Methods Camp

UT Austin, Department of Government

Andrés Cruz and Matt Martin

2023-08-10

Table of contents

CI	ass so	nedule	5
	Desc	iption	5
	Cou	se outline	5
	Cont	act info	7
	Ackı	owledgements	7
Se	tup		8
	Insta	ling R and RStudio	8
		ng up for Methods Camp	10
1	Intro	to R	11
	1.1	Objects	11
	1.2	Vectors and functions	12
	1.3	Data frames and lists	16
	1.4	Packages	18
2	Tidy	data analysis I	19
_	2.1	Loading data	19
	2.2	Wrangling data with dplyr	21
		2.2.1 Selecting columns	21
		2.2.2 Renaming columns	26
		2.2.3 Creating columns	27
		2.2.4 Filtering rows	28
		2.2.5 Ordering rows	$\frac{20}{31}$
		2.2.6 Summarizing data	33
		2.2.7 Overview	34
	2.3	Visualizing data with ggplot2	35
	۷.5	2.3.1 Univariate plots: categorical	35
		2.3.2 Univariate plots: numerical	33 41
		2.3.3 Bivariate plots	45
3	N/		50
3	Mat		
	3.1	Introduction	
		3.1.1 Scalars	
		3.1.2 Vectors	50

	3.2	Operators
		3.2.1 Summation
		3.2.2 Product
	3.3	Matrices
		3.3.1 Basics
		3.3.2 Structure
	3.4	Matrix operations
		3.4.1 Addition and subtraction
		3.4.2 Scalar multiplication
		3.4.3 Matrix multiplication
		3.4.4 Properties of operations
	3.5	Special matrices
	3.6	Transpose
	3.7	Inverse
	3.8	Linear systems and matrices
	3.9	OLS and matrices
	5.9	
		1
		1
		3.9.3 Linear regression model
		3.9.4 Estimates
4	Tidy	data analysis II 71
•	4.1	Loading data in different formats
	7.1	4.1.1 CSV and R data files
		4.1.2 Excel data files
		4.1.3 Stata and SPSS data files
	4.9	
	4.2	Recoding variables
	4.3	Missing values
	4.4	Pivoting data
	4.5	Merging datasets
	4.6	Plotting extensions: trend graphs, facets, and customization
5	Fund	ctions 94
•	5.1	Basics
	0.1	5.1.1 What is a function?
		5.1.2 Vertical line test
	5.2	Functions in R
	5.3	Common types of functions
		5.3.1 Linear functions
		5.3.2 Quadratic functions
		5.3.3 Cubic functions
		5.3.4 Polynomial functions

Referei	ıces		111
5.5	Comp	osite functions (functions of functions)	109
	5.4.6	Logarithms in R	107
	5.4.5	Illustration of e	106
	5.4.4	Natural logarithms	105
	5.4.3	Basic rules	105
	5.4.2	Relationships	105
	5.4.1	Logarithms	104
5.4	Logar	ithms and exponents	104
	5.3.5	Exponential functions	103

Class schedule

Date	Time	Location
Thurs, Aug. 10	9:00 AM - 4:00 PM	RLP 1.302D
Fri, Aug. 11	9:00 AM - 4:00 PM	RLP $1.302E$
Sat, Aug. 12	No class	-
Sun, Aug. 13	No class	-
Mon, Aug. 14	9:00 AM - 4:00 PM	RLP $1.302E$
Tues, Aug. 15	9:00 AM - 4:00 PM	RLP $1.302E$
Weds, Aug. 16	9:00 AM - 4:00 PM	RLP 1.302E

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 Stats 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 Thursday, August 10th and Friday, August 11th as well as from Monday, August 14th through Wednesday, August 16th. Classes at UT begin the start of the following week on Monday, August 22nd. Below is a tentative schedule outlining what will be covered in the class, although we may rearrange things a bit if we find we're going too slowly or too quickly through any of the material.

Course outline

- 1 Thursday morning: R and RStudio
 - Introductions

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

2 Thursday afternoon: tidyverse basics I

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

3 Friday 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

4 Friday afternoon: tidyverse basics II

- Loading data in different formats (.csv, R, Excel, Stata, SPSS)
- Recoding values (if_else(), case_when())
- Missing values
- Pivoting data
- Merging data
- Plotting extensions (trend graphs, facets, customization)

5 Monday morning: Functions

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

6 Monday afternoon: Calculus

- Limits
- Derivatives
- Integrals
- Calculus in R

7 Tuesday morning: Probability

• Basic concepts

- Random variables and their properties
- Common distributions

8 Tuesday afternoon: Simulations

- Monte Carlo simulations
- Sampling
- Loops in R
- Bootstrapping

9 Wednesday morning: Text analysis

- String manipulation with stringr
- Simple text analysis with tidytext and visualization

10 Wednesday 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:

- Andrés Cruz: andres.cruz@utexas.edu
- Matt Martin: mjmartin@utexas.edu

If you are interested in learning more about our research, you can also check out our respective websites and Twitter accounts (or should we say X...):

- Andrés Cruz: [Website] [Twitter]
- Matt Martin: [Website] [Twitter]

Acknowledgements

We thank previous Methods Camp instructors for their accumulated experience and materials, which we have based ours upon. UT GOV professors Stephen Jessee, Connor Jerzak, and Dan Nielson have given us amazing feedback for this iteration of Methods Camp. All errors remain our own (and will hopefully be fixed with your help!).

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, follow the instructions below):

- 1. Download and install R from the official website, CRAN. Click on "Download R for <Windows/Mac>" 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 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.

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

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:

- 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...

¹The 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.

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.

There are two 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.
- Download Part 2 of the class materials (available soon).



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 O1_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).

