted in agreement with Trump.
- `agree_pred`: Predicted proportion of vote agreement, calculated using Trump's margin (see next variable).

- `margin_trump`: Margin of victory (percentage points) of Trump in the senator's state.

We can inspect our data by using the interface above. An alternative is to run the command `View(trump_scores)` or click on the object in RStudio's environment panel (in the top-right section).

Do you have any questions about the data?

By the way, the `tidyverse` works amazingly with *tidy data*. If you can get your data to this format (and we will see ways to do this), your life will be much easier:

2.2 Wrangling data with `dplyr`

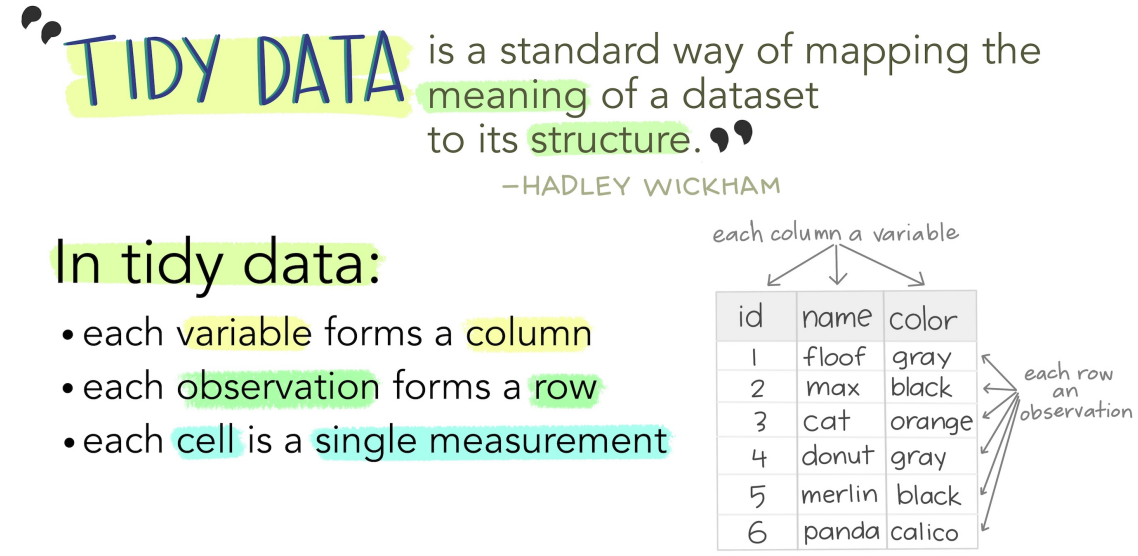
We often need to modify data to conduct our analyses, e.g., creating columns, filtering rows, etc. In the `tidyverse`, these operations are conducted with multiple *verbs*, which we will review now.

2.2.1 Selecting columns

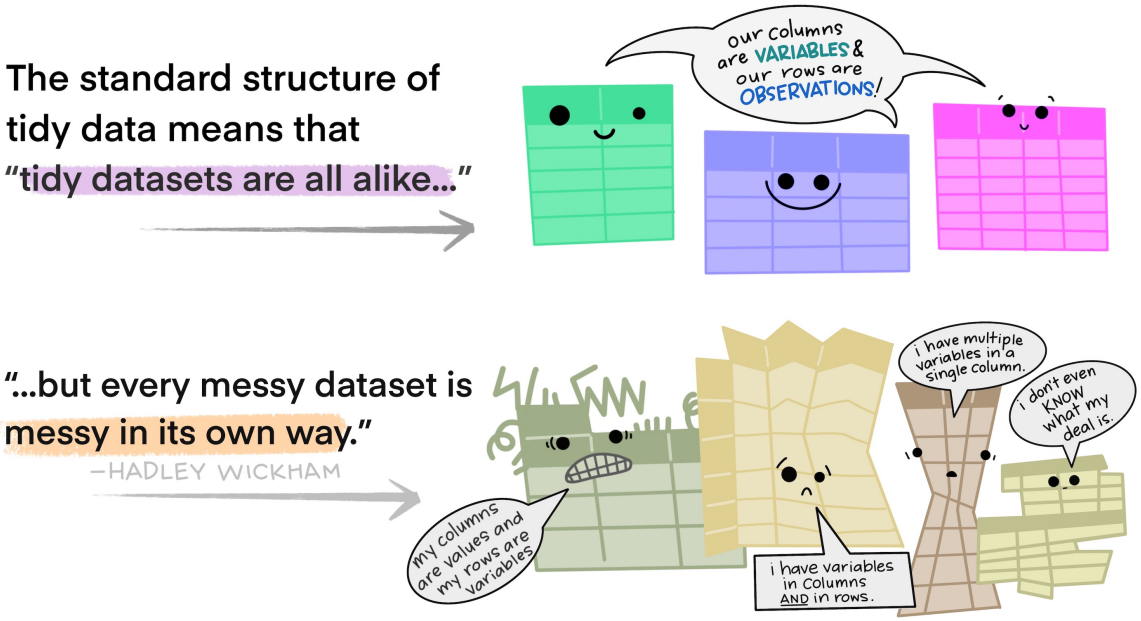
We can select specific columns in our dataset with the `select()` function. All `dplyr` wrangling verbs take a data frame as their first argument—in this case, the columns we want to select are the other arguments.

```
select(trump_scores, last_name, party)
```

```
# A tibble: 122 x 2
   last_name  party
   <chr>      <chr>
 1 Alexander  R
 2 Blunt      R
 3 Brown      D
 4 Burr       R
 5 Baldwin    D
 6 Boozman    R
 7 Blackburn  R
 8 Barrasso   R
 9 Bennet     D
10 Blumenthal D
# i 112 more rows
```



Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10



(a) Source: Illustrations from the Openscapes blog *Tidy Data for reproducibility, efficiency, and collaboration* by Julia Lowndes and Allison Horst.

This is a good moment to talk about "pipes." Notice how the code below produces the same output as the one above, but with a slightly different syntax. Pipes (`|>`) "kick" the object on the left of the pipe to the first argument of the function on the right. One can read pipes as "then," so the code below can be read as "take `trump_scores`, then select the columns `last_name` and `party`." Pipes are very useful to *chain multiple operations*, as we will see in a moment.

```
trump_scores |>
  select(last_name, party)
```

```
# A tibble: 122 x 2
   last_name  party
   <chr>      <chr>
 1 Alexander  R
 2 Blunt      R
 3 Brown      D
 4 Burr       R
 5 Baldwin    D
 6 Boozman    R
 7 Blackburn  R
 8 Barrasso   R
 9 Bennet     D
10 Blumenthal D
# i 112 more rows
```

> **Tip**
>
> You can insert a pipe with the `Cmd/Ctrl + Shift + M` shortcut. If you have not changed the default RStudio settings, an "old" pipe (`%>%`) might appear. While most of the functionality is the same, the `|>` "new" pipes are more readable and don't need any extra packages (to use `%>%` you need the `tidyverse` or one of its packages). You can change this RStudio option in `Tools > Global Options > Code > Use native pipe operator`. Make sure to check the other suggested settings in our Setup module!

