

Section 5: Getting data into R and summarizing it

Gov 50 Section

October 7, 2026

Today

By the end of the hour you can:

1. Call a function, know its arguments and defaults, and write a short one of your own.
2. Put a data file where R can find it, read it in, and say what kind of data it is.
3. Count rows, compute a mean for each group, and compute a weighted mean.
4. Build a line plot with ggplot, one layer at a time.

Same format as last time. In each block your TF codes on the screen while you run the same lines on your laptop. That part is labeled **Watch**. Then you and a partner try a similar task on your own. That part is labeled **Try it**. When most pairs are done, we go over the answer together. Block 2 is the exception: you do it step by step along with your TF. At the end you knit to PDF and upload to Canvas. It is graded for completion only.

Two shortcuts: **Cmd + Enter** (Mac) or **Ctrl + Enter** (Windows) runs the line your cursor is on. The green arrow at the top right of a chunk runs the whole chunk.

Block 0: Warm up

Run the chunk below. You will knit the whole file in Block 2, once the data file is in place.

```
2 + 2
```

Try it

The chunk below rebuilds the ten-country table from section 3, with one change: Kenya's 1952 value is missing. Run it as it is, then answer the two questions in the chunk under it. Each is one line you wrote in section 3.

```
countries <- data.frame(  
  country = c("Japan", "India", "China", "Vietnam", "Afghanistan",  
              "Egypt", "Kenya", "Nigeria", "Botswana", "Ethiopia"),  
  continent = c("Asia", "Asia", "Asia", "Asia", "Asia",  
                "Africa", "Africa", "Africa", "Africa", "Africa"),  
  life_1952 = c(63.0, 37.4, 44.0, 40.4, 28.8, 41.9, NA, 36.3, 47.6, 34.1),  
  life_2007 = c(82.6, 64.7, 73.0, 74.2, 43.8, 71.3, 54.1, 46.9, 50.7, 52.9)  
)
```

```
# 1. The mean of life_1952. If you get NA, fix it.
# 2. Show only the African rows
```

Block 1: Functions

This block is the last block of section 3, which we did not reach. If you already did it on your own, run through it again quickly, then help a neighbor while everyone catches up.

Watch

You called functions all through section 3. `mean()`, `sum()`, and `c()` are all functions. A function is a named recipe that takes some inputs and gives back a result. Here is one call with its parts labeled.

```
round(3.14159, digits = 2)
|      |           |      |
|      |           |      +--- argument: the value handed to digits
|      |           +----- parameter: the name of the second input
|      +----- argument: the value handed to x, by position
+----- the function
```

A **parameter** is a named slot in the function's definition. `round()` has two that matter here, `x` and `digits`. An **argument** is the value you put into a slot when you call the function. R matches arguments to parameters in two ways. A bare value is matched by position. The first bare value goes to the first parameter, the second to the second, and so on. A value written as `name = value` is matched by name, and then its position does not matter. Positional is shorter. Named is safer once a function has more than one or two parameters, because you do not have to remember the order. A common habit is to give the first argument by position and the rest by name, as in `mean(x, na.rm = TRUE)`.

```
round(3.14159, 2)           # both by position
round(3.14159, digits = 2)  # the second one by name
round(digits = 2, x = 3.14159) # both by name, so the order can change
```

A **default** is the value a parameter falls back on when you do not supply an argument for it. `digits` has a default of 0, so `round(3.14159)` works and rounds to a whole number. `x` has no default. If you give `mean()` nothing, R has nothing to average and stops with an error.

```
round(3.14159)
```

```
mean()
```

Every function has a help page. Type `?round` in the Console and one opens in the bottom right pane. Under the heading **Usage**, find the line `round(x, digits = 0, ...)`. That line lists the parameters in order, and the `= 0` after `digits` is how the help page tells you the default. You can ignore the `...` for now.

You can also write your own function. The pattern is `name <- function(parameters) { body }`. The body, inside the curly brackets, is the recipe, and whatever the last line of the body produces is what the function gives back. Here is a function that converts a temperature from Fahrenheit to Celsius.

```
f_to_c <- function(fahrenheit) {
  (fahrenheit - 32) * 5 / 9
}

f_to_c(212)           # boiling
f_to_c(98.6)          # body temperature
f_to_c(c(32, 50, 100)) # works on a whole vector at once
```

The last call handed the function a vector of three temperatures and got three answers back. That works because the body of the function is ordinary arithmetic, and arithmetic in R works on a whole vector at once, as you saw in section 3. Anything you can do to one number, this function can do to a whole column.