²Perhaps 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 ^{\circ} 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.

i 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</pre>
```

```
[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</pre>
```

[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</pre>
```

One can also define vectors by sequences:

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

We can use square brackets to retrieve parts of vectors:

```
my_vector[4] # fourth element
[1] 0
  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:
  sqrt(my_object) # squared root
[1] 3.162278
  log(my_object) # logarithm (natural by default)
[1] 2.302585
  abs(-5) # absolute value
[1] 5
  mean(my_vector)
[1] 5
  median(my_vector)
[1] 6
  sd(my_vector) # standard deviation
[1] 11.53256
```

```
sum(my_vector)

[1] 25

min(my_vector) # minimum value

[1] -11

max(my_vector) # maximum value

[1] 20

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:

```
my_mean <- mean(my_vector)
my_mean</pre>
```

[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*:

```
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



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.

i Exercise

Examine the help file of the log() function. How can we compute 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]</pre>
```

[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,</pre>
                                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
       Apple
                             52
                                           85.6
1
2
      Orange
                             47
                                           86.8
3 Watermelon
                             30
                                           91.4
                                           74.9
      Banana
                             89
```

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

i 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
```

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)</pre>
  my_list
[[1]]
[1] 10
[[2]]
Γ1]
      6 -11 10
                   0 20
[[3]]
       fruit calories_per_100g water_per_100g
1
       Apple
                              52
                                            85.6
                              47
                                            86.8
      Orange
                                            91.4
3 Watermelon
                              30
      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 --
            1.1.2
                      v readr
v dplyr
                                   2.1.4
v forcats
            1.0.0
                                   1.5.0
                      v stringr
                      v tibble
v ggplot2
            3.4.2
                                   3.2.1
                                   1.3.0
v lubridate 1.9.2
                      v tidyr
v purrr
            1.0.2
-- Conflicts -----
                                           -----ctidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
                  masks stats::lag()
x dplyr::lag()
i Use the conflicted package (<a href="http://conflicted.r-lib.org/">http://conflicted.r-lib.org/</a>) to force all conflicts to become
```

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
           1.1.2
                     v readr
                                 2.1.4
v forcats
           1.0.0
                     v stringr
                                 1.5.0
v ggplot2 3.4.2
                     v tibble
                                 3.2.1
v lubridate 1.9.2
                     v tidyr
                                 1.3.0
           1.0.2
v purrr
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()
                 masks stats::lag()
i Use the conflicted package (<a href="http://conflicted.r-lib.org/">http://conflicted.r-lib.org/</a>) to force all conflicts to become
```



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")</pre>
```

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

# 1	A tibble:	122 x 8						
	bioguide	last_name	state	party	${\tt num_votes}$	agree	agree_pred	margin_trump
	<chr></chr>	<chr></chr>	<chr></chr>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<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	${\tt Blumenthal}$	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
2 Blunt
              R
3 Brown
              D
4 Burr
              R
5 Baldwin
              D
6 Boozman
              R
7 Blackburn
8 Barrasso
9 Bennet
10 Blumenthal D
# i 112 more rows
```



-HADLEY WICKHAM

In tidy data:

- each variable forms a column
- each observation forms a row
- each cell is a single measurement

6	each column a variable									
	id	name	color							
	I		gray	each row						
	2	max	black	← an						
	3	cat	orange	Dobservation						
	4	donut	gray	2//						
	5	merlin	black	4/						
	6	panda	calico	1						

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



Figure 2.1: 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
              D
3 Brown
4 Burr
              R
5 Baldwin
              D
6 Boozman
              R
7 Blackburn
8 Barrasso
9 Bennet
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. 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:

```
1 A000360 Alexander TN
                           R
2 B000575 Blunt
                     MO
                           R
3 B000944 Brown
                     OH
                           D
4 B001135 Burr
                     NC
                           R
5 B001230 Baldwin
                     WΙ
                           D
6 B001236 Boozman
                           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></chr>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<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 \mathtt{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	${\tt Blumenthal}$	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>	<chr>></chr>	<chr>></chr>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>
Alexander	A000360	TN	R	118	0.890	0.856	26.0
Blunt	B000575	MO	R	128	0.906	0.787	18.6
Brown	B000944	OH	D	128	0.258	0.642	8.13
Burr	B001135	NC	R	121	0.893	0.560	3.66
Baldwin	B001230	WI	D	128	0.227	0.510	0.764
Boozman	B001236	AR	R	129	0.915	0.851	26.9
Blackburn	B001243	TN	R	131	0.885	0.889	26.0
Barrasso	B001261	WY	R	129	0.891	0.895	46.3
Bennet	B001267	CO	D	121	0.273	0.417	-4.91
${\tt Blumenthal}$	B001277	CT	D	128	0.203	0.294	-13.6
	<pre><chr> <chr> Alexander Blunt Brown Burr Baldwin Boozman Blackburn Barrasso Bennet</chr></chr></pre>	<chr><chr> Alexander A000360 Blunt B000575 Brown B000944 Burr B001135 Baldwin B001230 Boozman B001236 Blackburn B001243 Barrasso B001261</chr></chr>	<chr><chr><chr> Alexander A000360 TN Blunt B000575 MO Brown B000944 OH Burr B001135 NC Baldwin B001230 WI Boozman B001236 AR Blackburn B001243 TN Barrasso B001261 WY Bennet B001267 CO</chr></chr></chr>	<chr> <chr> <chr> <chr> <chr> <chr> <chr> Alexander A000360 TN R Blunt B000575 MO R Brown B000944 OH D Burr B001135 NC R Baldwin B001230 WI D Boozman B001236 AR R Blackburn B001243 TN R Barrasso B001261 WY R Bennet B001267 CO D</chr></chr></chr></chr></chr></chr></chr>	<chr> <chr> <chr> <chr> <chr> <chr> Alexander A000360 TN R 118 Blunt B000575 MO R 128 Brown B000944 OH D 128 Burr B001135 NC R 121 Baldwin B001230 WI D 128 Boozman B001236 AR R 129 Blackburn B001243 TN R 131 Barrasso B001261 WY R 129 Bennet B001267 CO D 121</chr></chr></chr></chr></chr></chr>	<chr><chr><chr><chr><chr><dbl> Alexander A000360 TN R 118 0.890 Blunt B000575 MO R 128 0.906 Brown B000944 OH D 128 0.258 Burr B001135 NC R 121 0.893 Baldwin B001230 WI D 128 0.227 Boozman B001236 AR R 129 0.915 Blackburn B001243 TN R 131 0.885 Barrasso B001261 WY R 129 0.891 Bennet B001267 CO D 121 0.273</dbl></chr></chr></chr></chr></chr>	Chr> Chr Chr

i 112 more rows



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
              <dbl>
   <chr>>
                          <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
              0.227
 5 Baldwin
                         0.510
              0.915
 6 Boozman
                         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
                                                   0.890
                                                                   0.856
                       TN
                             R
                                         118
2 B000575 Blunt
                                                                   0.787
                       MO
                             R
                                         128
                                                   0.906
3 B000944 Brown
                                                                   0.642
                       OH
                             D
                                         128
                                                   0.258
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
                                                   0.891
                                                                    0.895
                       WY
                             R
                                          129
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:

A tibble: 122 x 5 last_name agree agree_pred pct_agree pct_agree_pred <chr> <dbl> <dbl> <dbl> <dbl> 1 Alexander 0.890 89.0 85.6 0.856 2 Blunt 0.906 0.787 90.6 78.7 3 Brown 0.258 25.8 64.2 0.642 4 Burr 0.893 0.560 89.3 56.0 5 Baldwin 0.227 0.510 22.7 51.0 6 Boozman 85.1 0.915 0.851 91.5 7 Blackburn 0.885 0.889 88.5 88.9 89.1 89.5 8 Barrasso 0.891 0.895 9 Bennet 27.3 41.7 0.273 0.417 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></chr>	<dbl></dbl>	<dbl></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 121 0.273 -4.91D 0.417 4 B001277 Blumenthal CT D 128 0.203 0.294 -13.65 B001288 Booker NJ D 119 0.160 0.290 -14.1 6 C000127 Cantwell WA D 128 0.242 0.276 -15.57 C000141 Cardin MD D 128 0.25 0.209 -26.4 -11.48 C000174 Carper DΕ D 129 0.295 0.318 9 C001070 Casey PAD 129 0.287 0.508 0.724 128 0.289 10 C001088 Coons DΕ D 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)
1	"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>	<chr></chr>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<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	${\tt Blackburn}$	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	${\tt num_votes}$	agree	agree_pred	margin_trump
	<chr></chr>	<chr></chr>	<chr></chr>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<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> <chr> <chr> 1 M001183 Manchin WV D 129 0.504 0.893 42.2

i 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)
```

# 1	A tibble:	122 x 8						
	bioguide	last_name	state	party	${\tt num_votes}$	agree	agree_pred	margin_trump
	<chr></chr>	<chr></chr>	<chr></chr>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>
1	H000273	${\tt Hickenlooper}$	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> <chr> <chr> 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>	<chr></chr>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<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
                                                        0.323
                                                                     -11.0
                       \mathsf{OR}
                             D
                                         129 0.155
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
                                                                     -26.4
                                                        0.221
7 H001075 Harris
                       CA
                             D
                                         116 0.164
                                                        0.209
                                                                     -30.1
8 M000133 Markey
                                         127 0.165
                                                                     -27.2
                       MA
                             D
                                                        0.213
9 W000779 Wyden
                       OR
                                         129 0.186
                             D
                                                        0.323
                                                                     -11.0
10 B001277
           Blumenthal CT
                                         128 0.203
                                                         0.294
                                                                     -13.6
                             D
# i 105 more rows
```

i 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:

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),
```

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

i 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
<pre>mutate()</pre>	Creating columns
filter()	Filtering rows
arrange()	Ordering rows
<pre>summarize()</pre>	Summarizing data
summarize(, .by =)	Summarizing data (by groups)

Function	Purpose

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 <code>geom_bar()</code> 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 <code>geom_col()</code> geom specifying both <code>x</code> and <code>y</code> aesthetic mappings, in which is sometimes called a "column plot:"



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:



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:



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:



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:



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)")
```





quent module.

We will review a few more customization options, including text labels and facets, in a subse-

3 Matrices

Matrices are rectangular collections of numbers. In this module we will introduce them and review some basic operators, to then introduce a sneak peek of why matrices are useful (and cool).

3.1 Introduction

3.1.1 Scalars

One number (for example, 12) is referred to as a scalar.

$$a = 12$$

3.1.2 Vectors

We can put several scalars together to make a vector. Here is an example:

$$\vec{b} = \begin{bmatrix} 12\\14\\15 \end{bmatrix}$$

Since this is a column of numbers, we cleverly refer to it as a column vector.

Here is another example of a vector, this time represented as a row vector:

$$\vec{c} = \begin{bmatrix} 12 & 14 & 15 \end{bmatrix}$$

Column vectors are possibly more common and useful, but we sometimes write things down using row vectors to

Vectors are fairly easy to construct in R. As we saw before, we can use the c() function to combine elements:

```
c(5, 25, -2, 1)
```

[1] 5 25 -2 1



⚠ Warning

Remember that the code above does not create any objects. To do so, you'd need to use the assignment operator (<-):

```
vector_example <- c(5, 25, -2, 1)
vector_example
```

Or we can also create vectors from sequences with the : operator or the seq() function:

10:20

[1] 10 11 12 13 14 15 16 17 18 19 20

```
seq(from = 3, to = 27, by = 3)
```

6 9 12 15 18 21 24 27

3.2 Operators

3.2.1 Summation

The summation operator \sum (i.e., the uppercase Sigma letter) lets us perform an operation on a sequence of numbers, which is often but not always a vector.

$$\vec{d} = \begin{bmatrix} 12 & 7 & -2 & 3 & -1 \end{bmatrix}$$

We can then calculate the sum of the first three elements of the vector, which is expressed as follows:

$$\sum_{i=1}^{3} d_i$$

Then we do the following math:

$$12 + 7 + (-2) = 17$$

It is also common to use n in the superscript to indicate that we want to sum all elements:

$$\sum_{i=1}^{n} d_i = 12 + 7 + (-2) + 3 + (-1) = 19$$

We can perform these operations using the sum() function in R:

```
vector_d <- c(12, 7, -2, 3, -1)
sum(vector_d[1:3])

[1] 17
sum(vector_d)

[1] 19</pre>
```

3.2.2 Product

The product operator \prod (i.e., the uppercase Pi letter) can also perform operations over a sequence of elements in a vector. Recall our previous vector:

$$\vec{d} = \begin{bmatrix} 12 & 7 & -2 & 3 & 1 \end{bmatrix}$$

We might want to calculate the product of all its elements, which is expressed as follows:

$$\prod_{i=1}^{n} d_i = 12 \cdot 7 \cdot (-2) \cdot 3 \cdot (-1) = 504$$

In R, we can compute products using the prod() function:

```
prod(vector_d)
```

[1] 504

i Exercise

Get the product of the first three elements of vector d. Write the notation by hand and use R to obtain the number.