Going back to selecting columns, you can select ranges:

```
trump_scores |>
  select(bioguide:party)
```

```
# A tibble: 122 x 4
   bioguide last_name  state party
```

```
   <chr>    <chr>      <chr> <chr>
 1 A000360  Alexander  TN    R
 2 B000575  Blunt      MO    R
 3 B000944  Brown      OH    D
 4 B001135  Burr       NC    R
 5 B001230  Baldwin    WI    D
 6 B001236  Boozman    AR    R
 7 B001243  Blackburn  TN    R
 8 B001261  Barrasso   WY    R
 9 B001267  Bennet     CO    D
10 B001277  Blumenthal CT    D
# i 112 more rows
```

You can also **de**select columns using a minus sign:

```
trump_scores |>
  select(-last_name)
```

```
# A tibble: 122 x 7
   bioguide state party num_votes agree agree_pred margin_trump
   <chr>    <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 A000360  TN    R           118 0.890      0.856       26.0
 2 B000575  MO    R           128 0.906      0.787       18.6
 3 B000944  OH    D           128 0.258      0.642        8.13
 4 B001135  NC    R           121 0.893      0.560        3.66
 5 B001230  WI    D           128 0.227      0.510        0.764
 6 B001236  AR    R           129 0.915      0.851       26.9
 7 B001243  TN    R           131 0.885      0.889       26.0
 8 B001261  WY    R           129 0.891      0.895       46.3
 9 B001267  CO    D           121 0.273      0.417       -4.91
10 B001277  CT    D           128 0.203      0.294      -13.6
# i 112 more rows
```

And use a few helper functions, like `matches()`:

```
trump_scores |>
  select(last_name, matches("agree"))
```

```
# A tibble: 122 x 3
   last_name  agree agree_pred
   <chr>      <dbl>      <dbl>
```

```
 1 Alexander  0.890      0.856
 2 Blunt      0.906      0.787
 3 Brown      0.258      0.642
 4 Burr       0.893      0.560
 5 Baldwin    0.227      0.510
 6 Boozman    0.915      0.851
 7 Blackburn  0.885      0.889
 8 Barrasso   0.891      0.895
 9 Bennet     0.273      0.417
10 Blumenthal 0.203      0.294
# i 112 more rows
```

Or `everything()`, which we usually use to reorder columns:

```
trump_scores |>
  select(last_name, everything())
```

```
# A tibble: 122 x 8
   last_name  bioguide state party num_votes agree agree_pred margin_trump
   <chr>      <chr>    <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 Alexander  A000360  TN    R           118 0.890      0.856       26.0
 2 Blunt      B000575  MO    R           128 0.906      0.787       18.6
 3 Brown      B000944  OH    D           128 0.258      0.642        8.13
 4 Burr       B001135  NC    R           121 0.893      0.560        3.66
 5 Baldwin    B001230  WI    D           128 0.227      0.510        0.764
 6 Boozman    B001236  AR    R           129 0.915      0.851       26.9
 7 Blackburn  B001243  TN    R           131 0.885      0.889       26.0
 8 Barrasso   B001261  WY    R           129 0.891      0.895       46.3
 9 Bennet     B001267  CO    D           121 0.273      0.417       -4.91
10 Blumenthal B001277  CT    D           128 0.203      0.294      -13.6
# i 112 more rows
```

> **Tip**
>
> Notice that all these commands have not edited our existent objects—they have just printed the requested outputs to the screen. In order to modify objects, you need to use the assignment operator (`<-`). For example:
>
> ```
> trump_scores_reduced <- trump_scores |>
>   select(last_name, matches("agree"))
> ```

```
trump_scores_reduced
```

```
# A tibble: 122 x 3
   last_name  agree agree_pred
   <chr>      <dbl>      <dbl>
 1 Alexander  0.890      0.856
 2 Blunt      0.906      0.787
 3 Brown      0.258      0.642
 4 Burr       0.893      0.560
 5 Baldwin    0.227      0.510
 6 Boozman    0.915      0.851
 7 Blackburn  0.885      0.889
 8 Barrasso   0.891      0.895
 9 Bennet     0.273      0.417
10 Blumenthal 0.203      0.294
# i 112 more rows
```

i Exercise

Select the variables `last_name`, `party`, `num_votes`, and `agree` from the data frame. Your code:

2.2.2 Renaming columns

We can use the `rename()` function to rename columns, with the syntax `new_name = old_name`. For example:

```
trump_scores |>
  rename(prop_agree = agree, prop_agree_pred = agree_pred)
```

```
# A tibble: 122 x 8
   bioguide last_name  state party num_votes prop_agree prop_agree_pred
   <chr>    <chr>      <chr> <chr>     <dbl>      <dbl>           <dbl>
 1 A000360  Alexander  TN    R           118      0.890           0.856
 2 B000575  Blunt      MO    R           128      0.906           0.787
 3 B000944  Brown      OH    D           128      0.258           0.642
 4 B001135  Burr       NC    R           121      0.893           0.560
 5 B001230  Baldwin    WI    D           128      0.227           0.510
 6 B001236  Boozman    AR    R           129      0.915           0.851
 7 B001243  Blackburn  TN    R           131      0.885           0.889
```

```
 8 B001261  Barrasso   WY    R           129      0.891           0.895
 9 B001267  Bennet     CO    D           121      0.273           0.417
10 B001277  Blumenthal CT    D           128      0.203           0.294
# i 112 more rows
# i 1 more variable: margin_trump <dbl>
```

This is a good occasion to show how pipes allow us to chain operations. How do we read the following code out loud? (Remember that pipes are read as "then").

```
trump_scores |>
  select(last_name, matches("agree")) |>
  rename(prop_agree = agree, prop_agree_pred = agree_pred)
```

```
# A tibble: 122 x 3
   last_name  prop_agree prop_agree_pred
   <chr>           <dbl>           <dbl>
 1 Alexander       0.890           0.856
 2 Blunt           0.906           0.787
 3 Brown           0.258           0.642
 4 Burr            0.893           0.560
 5 Baldwin         0.227           0.510
 6 Boozman         0.915           0.851
 7 Blackburn       0.885           0.889
 8 Barrasso        0.891           0.895
 9 Bennet          0.273           0.417
10 Blumenthal      0.203           0.294
# i 112 more rows
```

2.2.3 Creating columns

It is common to want to create columns, based on existing ones. We can use `mutate()` to do so. For example, we could want our main variables of interest in terms of percentages instead of proportions:

```
trump_scores |>
  select(last_name, agree, agree_pred) |> # select just for clarity
  mutate(pct_agree = 100 * agree,
         pct_agree_pred = 100 * agree_pred)
```

```
# A tibble: 122 x 5
   last_name  agree agree_pred pct_agree pct_agree_pred
   <chr>      <dbl>      <dbl>     <dbl>          <dbl>
 1 Alexander  0.890      0.856      89.0           85.6
 2 Blunt      0.906      0.787      90.6           78.7
 3 Brown      0.258      0.642      25.8           64.2
 4 Burr       0.893      0.560      89.3           56.0
 5 Baldwin    0.227      0.510      22.7           51.0
 6 Boozman    0.915      0.851      91.5           85.1
 7 Blackburn  0.885      0.889      88.5           88.9
 8 Barrasso   0.891      0.895      89.1           89.5
 9 Bennet     0.273      0.417      27.3           41.7
10 Blumenthal 0.203      0.294      20.3           29.4
# i 112 more rows
```

We can also use multiple columns for creating a new one. For example, let's retrieve the total *number* of votes in which the senator agreed with Trump:

```
trump_scores |>
  select(last_name, num_votes, agree) |> # select just for clarity
  mutate(num_votes_agree = num_votes * agree)
```

```
# A tibble: 122 x 4
   last_name  num_votes agree num_votes_agree
   <chr>          <dbl> <dbl>           <dbl>
 1 Alexander        118 0.890           105
 2 Blunt            128 0.906           116
 3 Brown            128 0.258            33
 4 Burr             121 0.893           108
 5 Baldwin          128 0.227            29
 6 Boozman          129 0.915           118
 7 Blackburn        131 0.885           116
 8 Barrasso         129 0.891           115
 9 Bennet           121 0.273            33.0
10 Blumenthal       128 0.203            26
# i 112 more rows
```

2.2.4 Filtering rows

Another common operation is to filter rows based on logical conditions. We can do so with the `filter()` function. For example, we can filter to only get Democrats:

```
trump_scores |>
  filter(party == "D")
```

```
# A tibble: 55 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 B000944  Brown      OH    D           128 0.258      0.642         8.13
 2 B001230  Baldwin    WI    D           128 0.227      0.510         0.764
 3 B001267  Bennet     CO    D           121 0.273      0.417        -4.91
 4 B001277  Blumenthal CT    D           128 0.203      0.294       -13.6
 5 B001288  Booker     NJ    D           119 0.160      0.290       -14.1
 6 C000127  Cantwell   WA    D           128 0.242      0.276       -15.5
 7 C000141  Cardin     MD    D           128 0.25       0.209       -26.4
 8 C000174  Carper     DE    D           129 0.295      0.318       -11.4
 9 C001070  Casey      PA    D           129 0.287      0.508         0.724
10 C001088  Coons      DE    D           128 0.289      0.319       -11.4
# i 45 more rows
```

Notice that `==` here is a *logical operator*, read as "is equal to." So our full chain of operations says the following: take `trump_scores`, then filter it to get rows where party is equal to "D".