Try it

Write the function that goes the other way, from Celsius to Fahrenheit. The recipe is to multiply by 9/5 and then add 32. Start from the skeleton below and fill in the blank line.

```
c_to_f <- function(celsius) {
  ---
}
```

```
# 1. Fill in the skeleton and run the chunk so R learns the recipe
# 2. Call it for 100 and for 37
# 3. Call it on the vector c(0, 20, 30)
```

Block 2: Reading a data file

This block is different: there is no separate Watch and Try it. Do each step at the same time as your TF, and do not move on until your screen matches.

So far every table you have used was typed in by hand. Real data arrives as a file, and R has to be able to find it. The rule for this course: **one folder for each section, with the .Rmd and its data file side by side.**

Step 1. Make a folder. On your Desktop or in Documents, make a folder named exactly `gov50-section5`, all lowercase, with no spaces.

Step 2. Save this file into it. In RStudio, choose **File > Save As**, go to `gov50-section5`, and save. The copy you downloaded is probably in your Downloads folder. From now on, work in the copy in `gov50-section5`.

Step 3. Get the data file. Go to the section page, ascamara.github.io/gov50, and under Section 5 click the link `nominate.csv`. Your browser saves it to Downloads. Drag it into `gov50-section5`, next to this file. If it opens in Excel or Numbers instead, close it without saving and find the file in Downloads.

Step 4. Point R at the folder. R always has a **working directory**, which is the folder it looks in when you give it a file name. `getwd()`, short for “get working directory,” shows which folder that is. Click in the Console at the bottom of RStudio, type this, and press Enter:

```
getwd()
```

It probably shows your home folder, not `gov50-section5`. To change it, choose **Session > Set Working Directory > To Source File Location**, which means the folder this .Rmd is saved in. Look at the Console: the menu typed a line for you that looks like this.

```
setwd("~/Desktop/gov50-section5")
```

`setwd()`, short for “set working directory,” moves R to the folder in quotes. Run `getwd()` in the Console again, and it now ends in `gov50-section5`. Use the menu, or type `setwd()` in the Console, but never put a `setwd()` line in your `.Rmd`. The folder’s address is different on every laptop, so that line would break the file for your partner and your TF. Chunks and knitting already look in the folder where the `.Rmd` is saved.

Step 5. Check, then knit. Run this chunk. Both this `.Rmd` and `nominate.csv` should be on the list.

```
list.files()      # the files in the working directory
```

Now knit the whole document with the **Knit** button, or **Cmd + Shift + K** (Mac) or **Ctrl + Shift + K** (Windows). Knitting runs every chunk in the file, including the one below that reads `nominate.csv`. So if a PDF opens, your setup works. If you get an error that says **cannot open the connection**, the data file and this file are not in the same folder. If it says **there is no package called 'ggplot2'**, run `install.packages("ggplot2")` in the Console. Either way, raise a hand now rather than later.

Step 6. Read the file and look at it. The function you need depends on the file type, which is the ending of the file name. This file ends in `.csv`, which is plain text with commas between the values, and base R reads it with `read.csv()`. You give it the file name in quotes and store the result under a name of your choosing. What comes back is a data frame, so everything from section 3 applies. Then `str()` from section 3 shows what you loaded: how many rows and columns, each column’s name and class, and its first few values.

```
nominate <- read.csv("nominate.csv")
str(nominate)
```

The first line of that output says the file has 31,663 rows, which R calls observations, and 7 columns, which R calls variables. To see the whole table, click `nominate` in the **Environment** pane at the top right. It opens like a spreadsheet you can scroll and sort. Do not print the whole thing in a chunk, or your knitted PDF will run to hundreds of pages.

Column	What it holds
year	the first year of that Congress: 1879, 1881, ..., 2025
id	a number for each member, the same in every Congress they served. A member who switches party gets a new one
name	the member’s name, last name first
state	two-letter state abbreviation
district	the district number within the state; 0 marks a non-voting delegate from DC or a territory
party	"Democrat", "Republican", or "Other"
score	the NOMINATE score, from about -1 (most liberal) to 1 (most conservative)

The data come from Voteview. This file has every member of the House of Representatives from 1879 to 2025. One row is one member in one two-year Congress, so a representative who served 30 years appears 15 times.

Where the score comes from. Start with every vote a Congress took. Picture a table with one row per member and one column per bill, holding 1 for a yes vote and 0 for a no. NOMINATE squishes that whole table down into one number per member, so that members who vote alike end up close together. Here is a tiny version with four members and three bills.

Member	Bill 1	Bill 2	Bill 3
A	1	1	1
B	1	1	0
C	0	1	0
D	0	1	0

Bills differ in how much they tell us.

- Bill 1 is a bill where **all liberals vote one way and all conservatives vote the opposite way**. It is very useful for telling members apart: A and B on one side, C and D on the other.
- Bill 2 is a bill **everyone votes for**. It is useless for telling members apart, and a bill everyone votes against would be just as useless.
- Bill 3 is in between. It separates A from B, so A ends up furthest to one side.