3.3 Matrices

3.3.1 Basics

We can append vectors together to form a matrix:

$$A = \begin{bmatrix} 12 & 14 & 15\\ 115 & 22 & 127\\ 193 & 29 & 219 \end{bmatrix}$$

The number of rows and columns of a matrix constitute the *dimensions* of the matrix. The first number is the number of rows ("r") and the second number is the number of columns ("c") in the matrix.

Important

Find a way to remember "r x c" permanently. The order of the dimensions never changes.

Matrix A above, for example, is a 3x3 matrix. Sometimes we'd refer to it as A_{3x3} .



🕊 Tip

It is common to use capital letters (sometimes **bold-faced**) to represent matrices. In contrast, vectors are usually represented with either bold lowercase letters or lowercase letters with an arrow on top (e.g., \vec{v}).

Constructing matrices in R

There are different ways to create matrices in R. One of the simplest is via rbind() or cbind(), which paste vectors together (either by rows or by columns):

```
# Create some vectors
vector1 <- 1:4
vector2 <- 5:8
vector3 <- 9:12
```

```
vector4 <- 13:16
  # Using rbind(), each vector will be a row
  rbind_mat <- rbind(vector1, vector2, vector3, vector4)</pre>
  rbind_mat
        [,1] [,2] [,3] [,4]
                2
                      3
vector1
           5
                6
                     7
vector2
                           8
vector3
           9
               10
                     11
                          12
vector4
          13
               14
                     15
                          16
  # Using cbind(), each vector will be a column
  cbind_mat <- cbind(vector1, vector2, vector3, vector4)</pre>
  cbind_mat
     vector1 vector2 vector3 vector4
[1,]
                   5
           1
                           9
                                   13
[2,]
           2
                    6
                           10
                                    14
```

11

12

An alternative is to use to properly named matrix() function. The basic syntax is matrix(data, nrow, ncol, byrow):

• data is the input vector which becomes the data elements of the matrix.

15

16

• nrow is the number of rows to be created.

7

8

- ncol is the number of columns to be created.
- byrow is a logical clue. If TRUE then the input vector elements are arranged by row. By default (FALSE), elements are arranged by column.

Let's see some examples:

3

4

[3,]

[4,]

```
# Elements are arranged sequentially by row.
M <- matrix(c(1:12), nrow = 4, byrow = T)
M</pre>
```

```
[3,] 7 8 9 [4,] 10 11 12
```

```
# Elements are arranged sequentially by column (byrow = F by default). N <- matrix(c(1:12), nrow = 4) N
```

	[,1]	[,2]	[,3]
[1,]	1	5	9
[2,]	2	6	10
[3,]	3	7	11
[4,]	4	8	12

3.3.2 Structure

How do we refer to specific elements of the matrix? For example, matrix A is an $m \times n$ matrix where m = n = 3. This is sometimes called a *square matrix*.

More generally, matrix B is an $m \times n$ matrix where the elements look like this:

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \dots & b_{mn} \end{bmatrix}$$

Thus b_{23} refers to the second unit down and third across. More generally, we refer to row indices as i and to column indices as j.

In R, we can access a matrix's elements using square brackets:

```
# In matrix N, access the element at 1st row and 3rd column. N[1,3]
```

[1] 9

```
# In matrix N, access the element at 4th row and 2nd column. \text{N}\left[4,2\right]
```

[1] 8

Tip

When trying to identify a specific element, the first subscript is the element's row and the second subscript is the element's column (always in that order).

3.4 Matrix operations

3.4.1 Addition and subtraction

- Addition and subtraction are straightforward operations.
- Matrices must have *exactly* the same dimensions for both of these operations.
- We add or subtract each element with the corresponding element from the other matrix.
- This is expressed as follows:

$$A + B = C$$

$$c_{ij} = a_{ij} \pm b_{ij} \ \forall i,j$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \pm \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} & a_{13} \pm b_{13} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} & a_{23} \pm b_{23} \\ a_{31} \pm b_{31} & a_{32} \pm b_{32} & a_{33} \pm b_{33} \end{bmatrix}$$

Addition and subtraction in R

We start by creating two 2x3 matrices:

```
# Create two 2x3 matrices.
matrix1 \leftarrow matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
matrix1
```

We can simply use the + and - operators for addition and substraction:

i Exercise

(Use code for one of these and do the other one by hand!)

1) Calculate A + B

$$A = \begin{bmatrix} 1 & 0 \\ -2 & -1 \end{bmatrix}$$
$$B = \begin{bmatrix} 5 & 1 \\ 2 & -1 \end{bmatrix}$$

$$B = \begin{bmatrix} 5 & 1 \\ 2 & -1 \end{bmatrix}$$

2) Calculate A - B

$$A = \begin{bmatrix} 6 & -2 & 8 & 12 \\ 4 & 42 & 8 & -6 \end{bmatrix}$$
$$B = \begin{bmatrix} 18 & 42 & 3 & 7 \\ 0 & -42 & 15 & 4 \end{bmatrix}$$

3.4.2 Scalar multiplication

Scalar multiplication is very intuitive. As we know, a scalar is a single number. We multiply each value in the matrix by the scalar to perform this operation.

Formally, this is expressed as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$cA = \begin{bmatrix} ca_{11} & ca_{12} & ca_{13} \\ ca_{21} & ca_{22} & ca_{23} \\ ca_{31} & ca_{32} & ca_{33} \end{bmatrix}$$

In R, all we need to do is take an established matrix and multiply it by some scalar:

matrix1 from our previous example
matrix1

matrix1 * 3

i Exercise

Calculate $2 \times A$ and $-3 \times B$. Again, do one by hand and the other one using R.

$$A = \begin{bmatrix} 1 & 4 & 8 \\ 0 & -1 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} -15 & 1 & 5\\ 2 & -42 & 0\\ 7 & 1 & 6 \end{bmatrix}$$

3.4.3 Matrix multiplication

- Multiplying matrices is slightly trickier than multiplying scalars.
- Two matrices must be *conformable* for them to be multiplied together. This means that the number of columns in the first matrix equals the number of rows in the second.
- When multiplying $A \times B$, if A is $m \times n$, B must have n rows.

Important

The conformability requirement never changes. Before multiplying anything, check to make sure the matrices are indeed conformable.

• The resulting matrix will have the same number of rows as the first matrix and the number of columns in the second. For example, if A is $i \times k$ and B is $k \times j$, then $A \times B$ will be $i \times j$.