There are other logical operators:

Logical operator	Meaning
`==`	"is equal to"
`!=`	"is not equal to"
`>`	"is greater than"
`<`	"is less than"
`>=`	"is greater than or equal to"
`<=`	"is less than or equal to"
`%in%`	"is contained in"
`&`	"and" (intersection)
`|`	"or" (union)

Let's see a couple of other examples.

```
trump_scores |>
  filter(agree > 0.5)
```

```
# A tibble: 69 x 8
   bioguide last_name state party num_votes agree agree_pred margin_trump
   <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 A000360  Alexander TN    R           118 0.890      0.856         26.0
 2 B000575  Blunt     MO    R           128 0.906      0.787         18.6
 3 B001135  Burr      NC    R           121 0.893      0.560          3.66
 4 B001236  Boozman   AR    R           129 0.915      0.851         26.9
 5 B001243  Blackburn TN    R           131 0.885      0.889         26.0
 6 B001261  Barrasso  WY    R           129 0.891      0.895         46.3
 7 B001310  Braun     IN    R            44 0.909      0.713         19.2
 8 C000567  Cochran   MS    R            68 0.971      0.830         17.8
 9 C000880  Crapo     ID    R           125 0.904      0.870         31.8
10 C001035  Collins   ME    R           129 0.651      0.441         -2.96
# i 59 more rows
```

```
trump_scores |>
  filter(state %in% c("CA", "TX"))
```

```
# A tibble: 4 x 8
  bioguide last_name state party num_votes agree agree_pred margin_trump
  <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
1 C001056  Cornyn    TX    R           129 0.922      0.659         9.00
2 C001098  Cruz      TX    R           126 0.921      0.663         9.00
3 F000062  Feinstein CA    D           128 0.242      0.201       -30.1
4 H001075  Harris    CA    D           116 0.164      0.209       -30.1
```

```
trump_scores |>
  filter(state == "WV" & party == "D")
```

```
# A tibble: 1 x 8
  bioguide last_name state party num_votes agree agree_pred margin_trump
  <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
1 M001183  Manchin   WV    D           129 0.504      0.893         42.2
```

Exercise

1. Add a new column to the data frame, called `diff_agree`, which subtracts `agree` and `agree_pred`. How would you create `abs_diff_agree`, defined as the absolute value of `diff_agree`? Your code:

2. Filter the data frame to only get senators for which we have information on fewer

than (or equal to) five votes. Your code:

3. Filter the data frame to only get Democrats who agreed with Trump in at least 30% of votes. Your code:

2.2.5 Ordering rows

The `arrange()` function allows us to order rows according to values. For example, let's order based on the `agree` variable:

```
trump_scores |>
  arrange(agree)
```

```
# A tibble: 122 x 8
   bioguide last_name    state party num_votes agree agree_pred margin_trump
   <chr>    <chr>        <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 H000273  Hickenlooper CO    D             2 0         0.0302        -4.91
 2 H000601  Hagerty      TN    R             2 0         0.115         26.0
 3 L000570  Luján        NM    D           186 0.124     0.243         -8.21
 4 G000555  Gillibrand   NY    D           121 0.124     0.242        -22.5
 5 M001176  Merkley      OR    D           129 0.155     0.323        -11.0
 6 W000817  Warren       MA    D           116 0.155     0.216        -27.2
 7 B001288  Booker       NJ    D           119 0.160     0.290        -14.1
 8 S000033  Sanders      VT    D           112 0.161     0.221        -26.4
 9 H001075  Harris       CA    D           116 0.164     0.209        -30.1
10 M000133  Markey       MA    D           127 0.165     0.213        -27.2
# i 112 more rows
```

Maybe we only want senators with more than a few data points. Remember that we can chain operations:

```
trump_scores |>
  filter(num_votes >= 10) |>
  arrange(agree)
```

```
# A tibble: 115 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 L000570  Luján      NM    D           186 0.124      0.243        -8.21
 2 G000555  Gillibrand NY    D           121 0.124      0.242       -22.5
```

```
 3 M001176  Merkley    OR    D           129 0.155      0.323       -11.0
 4 W000817  Warren     MA    D           116 0.155      0.216       -27.2
 5 B001288  Booker     NJ    D           119 0.160      0.290       -14.1
 6 S000033  Sanders    VT    D           112 0.161      0.221       -26.4
 7 H001075  Harris     CA    D           116 0.164      0.209       -30.1
 8 M000133  Markey     MA    D           127 0.165      0.213       -27.2
 9 W000779  Wyden      OR    D           129 0.186      0.323       -11.0
10 B001277  Blumenthal CT    D           128 0.203      0.294       -13.6
# i 105 more rows
```

By default, `arrange()` uses increasing order (like `sort()`). To use decreasing order, add a minus sign:

```
trump_scores |>
  filter(num_votes >= 10) |>
  arrange(-agree)
```

```
# A tibble: 115 x 8
   bioguide last_name state party num_votes agree agree_pred margin_trump
   <chr>    <chr>     <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 M001198  Marshall  KS    R           183 0.973      0.933        20.6
 2 C000567  Cochran   MS    R            68 0.971      0.830        17.8
 3 H000338  Hatch     UT    R            84 0.964      0.825        18.1
 4 M001197  McSally   AZ    R           136 0.949      0.562         3.55
 5 P000612  Perdue    GA    R           119 0.941      0.606         5.16
 6 C001096  Cramer    ND    R           135 0.941      0.908        35.7
 7 R000307  Roberts   KS    R           127 0.937      0.818        20.6
 8 C001056  Cornyn    TX    R           129 0.922      0.659         9.00
 9 H001061  Hoeven    ND    R           129 0.922      0.883        35.7
10 C001047  Capito    WV    R           127 0.921      0.896        42.2
# i 105 more rows
```

You can also order rows by more than one variable. What this does is to order by the first variable, and resolve any ties by ordering by the second variable (and so forth if you have more than two ordering variables). For example, let's first order our data frame by party, and then within party order by agreement with Trump:

```
trump_scores |>
  filter(num_votes >= 10) |>
  arrange(party, agree)
```

```
# A tibble: 115 x 8
   bioguide last_name  state party num_votes agree agree_pred margin_trump
   <chr>    <chr>      <chr> <chr>     <dbl> <dbl>      <dbl>        <dbl>
 1 L000570  Luján      NM    D           186 0.124      0.243        -8.21
 2 G000555  Gillibrand NY    D           121 0.124      0.242       -22.5
 3 M001176  Merkley    OR    D           129 0.155      0.323       -11.0
 4 W000817  Warren     MA    D           116 0.155      0.216       -27.2
 5 B001288  Booker     NJ    D           119 0.160      0.290       -14.1
 6 S000033  Sanders    VT    D           112 0.161      0.221       -26.4
 7 H001075  Harris     CA    D           116 0.164      0.209       -30.1
 8 M000133  Markey     MA    D           127 0.165      0.213       -27.2
 9 W000779  Wyden      OR    D           129 0.186      0.323       -11.0
10 B001277  Blumenthal CT    D           128 0.203      0.294       -13.6
# i 105 more rows
```

> **Exercise**
>
> Arrange the data by `diff_pred`, the difference between agreement and predicted agreement with Trump. (You should have code on how to create this variable from the last exercise). Your code:

2.2.6 Summarizing data

`dplyr` makes summarizing data a breeze using the `summarize()` function:

```
trump_scores |>
  summarize(mean_agree = mean(agree),
            mean_agree_pred = mean(agree_pred))
```

```
# A tibble: 1 x 2
  mean_agree mean_agree_pred
       <dbl>           <dbl>
1      0.592           0.572
```

To make summaries, we can use any function that takes a vector and returns one value. Another example:

```
trump_scores |>
  filter(num_votes >= 5) |> # to filter out senators with few data points
  summarize(max_agree = max(agree),
            min_agree = min(agree))
```

```
# A tibble: 1 x 2
  max_agree min_agree
      <dbl>     <dbl>
1         1     0.124
```

Grouped summaries allow us to disaggregate summaries according to other variables (usually categorical):

```
trump_scores |>
  filter(num_votes >= 5) |> # to filter out senators with few data points
  summarize(mean_agree = mean(agree),
            max_agree = max(agree),
            min_agree = min(agree),
            .by = party) # to group by party
```

```
# A tibble: 2 x 4
  party mean_agree max_agree min_agree
  <chr>      <dbl>     <dbl>     <dbl>
1 R          0.876     1         0.651
2 D          0.272     0.548     0.124
```

> **Exercise**
>
> Obtain the maximum absolute difference in agreement with Trump (the `abs_diff_agree` variable from before) for each party.