Real bills fall everywhere between those two poles, and NOMINATE uses all of them to place each member. You do not need to know more than that to use the score.

What kind of data is this? Datasets come in three shapes.

- **Cross-sectional data** has many units at one point in time. One Congress is a cross-section, such as the 2025 House in Block 3.
- **Time-series data** has one unit across many points in time. One legislator's rows are a time series. So is one party's median score over the years, which you plot in Block 4.
- **Panel data** has many units, each followed across many points in time. This whole file is a panel: every member, in every Congress they served.

You can turn a panel into the other two. Keep one Congress and you have a cross-section. Keep one member, or summarize each Congress down to one number, and you have a time series.

Other file types need other functions. Your replication data may arrive as a `.dta` file, which is Stata's format. For that you install the `haven` package once, in the Console, with `install.packages("haven")`, put `library(haven)` at the top of the `.Rmd`, and read the file with `read_dta("file.dta")`. Same folder, same habit of looking first, one different function.

Block 3: Summaries

Watch

Now that the file is loaded, this block asks it some questions. Every move here is one you will make on your replication data in the first week.

Counting. `table()` counts how many times each value appears in a column. To count people instead of rows, use the `id` column: `unique()` drops the repeats from a vector, and `length()` counts what is left.

```
table(nominate$party)      # counts rows, so a long career counts many times
length(unique(nominate$id)) # how many different members
```

Narrowing down. `subset()` from section 3 keeps the rows that pass a test. Here the test keeps the current House, the Congress that began in 2025. The result is **cross-sectional data**: many members at one point in time.

```
house_now <- subset(nominate, year == 2025)
nrow(house_now)
table(house_now$party)
```

There are more rows than the House's 435 seats, because the non-voting delegates are in the file and so are members who replaced someone partway through the Congress.

Weighted means. `mean()` gives every value the same say. `weighted.mean(x, w)` lets each value in `x` count as much as its weight in `w`: it multiplies each value by its weight, adds those up, and divides by the total weight. If every weight is 1, that is the ordinary mean, which is the average ideology of the whole chamber. Two members have no score yet, so both need `na.rm = TRUE`.

```
house_now$everyone <- 1 # a weight of 1 for every member
weighted.mean(house_now$score, house_now$everyone, na.rm = TRUE)
mean(house_now$score, na.rm = TRUE) # the same number
```

A weight of 0 leaves a member out. So to get the mean for one party, make an **indicator** column: 1 for members of that party, 0 for everyone else. `ifelse()` does that. It takes a test, the value to use where the test is TRUE, and the value to use where it is FALSE. Read the line below as: if the member is a Democrat, 1, otherwise 0.

```
house_now$dem_indicator <- ifelse(house_now$party == "Democrat", 1, 0)
house_now$party[1:5] # check the first five members
house_now$dem_indicator[1:5] # 1 for each Democrat, 0 otherwise
weighted.mean(house_now$score, house_now$dem_indicator, na.rm = TRUE) # only Democrats count
```

Hold on to that number. `tapply()` below gives it again.

A mean for every group at once. Making an indicator for every party would be tedious. `tapply()` does all the groups in one call. It takes three arguments, in this order:

1. the column to summarize, here `house_now$score`;
2. the column that says which group each row belongs to, here `house_now$party`;
3. the function to apply to each group, here `mean`, written without parentheses because `tapply()` is the one that runs it. Any summary function works in this slot, such as `median`, `max`, or `min`.

Anything after those three is handed on to that function, which is where `na.rm = TRUE` goes. The result is one number per group, labeled with the group's name.

```
tapply(house_now$score, house_now$party, mean, na.rm = TRUE)
```

Finding a row. `which.max()` gives the position of the largest value in a vector, skipping any NAs. Put that position inside square brackets on another column and you get the value from the same row. Here that is the name of the most conservative member of the current House. `which.min()` does the same for the smallest value.

```
which.max(house_now$score)
house_now$name[which.max(house_now$score)]
```

Try it

```
# 1. Make a column rep_indicator that is 1 for Republicans and 0 for everyone
#     else. Use it as the weights in weighted.mean() to get the Republican mean.
#     Does it match tapply()?
# 2. The median score by party in the current House
# 3. Who is the most liberal member?
```

Block 4: A first plot

Watch

The file lets us draw one of the best-known pictures in American politics: the two parties' positions over time. Run the next chunk as it is; you do not need to type it. It uses `tapply()` from Block 3 to get each party's median score in every Congress, and puts them in a table with one row per Congress. That table is **time-series data**, made from the panel by summarizing each Congress down to one number per party.

```
dems <- subset(nominate, party == "Democrat")
reps <- subset(nominate, party == "Republican")
meds <- data.frame(year = unique(nominate$year),
                  dem = tapply(dems$score, dems$year, median, na.rm = TRUE),
                  rep = tapply(reps$score, reps$year, median, na.rm = TRUE))
meds[1:3, ]
```

Plotting with ggplot. You used the `ggplot2` package in Lab 1. Load it first. In your own files, put `library()` lines at the top.

```
library(ggplot2)
```

A `ggplot` is built in layers, and each `+` adds one layer on top of what is already there. We build this one in three steps.