Which of the following can we multiply? What will be the dimensions of the resulting matrix?

$$B = \begin{bmatrix} 2\\3\\4\\1 \end{bmatrix} M = \begin{bmatrix} 1 & 0 & 2\\1 & 2 & 4\\2 & 3 & 2 \end{bmatrix} L = \begin{bmatrix} 6 & 5 & -1\\1 & 4 & 3 \end{bmatrix}$$

Why can't we multiply in the opposite order?

Warning

When multiplying matrices, order matters. Even if multiplication is possible in both directions, in general $AB \neq BA$.

Multiplication steps

• Multiply each row by each column, summing up each pair of multiplied terms.



🕊 Tip

This is sometimes to referred to as the "dot product," where we multiply matching members, then sum up.

• The element in position ij is the sum of the products of elements in the ith row of the first matrix (A) and the corresponding elements in the jth column of the second matrix (B).

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Example

Suppose a company manufactures two kinds of furniture: chairs and sofas.

- A chair costs \$100 for wood, \$270 for cloth, and \$130 for feathers.
- Each sofa costs \$150 for wood, \$420 for cloth, and \$195 for feathers.

	Chair	Sofa
Wood	100	150
Cloth	270	420
Feathers	130	195

The same information about unit cost (C) can be presented as a matrix.

$$C = \begin{bmatrix} 100 & 150 \\ 270 & 420 \\ 130 & 195 \end{bmatrix}$$

Note that each of the three rows of this 3 x 2 matrix represents a material (wood, cloth, or feathers), and each of the two columns represents a product (chair or coach). The elements are the unit cost (in USD).

Now, suppose that the company will produce 45 chairs and 30 sofas this month. This production quantity can be represented in the following table, and also as a 2 x 1 matrix (Q):

Quantity
45
30

$$Q = \begin{bmatrix} 45\\30 \end{bmatrix}$$

What will be the company's total cost? The "total expenditure" is equal to the "unit cost" times the "production quantity" (the number of units).

The total expenditure (E) for each material this month is calculated by multiplying these two matrices.

$$E = CQ = \begin{bmatrix} 100 & 150 \\ 270 & 420 \\ 130 & 195 \end{bmatrix} \begin{bmatrix} 45 \\ 30 \end{bmatrix} = \begin{bmatrix} (100)(45) + (150)(30) \\ (270)(45) + (420)(30) \\ (130)(45) + (195)(30) \end{bmatrix} = \begin{bmatrix} 9,000 \\ 24,750 \\ 11,700 \end{bmatrix}$$

Multiplying the 3x2 Cost matrix (C) times the 2x1 Quantity matrix (Q) yields the 3x1 Expenditure matrix (E).

As a result of this matrix multiplication, we determine that this month the company will incur expenditures of:

- \$9,000 for wood
- \$24,750 for cloth
- \$11,700 for feathers.

Matrix multiplication in R

Before attempting matrix multiplication, we must make sure the matrices are conformable (as we do for our manual calculations).

Then we can multiply our matrices together using the %*% operator.

```
C \leftarrow matrix(c(100, 270, 130, 150, 420, 195), nrow = 3)
     [,1] [,2]
[1,] 100 150
[2,]
      270
           420
[3,] 130
           195
  Q \leftarrow matrix(c(45, 30), nrow = 2)
     [,1]
[1,]
       45
[2,]
       30
  C %*% Q
      [,1]
[1,] 9000
[2,] 24750
[3,] 11700
```

⚠ Warning

If you have a missing value or NA in one of the matrices you are trying to multiply (something we will discuss in further detail in the next module), you will have NAs in your resulting matrix.

3.4.4 Properties of operations

- Addition and subtraction:
 - Associative: $(A \pm B) \pm C = A \pm (B \pm C)$
 - Communicative: $A \pm B = B \pm A$

• Multiplication:

$$-AB \neq BA$$

$$-A(BC) = (AB)C$$

$$-A(B+C) = AB + AC$$

$$-(A+B)C = AC + BC$$

3.5 Special matrices

Square matrix

- In a square matrix, the number of rows equals the number of columns (m = n):
- The *diagonal* of a matrix is a set of numbers consisting of the elements on the line from the upper-left-hand to the lower-right-hand corner of the matrix. Diagonals are particularly useful in square matrices.
- The *trace* of a matrix, denoted as tr(A), is the sum of the diagonal elements of the matrix.

Diagonal matrix:

• In a diagonal matrix, all of the elements of the matrix that are not on the diagonal are equal to zero.

Scalar matrix:

• A scalar matrix is a diagonal matrix where the diagonal elements are all equal to each other. In other words, we're really only concerned with one scalar (or element) held in the diagonal.

Identity matrix:

- The identity matrix is a scalar matrix with all of the diagonal elements equal to one.
- Remember that, as with all diagonal matrices, the off-diagonal elements are equal to zero.
- The capital letter I is reserved for the identity matrix. For convenience, a 3x3 identity matrix can be denoted as I_3 .

3.6 Transpose

The transpose is the original matrix with the rows and the columns interchanged.

The notation is either J' ("J prime") or J^T ("J transpose").

$$J = \begin{bmatrix} 4 & 5 \\ 3 & 0 \\ 7 & -2 \end{bmatrix}$$

$$J' = J^T = \begin{bmatrix} 4 & 3 & 7 \\ 5 & 0 & -2 \end{bmatrix}$$

In R, we use t() to get the transpose.

$$[3,]$$
 7 -2

3.7 Inverse

- Just like a number has a reciprocal, a matrix has an inverse.
- When we multiply a matrix by its inverse we get the identity matrix (which is like "1" for matrices).

$$A \times A^{-1} = I$$

• The inverse of A is A^{-1} only when:

$$AA^{-1} = A^{-1}A = I$$

• Sometimes there is no inverse at all.

Note

For now, don't worry about calculating the inverse of a matrix manually. This is the type of task we use R for.

• In R, we use the solve() function to calculate the inverse of a matrix:

```
A <- matrix(c(3, 2, 5, 2, 3, 2, 5, 2, 4), ncol = 3)
A
```

solve(A)

3.8 Linear systems and matrices

- A system of equations can be represented by an augmented matrix.
- System of equations:

$$3x + 6y = 12$$
$$5x + 10y = 25$$

• In an augmented matrix, each row represents one equation in the system and each column represents a variable or the constant terms.

$$\begin{bmatrix} 3 & 6 & 12 \\ 5 & 10 & 25 \end{bmatrix}$$

3.9 OLS and matrices

- We can use the logic above to calculate estimates for our ordinary least squares (OLS) models.
- OLS is a linear regression technique used to find the best-fitting line for a set of data points (observations) by minimizing the residuals (the differences between the observed and predicted values).
- We minimize the sum of the squared errors.