2.2.7 Overview

Function	Purpose
`select()`	Select columns
`rename()`	Rename columns
`mutate()`	Creating columns
`filter()`	Filtering rows
`arrange()`	Ordering rows
`summarize()`	Summarizing data
`summarize(…, .by = )`	Summarizing data (by groups)

2.3 Visualizing data with `ggplot2`

`ggplot2` is the package in charge of data visualization in the `tidyverse`. It is extremely flexible and allows us to draw bar plots, box plots, histograms, scatter plots, and many other types of plots (see examples at R Charts).

Throughout this module we will use a subset of our data frame, which only includes senators with more than a few data points:

```
trump_scores_ss <- trump_scores |>
  filter(num_votes >= 10)
```

The `ggplot2` syntax provides a unifying interface (the "grammar of graphics" or "gg") for drawing all different types of plots. One draws plots by adding different "layers," and the core code always includes the following:

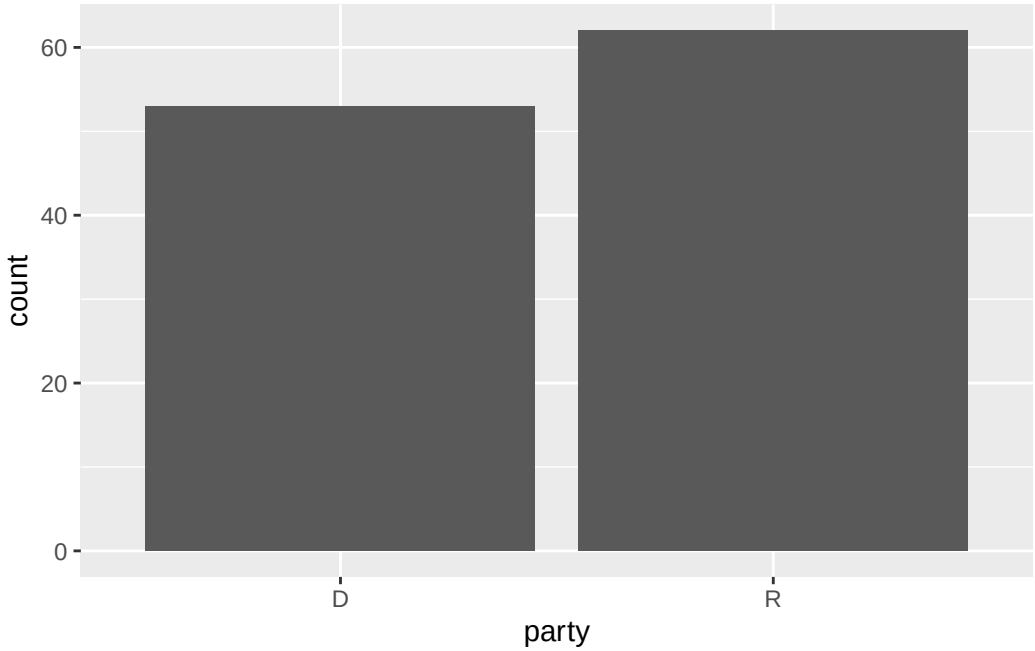
- A `ggplot()` command with a `data =` argument specifying a data frame and a `mapping = aes()` argument specifying "aesthetic mappings," i.e., how we want to use the columns in the data frame in the plot (for example, in the x-axis, as color, etc.).
- "geoms," such as `geom_bar()` or `geom_point()`, specifying what to draw on the plot.

So *all* `ggplot2` commands will have at least three elements: data, aesthetic mappings, and geoms.

2.3.1 Univariate plots: categorical

Let's see an example of a bar plot with a categorical variable:

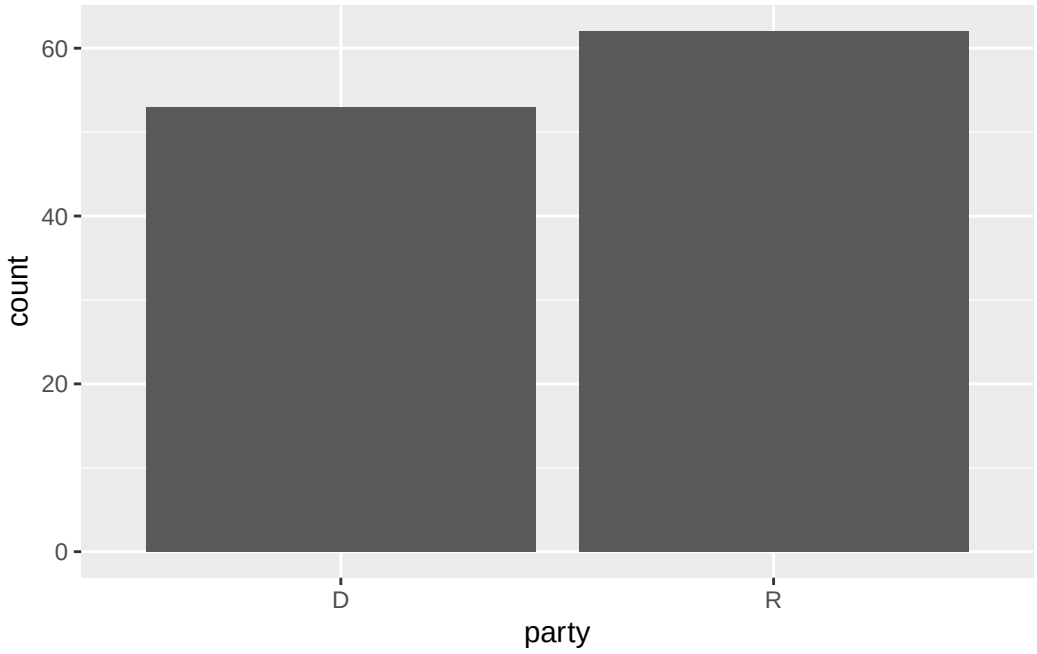
```
ggplot(data = trump_scores_ss, mapping = aes(x = party)) +
  geom_bar()
```



Tip

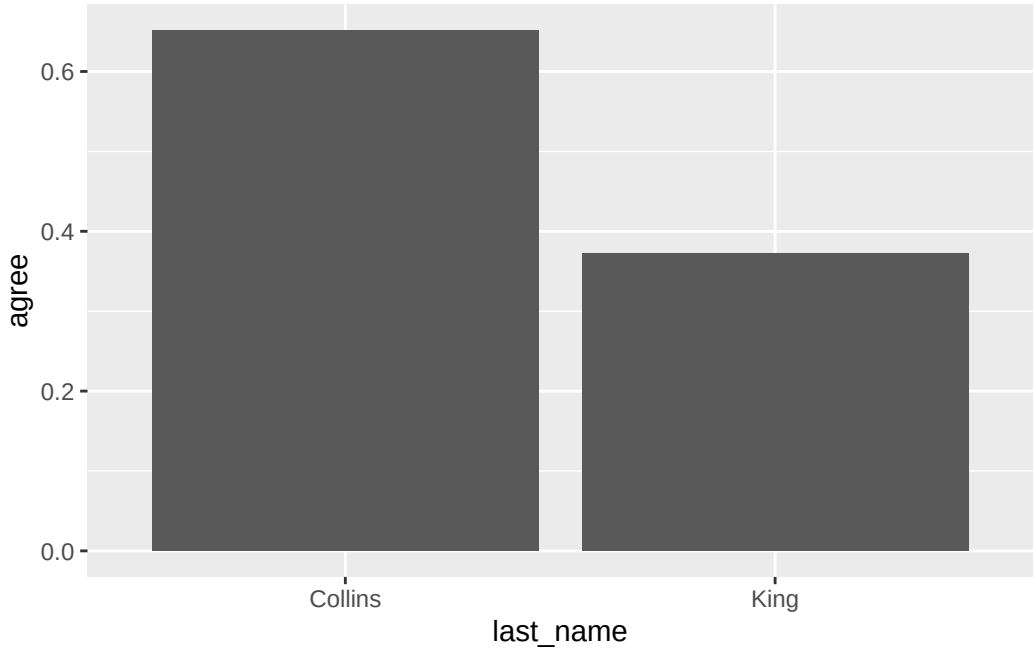
As with any other function, we can drop the argument names if we specify the argument values in order. This is common in `ggplot2` code:

```
ggplot(trump_scores_ss, aes(x = party)) +
  geom_bar()
```