Step 1: the data, the axes, and one line. `ggplot()` takes the data frame. Inside it, `aes()`, short for aesthetics, says which column goes where: `year` across the bottom, `rep` up the side. Then `+ geom_line()` draws a line through those points. “Geom” is short for geometric object, the shape that marks the data.

```
ggplot(meds, aes(x = year, y = rep)) +
  geom_line(color = "red")
```

Step 2: a second line. Another `geom_line()` adds a second layer. Its own `aes(y = dem)` swaps in the Democratic column for this layer only. It still uses `year` for `x`, from the first line.

```
ggplot(meds, aes(x = year, y = rep)) +
  geom_line(color = "red") +
  geom_line(aes(y = dem), color = "blue")
```

Step 3: finish it. A dashed line at zero, the y-axis set to the whole scale from -1 to 1 , and labels.

```
ggplot(meds, aes(x = year, y = rep)) +
  geom_line(color = "red") +
  geom_line(aes(y = dem), color = "blue") +
  geom_hline(yintercept = 0, linetype = "dashed") +
```

```
ylim(-1, 1) +
labs(x = "Year", y = "Median NOMINATE score",
     title = "House party medians, 1879 to 2025",
     subtitle = "Republicans in red, Democrats in blue")
```

Piece	What it adds
<code>ggplot(meds, aes(x = year, y = rep))</code>	the data, and which column goes on which axis
<code>geom_line(color = "red")</code>	a red line through the points
<code>geom_line(aes(y = dem), color = "blue")</code>	a second line, with dem up the side
<code>geom_hline(yintercept = 0, linetype = "dashed")</code>	a dashed horizontal line at 0
<code>ylim(-1, 1)</code>	the lowest and highest value on the y-axis
<code>labs(x = , y = , title = , subtitle =)</code>	the axis labels, title, and subtitle

One rule: **the + goes at the end of a line, never at the start of the next one.** A line that ends in + tells R the plot is not finished yet.

Read the plot before you move on. Both parties drifted toward the center in the middle of the 20th century, then apart. Since the mid-1970s the Republican median has moved much more than the Democratic one: from 0.25 in 1975 to 0.53 in 2025, while the Democrats went from -0.34 to -0.395 .

Try it

The picture has two lines. The question people usually ask about polarization is about the distance between them. Each row of `meds` has both medians for the same year, so the distance is one subtraction, the way you added the `change` column in section 3. Then plot it with `ggplot`.

```
# 1. Add a column called gap to meds: the rep column minus the dem column
# 2. Plot gap against year as a line with ggplot, with axis labels and a title
```

What you learned today

Objective	The move	Looks like
1. Call and write functions	<code>name(arg = value), ?name, function()</code>	<code>round(3.14, digits = 1), f_to_c(212)</code>
2. Get a file into R, and say what kind of data it is	a folder per section, <code>getwd()</code> , <code>setwd()</code> , <code>list.files()</code> , <code>read.csv()</code> , <code>str()</code> ; cross-section, time series, panel	<code>nominat <- read.csv("nominat.csv")</code>
3. Count and summarize by group	<code>table()</code> , <code>unique()</code> , <code>subset()</code> , <code>ifelse()</code> , <code>weighted.mean()</code> , <code>tapply()</code> , <code>which.max()</code>	<code>tapply(values, groups, mean, na.rm = TRUE)</code>
4. Build a ggplot	<code>ggplot()</code> , <code>aes()</code> , <code>geom_line()</code> , <code>labs()</code> , each added with +	<code>ggplot(meds, aes(x = year, y = gap)) + geom_line()</code>

Two habits from today. Keep the data file next to the `.Rmd` and refer to it by name alone. Look at what you loaded with `str()` before you compute anything.

Before you leave

Knit this file to a PDF one more time and upload the PDF to Canvas. It is graded for completion, so a file with honest attempts at each Try it counts in full.

If the knit fails, the error names the line that broke. Fix that line, or put a `#` in front of it so R skips it, and knit again.

For your replication: the first thing your group does with its data file is Blocks 2 and 3 of today. Put the file in your folder and read it with the function for its file type. Then look at it with `str()`, count the columns that matter with `table()`, and compute a mean or median by group with `tapply()`. Do that before you try to reproduce any figure.