3.9.1 Dependent variable

- Suppose, for example, we have a sample consisting of n observations.
- The dependent variable is denoted as an $n \times 1$ column vector.

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

3.9.2 Independent variables

- Suppose there are k independent variables and a constant term, meaning k+1 columns and n rows.
- We can represent these variables as an $n \times (k+1)$ matrix, expressed as follows:

$$X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1k} \\ 1 & x_{21} & \dots & x_{2k} \\ \vdots & \vdots & \dots & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{bmatrix}$$

• x_{ij} is the *i*-th observation of the *j*-th independent variable.

3.9.3 Linear regression model

- Let's say we have 173 observations (n = 173) and 2 IVs (k = 3).
- This can be expressed as the following linear equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

• In matrix form, we have:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{21} \\ 1 & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{1173} & x_{2173} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_{173} \end{bmatrix}$$

• All 173 equations can be represented by:

$$y = X\beta + \epsilon$$

3.9.4 Estimates

• Without getting too much into the mechanics, we can calculate our coefficient estimates with matrix algebra using the following equation:

$$\hat{\beta} = (X'X)^{-1}X'Y$$

- Read aloud, we say "X prime X inverse, X prime Y".
- The little hat on our beta $(\hat{\beta})$ signifies that these are estimates, that is our OLS estimators.
- Remember, the OLS method is to choose $\hat{\beta}$ such that the sum of squared residuals ("SSR") is minimized.

3.9.4.1 Example in R

• We will load the mtcars data set (our favorite) for this example, which contains data about many different car models.

• Now, we want to estimate the association between hp (horsepower) and wt (weight), our independent variables, and mpg (miles per gallon), our dependent variable.

• First, we transform our dependent variable into a matrix, using the as.matrix function and specifying the column of the mtcars data set to create a column vector of our observed values for the DV.

```
Y <- as.matrix(cars_df$mpg)</pre>
      [,1]
 [1,] 21.0
 [2,] 21.0
 [3,] 22.8
 [4,] 21.4
 [5,] 18.7
 [6,] 18.1
 [7,] 14.3
 [8,] 24.4
 [9,] 22.8
[10,] 19.2
[11,] 17.8
[12,] 16.4
[13,] 17.3
[14,] 15.2
[15,] 10.4
[16,] 10.4
[17,] 14.7
[18,] 32.4
[19,] 30.4
[20,] 33.9
[21,] 21.5
[22,] 15.5
[23,] 15.2
[24,] 13.3
[25,] 19.2
[26,] 27.3
[27,] 26.0
[28,] 30.4
[29,] 15.8
[30,] 19.7
[31,] 15.0
[32,] 21.4
```

• Next, we do the same thing for our independent variables of interest, and our constant.

```
# create two separate matrices for IVs
X1 <- as.matrix(cars_df$hp)
X2 <- as.matrix(cars_df$wt)

# create constant column

# bind them altogether into one matrix
constant <- rep(1, nrow(cars_df))
X <- cbind(constant, X1, X2)
X</pre>
```

constant

```
[1,]
            1 110 2.620
 [2,]
            1 110 2.875
 [3,]
            1 93 2.320
 [4,]
            1 110 3.215
 [5,]
            1 175 3.440
 [6,]
            1 105 3.460
 [7,]
            1 245 3.570
 [8,]
            1 62 3.190
 [9,]
            1 95 3.150
[10,]
            1 123 3.440
[11,]
            1 123 3.440
[12,]
            1 180 4.070
            1 180 3.730
[13,]
[14,]
             1 180 3.780
[15,]
            1 205 5.250
[16,]
             1 215 5.424
[17,]
            1 230 5.345
            1 66 2.200
[18,]
[19,]
            1 52 1.615
[20,]
             1 65 1.835
[21,]
             1 97 2.465
[22,]
            1 150 3.520
[23,]
             1 150 3.435
            1 245 3.840
[24,]
            1 175 3.845
[25,]
[26,]
            1 66 1.935
[27,]
            1 91 2.140
[28,]
            1 113 1.513
[29,]
            1 264 3.170
[30,]
            1 175 2.770
```

```
[31,] 1 335 3.570
[32,] 1 109 2.780
```

• Next, we calculate X'X, X'Y, and $(X'X)^{-1}$.

Don't forget to use **%*%** for matrix multiplication!

```
# X prime X
XpX <- t(X) %*% X

# X prime X inverse
XpXinv <- solve(XpX)

# X prime Y
XpY <- t(X) %*% Y

# beta coefficient estimates
bhat <- XpXinv %*% XpY
bhat</pre>
```

[,1] constant 37.22727012 -0.03177295 -3.87783074

4 Tidy data analysis II

library(tidyverse)

In this session, we'll cover a few more advanced topics related to data wrangling. Again we'll use the tidyverse:

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr
          1.1.2
                    v readr
                               2.1.4
v forcats 1.0.0
                    v stringr
                               1.5.0
v ggplot2 3.4.2
                    v tibble
                               3.2.1
v lubridate 1.9.2
                    v tidyr
                               1.3.0
          1.0.2
v purrr
```

```
-- 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 become

4.1 Loading data in different formats.

In this module we will use cross-national data from the Quality of Government (QoG) project (Dahlberg et al., 2023).

Notice how in the data/ folder we have multiple versions of the same dataset (a subset of the QOG basic dataset): .csv (comma-separated values), .rds (R), .xlsx (Excel), .dta (Stata), and .sav (SPSS).

4.1.1 CSV and R data files

We can use the read_csv() and read_rds() functions from the tidyverse¹ to read the .csv and .rds (R) data files:

¹Technically, the read_csv() and read_rds() functions come from readr, one of the tidyverse constituent packages.

```
qog_csv <- read_csv("data/sample_qog_bas_ts_jan23.csv")

Rows: 1085 Columns: 8
-- Column specification ------
Delimiter: ","
chr (4): cname, ccodealp, region, ht_colonial
dbl (4): year, wdi_pop, vdem_polyarchy, vdem_corr

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.

qog rds <- read rds("data/sample qog bas ts jan23.rds")</pre>
```

For reading files from other software (Excel, Stata, or SPSS), we need to load additional packages. Luckily, they are automatically installed when one installs the tidyverse.