Notice how `geom_bar()` automatically computes the number of observations in each category for us. Sometimes we want to use numbers in our data frame as part of a bar plot. Here we can use the `geom_col()` geom specifying both `x` and `y` aesthetic mappings, in which is sometimes called a "column plot:"

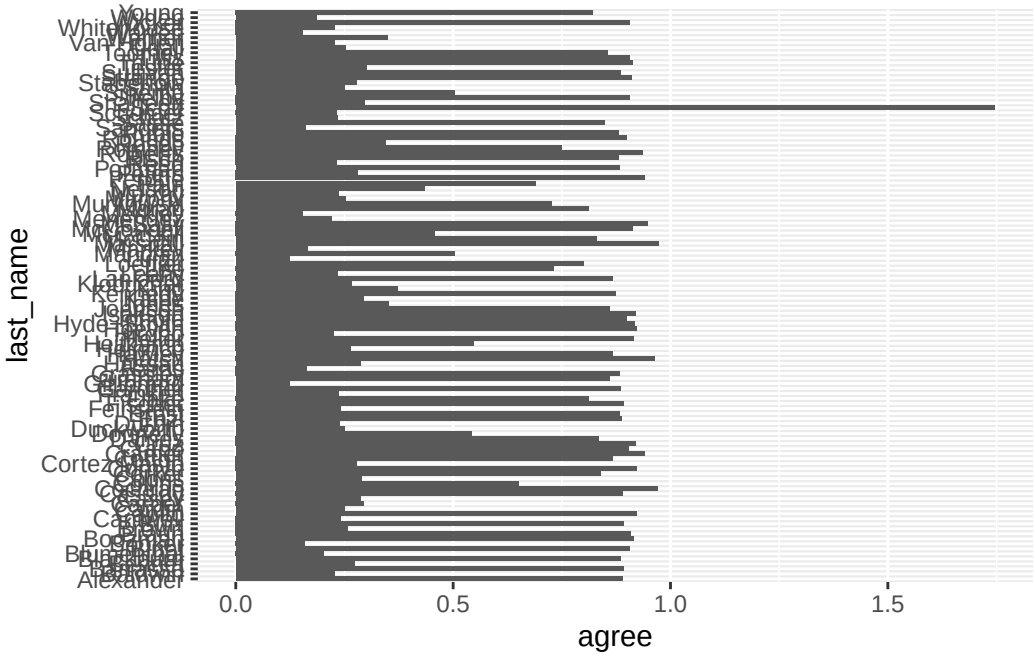
```
ggplot(trump_scores_ss |> filter(state == "ME"),
       aes(x = last_name, y = agree)) +
  geom_col()
```



> **Exercise**
>
> Draw a column plot with the agreement with Trump of Bernie Sanders and Ted Cruz. What happens if you use `last_name` as the `y` aesthetic mapping and `agree` in the `x` aesthetic mapping? Your code:

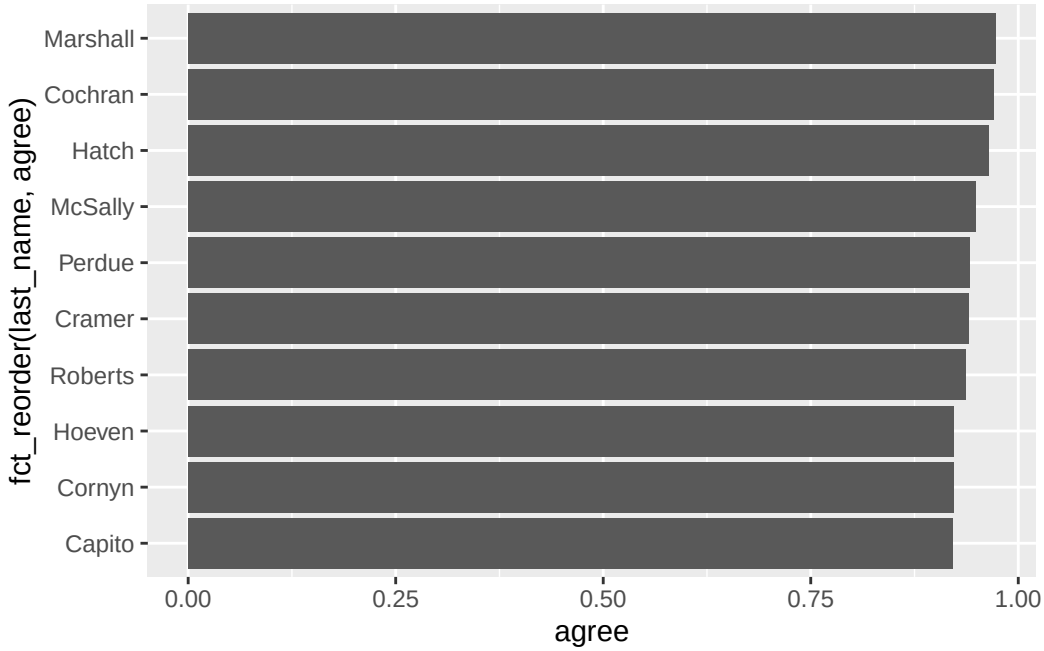
A common use of `geom_col()` is to create "ranking plots." For example, who are the senators with highest agreement with Trump? We can start with something like this:

```
ggplot(trump_scores_ss,
       aes(x = agree, y = last_name)) +
  geom_col()
```



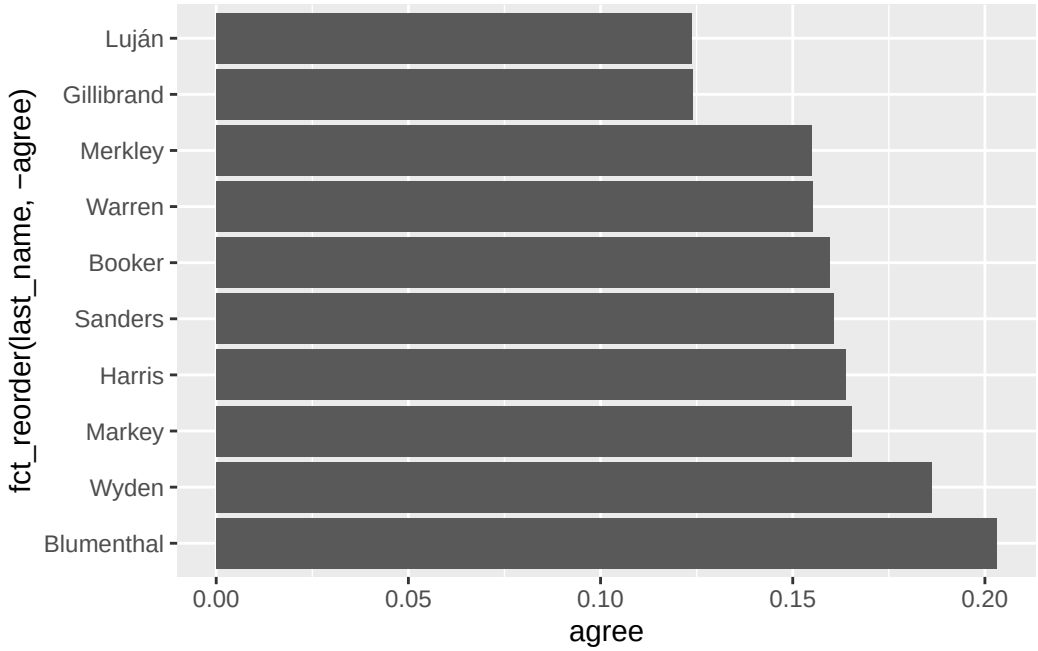
We might want to (1) select the top 10 observations and (2) order the bars according to the `agree` values. We can do these operations with `slice_max()` and `fct_reorder()`, as shown below:

```
ggplot(trump_scores_ss |> slice_max(agree, n = 10),
       aes(x = agree, y = fct_reorder(last_name, agree))) +
  geom_col()
```