4.1.2 Excel data files

For Excel files (.xls or .xlsx files), the readxl package has a handy read_excel() function.

```
library(readxl)
qog_excel <- read_excel("data/sample_qog_bas_ts_jan23.xlsx")</pre>
```



Useful arguments of the read_excel() function include sheet =, which reads particular sheets (specified via their positions or sheet names), and range =, which extracts a particular cell range (e.g., 'A5:E25').

4.1.3 Stata and SPSS data files

To load files from Stata (.dta) or SPSS (.spss), one needs the haven package and its properly-named read_stata() and read_spss() functions:

```
library(haven)
qog_stata <- read_stata("data/sample_qog_bas_ts_jan23.dta")
qog_spss <- read_spss("data/sample_qog_bas_ts_jan23.sav")</pre>
```



Datasets from Stata and SPSS can have additional properties, like variable labels and special types of missing values. To learn more about this, check out the "Labelled data" chapter from Danny Smith's $Survey\ Research\ Datasets\ and\ R\ (2020)$.

4.1.4 Our data for this session

We will rename one of our objects to qog:

```
qog <- qog_csv
qog</pre>
```

```
# A tibble: 1,085 x 8
   cname
              ccodealp
                         year region wdi_pop vdem_polyarchy vdem_corr ht_colonial
   <chr>
              <chr>
                        <dbl> <chr>
                                        <dbl>
                                                        <dbl>
                                                                   <dbl> <chr>
                         1990 Carib~
                                        63328
                                                                      NA British
 1 Antigua a~ ATG
                                                           NA
2 Antigua a~ ATG
                         1991 Carib~
                                        63634
                                                           NA
                                                                      NA British
                         1992 Carib~
3 Antigua a~ ATG
                                        64659
                                                           NA
                                                                      NA British
4 Antigua a~ ATG
                         1993 Carib~
                                                           NA
                                                                      NA British
                                        65834
5 Antigua a~ ATG
                         1994 Carib~
                                        67072
                                                           NA
                                                                      NA British
6 Antigua a~ ATG
                         1995 Carib~
                                        68398
                                                           NA
                                                                      NA British
7 Antigua a~ ATG
                         1996 Carib~
                                        69798
                                                           NA
                                                                      NA British
8 Antigua a~ ATG
                         1997 Carib~
                                        71218
                                                                      NA British
                                                           NA
9 Antigua a~ ATG
                         1998 Carib~
                                        72572
                                                           NA
                                                                      NA British
10 Antigua a~ ATG
                         1999 Carib~
                                        73821
                                                           NA
                                                                      NA British
# i 1,075 more rows
```

This dataset is a small sample of QOG, which contains data for countries in the Americas from 1990 to 2020. The observational unit is thus country-year. You can access the full codebook online. The variables are as follows:

Variable	Description
cname	Country name
ccodealp	Country code (ISO-3 character convention)
year	Year
region	Region (following legacy WDI convention). Added to QOG by
	us.
wdi_pop	Total population, from the World Development Indicators
vdem_polyarchy	V-Dem's polyarchy index (electoral democracy)

Variable	Description			
vdem_corr	V-Dem's corruption index			
ht_colonial	Former colonial ruler			

4.2 Recoding variables

Take a look at the ht_colonial variable. We can do a simple tabulation with count():

```
qog |>
    count(ht_colonial)
# A tibble: 6 x 2
 ht_colonial
  <chr>>
                   <int>
1 British
                     372
2 Dutch
                      31
                      31
3 French
4 Never colonized
                      62
5 Portuguese
                      31
6 Spanish
                     558
```

We might want to recode this variable. For instance, we could create a *dummy/binary* variable for whether the country was a British colony. We can do this with <code>if_else()</code>, which works with logical conditions:

Instead of a numeric classification (0 and 1), we could use characters:

if_else() is great for binary recoding. But sometimes we want to create more than two
categories. We can use case_when():

```
qog |>
    # syntax is condition ~ value
    mutate(cat_col = case_when(
      ht_colonial == "British" ~ "British",
      ht_colonial == "Spanish" ~ "Spanish",
      .default = "Other" # what to do in all other cases
    )) |>
    count(cat_col)
# A tibble: 3 x 2
 cat_col
 <chr> <int>
1 British 372
2 Other
           155
3 Spanish
           558
```

The .default = argument in case_when() can also be used to leave the variable as-is for non-specified cases. For example, let's combine Portuguese and Spanish colonies:

```
qog |>
    # syntax is condition ~ value
    mutate(cat_col = case_when(
        ht_colonial %in% c("Spanish", "Portuguese") ~ "Spanish/Portuguese",
        .default = ht_colonial # what to do in all other cases
)) |>
    count(cat_col)
```

```
# A tibble: 5 x 2
  cat_col
                           n
  <chr>
                      <int>
1 British
                        372
2 Dutch
                         31
3 French
                          31
4 Never colonized
                          62
5 Spanish/Portuguese
                        589
```

i Exercise

- 1. Create a dummy variable, d_large_pop, for whether the country-year has a population of more than 1 million. Then compute its mean. Your code:
- 2. Which countries are recorded as "Never colonized"? Change their values to other reasonable codings and compute a tabulation with count(). Your code:

4.3 Missing values

Missing values are commonplace in real datasets. In R, missing values are a special type of value in vectors, denoted as NA.