We can also plot the senators with the *lowest* agreement with Trump using `slice_min()` and `fct_reorder()` with a minus sign in the ordering variable:

```
ggplot(trump_scores_ss |> slice_min(agree, n = 10),
       aes(x = agree, y = fct_reorder(last_name, -agree))) +
  geom_col()
```

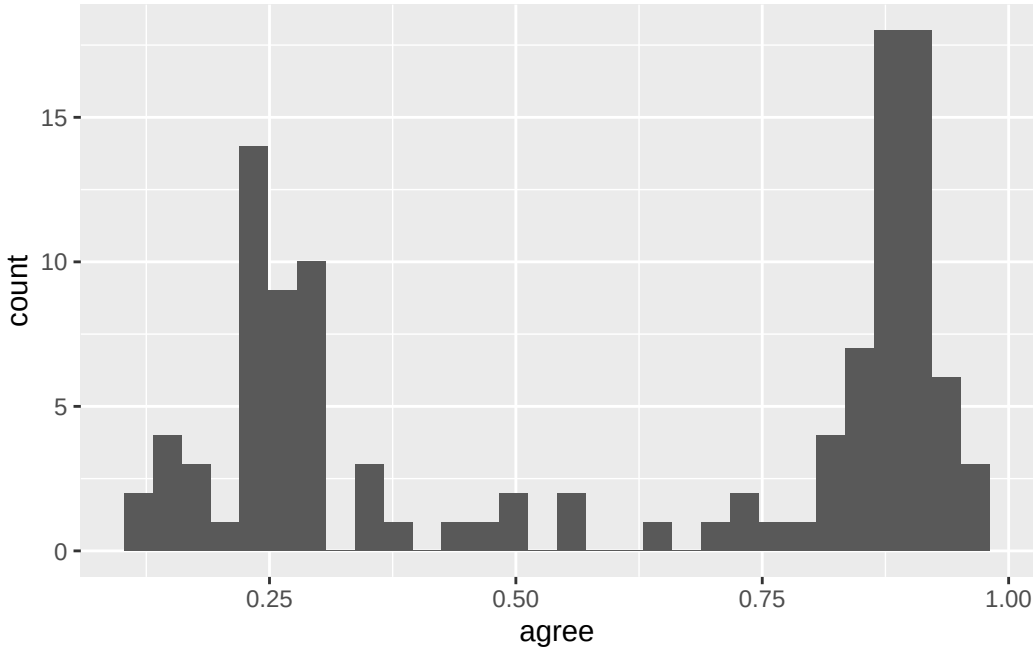


2.3.2 Univariate plots: numerical

We can draw a histogram with `geom_histogram()`:

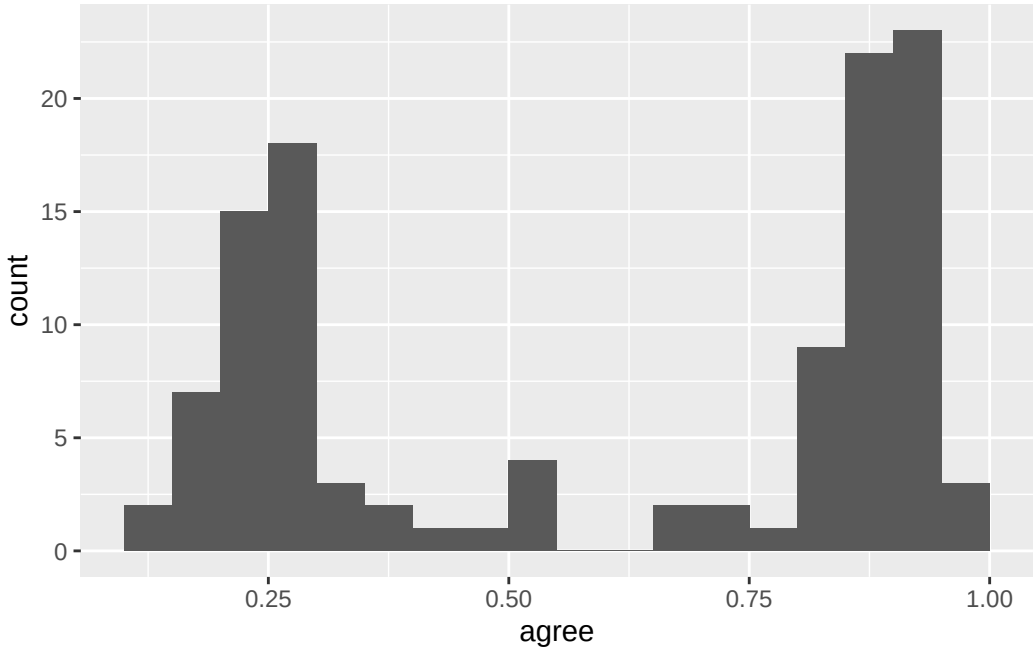
```
ggplot(trump_scores_ss, aes(x = agree)) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



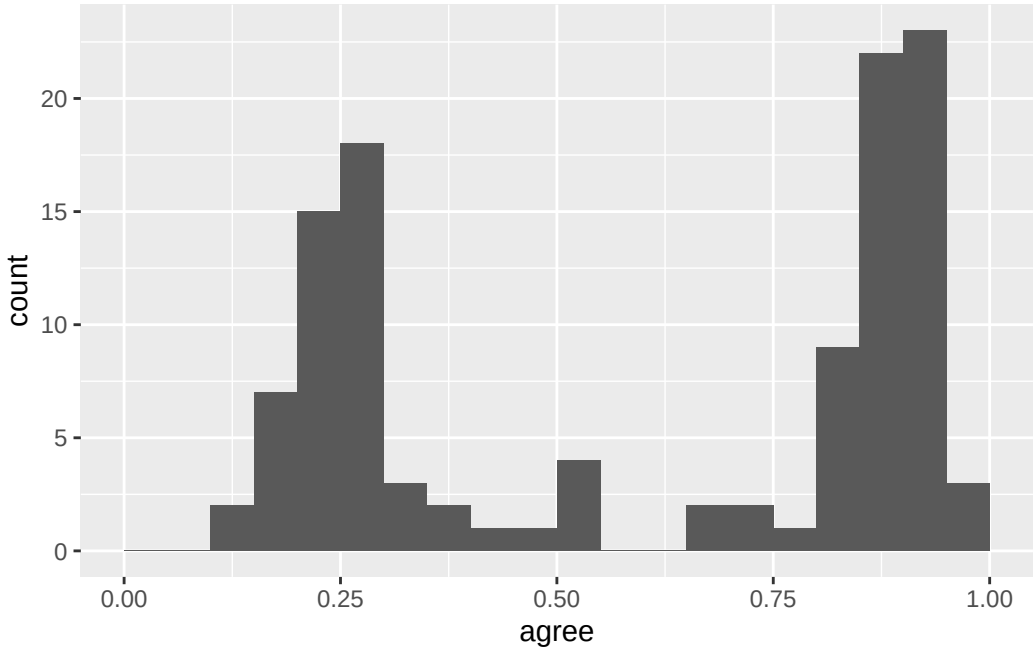
Notice the warning message above. It's telling us that, by default, `geom_histogram()` will draw 30 bins. Sometimes we want to modify this behavior. The following code has some common options for `geom_histogram()` and their explanations:

```
ggplot(trump_scores_ss, aes(x = agree)) +
  geom_histogram(binwidth = 0.05,   # draw bins every 0.05 jumps in x
                 boundary = 0,      # don't shift bins to integers
                 closed   = "left") # close bins on the left
```



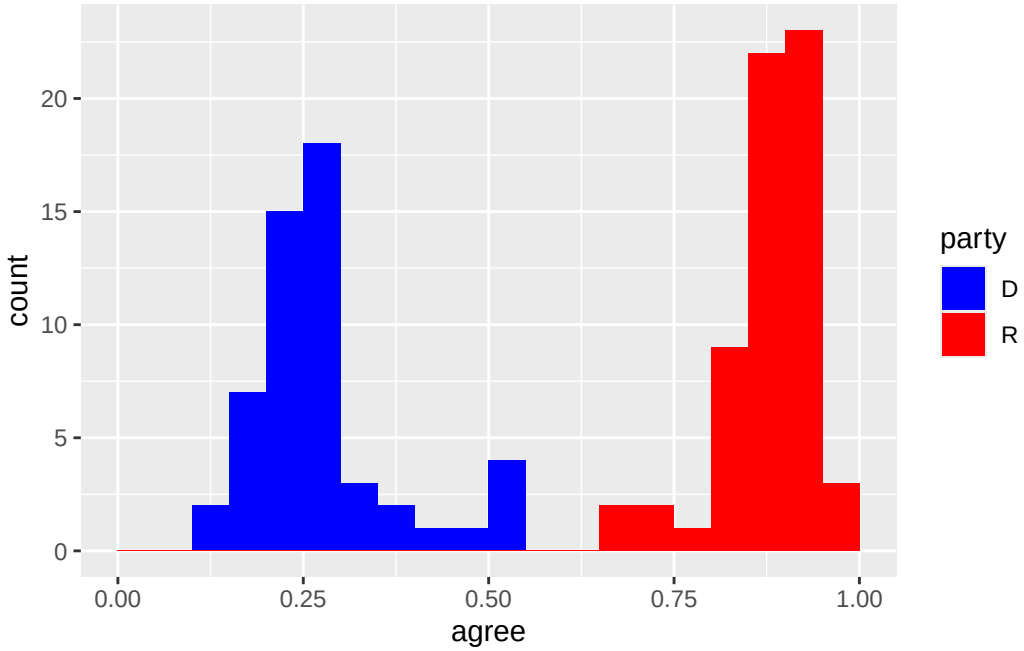
Sometimes we want to manually alter a scale. This is accomplished with the `scale_*()` family of `ggplot2` functions. Here we use the `scale_x_continuous()` function to make the x-axis go from 0 to 1:

```
ggplot(trump_scores_ss, aes(x = agree)) +
  geom_histogram(binwidth = 0.05, boundary = 0, closed   = "left") +
  scale_x_continuous(limits = c(0, 1))
```



Adding the `fill` aesthetic mapping to a histogram will divide it according to a categorical variable. This is actually a bivariate plot!

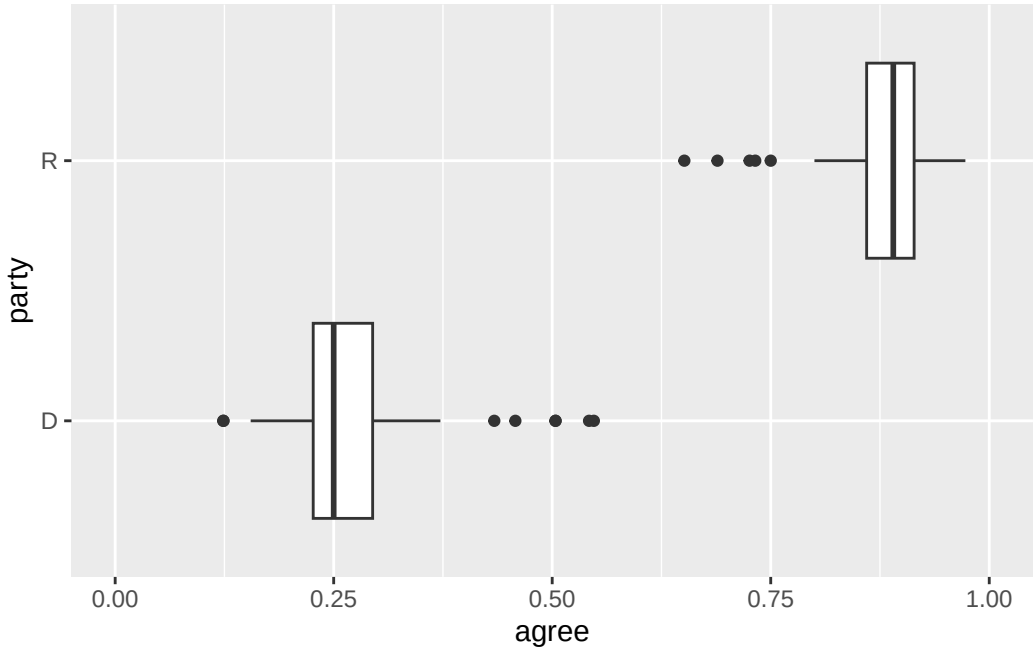
```
ggplot(trump_scores_ss, aes(x = agree, fill = party)) +
  geom_histogram(binwidth = 0.05, boundary = 0, closed   = "left") +
  scale_x_continuous(limits = c(0, 1)) +
  # change default colors:
  scale_fill_manual(values = c("D" = "blue", "R" = "red"))
```



2.3.3 Bivariate plots

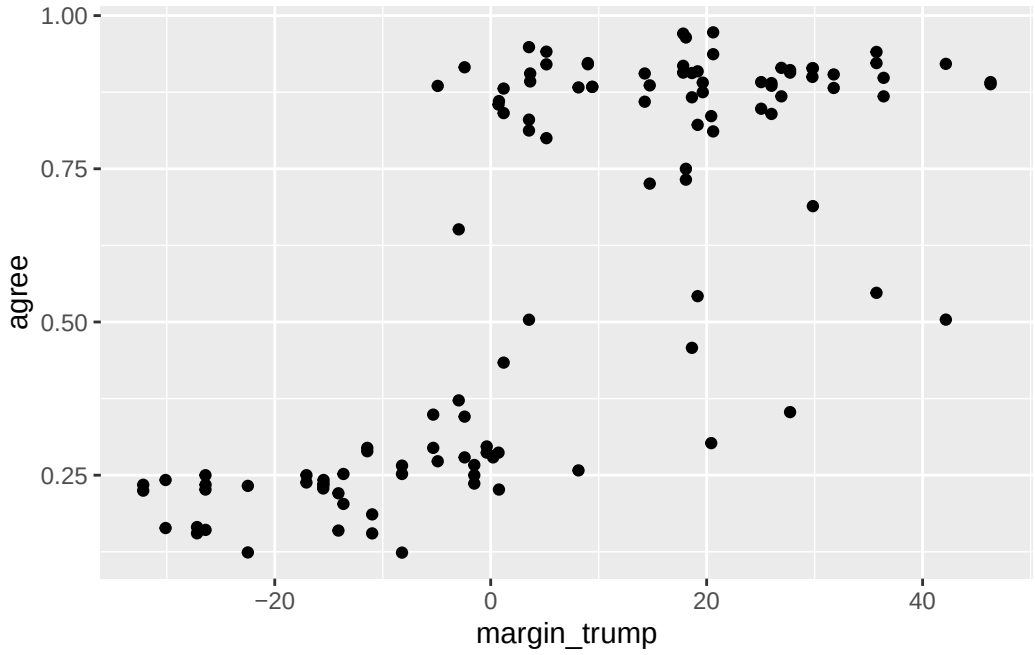
Another common bivariate plot for categorical and numerical variables is the grouped box plot:

```
ggplot(trump_scores_ss, aes(x = agree, y = party)) +
  geom_boxplot() +
  scale_x_continuous(limits = c(0, 1)) # same change as before
```



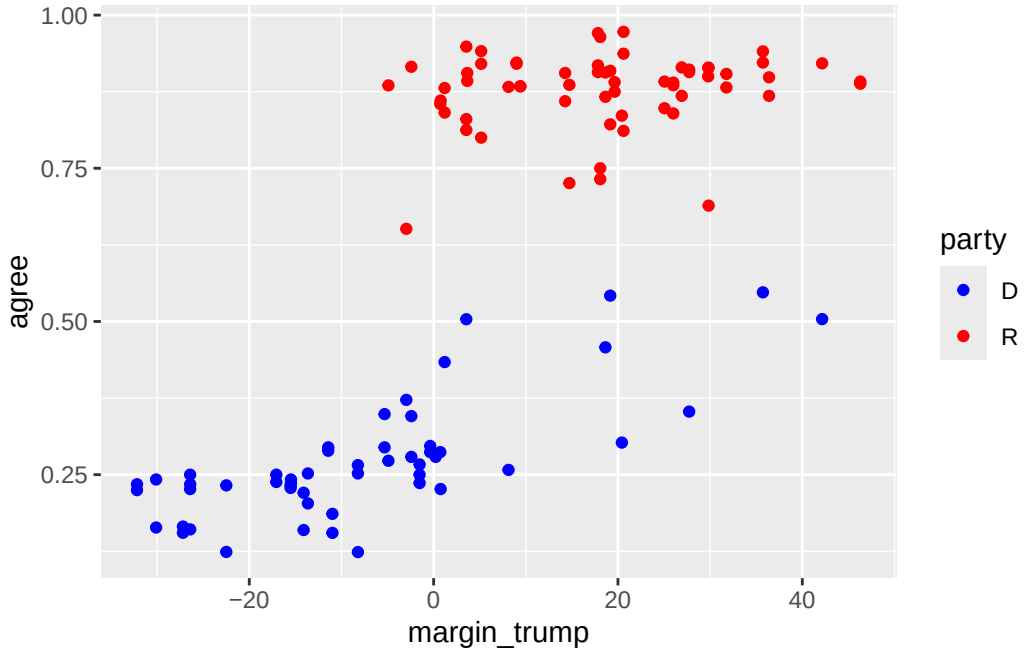
For bivariate plots of numerical variables, scatter plots are made with `geom_point()`:

```
ggplot(trump_scores_ss, aes(x = margin_trump, y = agree)) +
  geom_point()
```



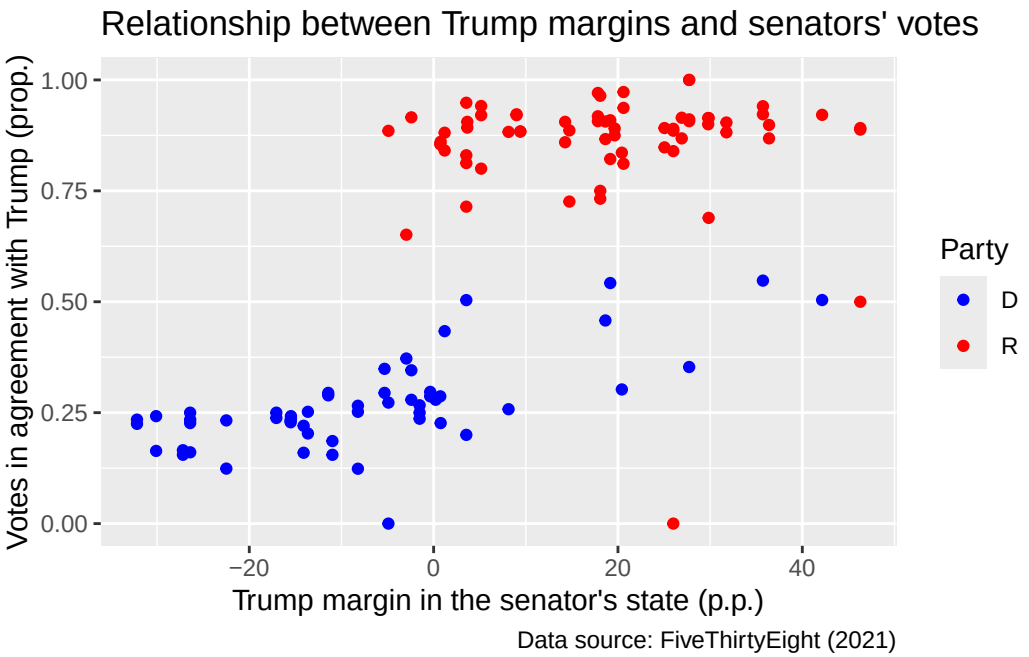
We can add the `color` aesthetic mapping to add a third variable:

```
ggplot(trump_scores_ss, aes(x = margin_trump, y = agree, color = party)) +
  geom_point() +
  scale_color_manual(values = c("D" = "blue", "R" = "red"))
```



Let's finish our plot with the `labs()` function, which allows us to add labels to our aesthetic mappings, as well as titles and notes:

```
ggplot(trump_scores, aes(x = margin_trump, y = agree, color = party)) +
  geom_point() +
  scale_color_manual(values = c("D" = "blue", "R" = "red")) +
  labs(x = "Trump margin in the senator's state (p.p.)",
       y = "Votes in agreement with Trump (prop.)",
       color = "Party",
       title = "Relationship between Trump margins and senators' votes",
       caption = "Data source: FiveThirtyEight (2021)")
```



We will review a few more customization options, including text labels and facets, in a subsequent module.

References

Arel-Bundock, Vincent, Nils Enevoldsen, and CJ Yetman. 2018. "Countrycode: An r Package to Convert Country Names and Country Codes." *Journal of Open Source Software* 3 (28): 848. https://doi.org/10.21105/joss.00848.

Aronow, Peter M, and Benjamin T Miller. 2019. *Foundations of Agnostic Statistics*. Cambridge University Press.

Bank, World. 2023. "World Bank Open Data." https://data.worldbank.org/.

Baydin, Atılım Günes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. "Automatic Differentiation in Machine Learning: A Survey." *The Journal of Machine Learning Research* 18 (1): 5595–5637.

Coppedge, Michael, John Gerring, Carl Henrik Knutsen, Staffan I. Lindberg, Jan Teorell, David Altman, Michael Bernhard, et al. 2022. "V-Dem Codebook V12." Varieties of Democracy (V-Dem) Project. https://www.v-dem.net/dsarchive.html.

Dahlberg, Stefan, Aksen Sundström, Sören Holmberg, Bo Rothstein, Natalia Alvarado Pachon, Cem Mert Dalli, and Yente Meijers. 2023. "The Quality of Government Basic Dataset, Version Jan23." University of Gothenburg: The Quality of Government Institute. https://www.gu.se/en/quality-government doi:10.18157/qogbasjan23.

FiveThirtyEight. 2021. "Tracking Congress In The Age Of Trump [Dataset]." https://projects.fivethirtyeight.com/congress-trump-score/.

Imai, Kosuke, and Nora Webb Williams. 2022. *Quantitative Social Science: An Introduction in Tidyverse*. Princeton; Oxford: Princeton University Press.

Moore, Will H., and David A. Siegel. 2013. *A Mathematics Course for Political and Social Research*. Princeton, NJ: Princeton University Pres.

Pontin, Jason. 2007. "Oppenheimer's Ghost." *MIT Technology Review*, October 15, 2007. https://www.technologyreview.com/2007/10/15/223531/oppenheimers-ghost-3/.

Robinson, David. 2020. *Fuzzyjoin: Join Tables Together on Inexact Matching*. https://github.com/dgrtwo/fuzzyjoin.

Rossi, Hugo. 1996. "Mathematics Is an Edifice, Not a Toolbox." *Notices of the AMS* 43 (10): 1108.

Smith, Danny. 2020. *Survey Research Datasets and R*. https://socialresearchcentre.github.io/r_survey_datasets/.

U. S. Department of Agriculture [USDA], Agricultural Research Service. 2019. "Department of Agriculture Agricultural Research Service." https://fdc.nal.usda.gov/.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software* 59 (10). https://doi.org/10.18637/jss.v059.i10.

Wickham, Hadley, Danielle Navarro, and Thomas Lin Pedersen. 2023. *Ggplot2: Elegant Graphics for Data Analysis*. 3rd ed. https://ggplot2-book.org/.