Warning

The special value NA is different from the character value "NA". For example, notice that a numeric vector can have NAs, while it obviously cannot hold the character value "NA":

```
c(5, 4.6, NA, 8)
```

[1] 5.0 4.6 NA 8.0

A quick way to check for missing values in small datasets is with the summary() function:

summary(qog)

cname	ccodealp	year	region	
Length: 1085	Length:1085	Min. :1990	Length: 1085	
Class :character	Class :character	1st Qu.:1997	Class :character	
Mode :character	Mode :character	Median :2005	Mode :character	
		Mean :2005		

3rd Qu.:2013 Max. :2020

```
wdi_pop
                    vdem_polyarchy
                                        vdem_corr
                                                        ht_colonial
                           :0.0710
                                                        Length: 1085
Min.
            40542
                    Min.
                                      Min.
                                              :0.0260
1st Qu.:
           389131
                    1st Qu.:0.5570
                                      1st Qu.:0.1890
                                                        Class : character
Median: 5687744
                    Median :0.7030
                                      Median :0.5550
                                                        Mode :character
Mean
       : 25004057
                    Mean
                            :0.6569
                                      Mean
                                              :0.4922
3rd Qu.: 16195902
                    3rd Qu.:0.8030
                                      3rd Qu.:0.7540
Max.
       :331501080
                    Max.
                            :0.9160
                                      Max.
                                              :0.9630
                    NA's
                                      NA's
                            :248
                                              :248
```

Notice that we have missingness in the vdem_polyarchy and vdem_corr variables. We might want to filter the dataset to see which observations are in this situation:

```
qog |>
   filter(vdem_polyarchy == NA | vdem_corr == NA)

# A tibble: 0 x 8
# i 8 variables: cname <chr>, ccodealp <chr>, year <dbl>, region <chr>,
# wdi_pop <dbl>, vdem_polyarchy <dbl>, vdem_corr <dbl>, ht_colonial <chr>>
```

But the code above doesn't work! To refer to missing values in logical conditions, we cannot use == NA. Instead, we need to use the is.na() function:

```
qog |>
  filter(is.na(vdem_polyarchy) | is.na(vdem_corr))
```

A tibble: 248 x 8

	cname	ccodealp	year	region	wdi_pop	vdem_polyarchy	vdem_corr	ht_colonial
	<chr></chr>	<chr></chr>	<dbl></dbl>	<chr></chr>	<dbl></dbl>	<dbl></dbl>	<dbl></dbl>	<chr></chr>
1	Antigua a~	ATG	1990	Carib~	63328	NA	NA	British
2	Antigua a~	ATG	1991	Carib~	63634	NA	NA	British
3	Antigua a~	ATG	1992	Carib~	64659	NA	NA	British
4	Antigua a~	ATG	1993	Carib~	65834	NA	NA	British
5	Antigua a~	ATG	1994	Carib~	67072	NA	NA	British
6	Antigua a~	ATG	1995	Carib~	68398	NA	NA	British
7	Antigua a~	ATG	1996	Carib~	69798	NA	NA	British
8	Antigua a~	ATG	1997	Carib~	71218	NA	NA	British
9	Antigua a~	ATG	1998	Carib~	72572	NA	NA	British

```
10 Antigua a~ ATG 1999 Carib~ 73821 NA NA British # i 238 more rows
```

Notice that, in most R functions, missing values are "contagious." This means that any missing value will contaminate the operation and carry over to the results. For example:

Sometimes we'd like to perform our operations even in the presence of missing values, simply excluding them. Most basic R functions have an na.rm = argument to do this:

i Exercise

Calculate the median value of the corruption variable for each region (i.e., perform a grouped summary). Your code:

4.4 Pivoting data

We will now load another time-series cross-sectional dataset, but in a slightly different format. It's adapted from the World Bank's World Development Indicators (WDI) (2023) and records gross domestic product at purchasing power parity (GDP PPP).

```
gdp <- read_excel("data/wdi_gdp_ppp.xlsx")</pre>
```

```
# A tibble: 266 x 35
                                       `1990`
                                                `1991`
                                                          1992
                                                                   1993
                                                                            1994
   country_name
                       country_code
   <chr>
                       <chr>
                                        <dbl>
                                                 <dbl>
                                                           <dbl>
                                                                    <dbl>
                                                                             <dbl>
 1 Aruba
                       ABW
                                      2.03e 9
                                               2.19e 9
                                                        2.32e 9
                                                                 2.48e 9
                                                                           2.69e 9
                                                        9.23e11
 2 Africa Eastern and~ AFE
                                      9.41e11
                                               9.42e11
                                                                 9.19e11
                                                                           9.35e11
3 Afghanistan
                       AFG
                                     NA
                                              NΔ
                                                       NA
                                                                 NA
                                                                          MΛ
                                      5.76e11
                                                        5.98e11 5.92e11
4 Africa Western and~ AFW
                                               5.84e11
                                                                           5.91e11
                                      6.85e10
                                               6.92e10
                                                        6.52e10 4.95e10
                                                                           5.02e10
5 Angola
                       AGO
6 Albania
                       ALB
                                      1.59e10
                                               1.14e10
                                                        1.06e10
                                                                1.16e10
                                                                           1.26e10
7 Andorra
                       AND
                                     NA
                                              NA
                                                       NA
                                                                 NA
                                                                          NA
8 Arab World
                       ARB
                                      2.19e12
                                               2.25e12
                                                        2.35e12
                                                                 2.41e12
                                                                           2.48e12
9 United Arab Emirat~ ARE
                                      2.01e11
                                               2.03e11
                                                       2.10e11 2.12e11
                                      4.61e11 5.04e11 5.43e11 5.88e11
10 Argentina
                       ARG
# i 256 more rows
# i 28 more variables: `1995` <dbl>, `1996` <dbl>, `1997` <dbl>, `1998` <dbl>,
    `1999` <dbl>, `2000` <dbl>, `2001` <dbl>, `2002` <dbl>, `2003` <dbl>,
    `2004` <dbl>, `2005` <dbl>, `2006` <dbl>, `2007` <dbl>, `2008` <dbl>,
    `2009` <dbl>, `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>,
#
    `2014` <dbl>, `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, `2018` <dbl>,
    `2019` <dbl>, `2020` <dbl>, `2021` <dbl>, `2022` <dbl>
```

Note how the information is recorded differently. Here columns are not variables, but years. We call datasets like this one **wide**, in contrast to the **long** datasets we have seen before. In general, R and the **tidyverse** work much nicer with long datasets. Luckily, the **tidyr** package of the **tidyverse** makes it easy to convert datasets between these two formats.

We will use the pivot longer() function:



Figure 4.1: Source: Illustration by Allison Horst, adapted by Peter Higgins.

```
2 Aruba
                ABW
                               1991 2186758474.
3 Aruba
                               1992 2315391348.
                ABW
4 Aruba
                ABW
                               1993 2484593045.
5 Aruba
                               1994 2688426606.
                ABW
6 Aruba
                ABW
                               1995 2756904694.
                               1996 2789595753.
7 Aruba
                ABW
8 Aruba
                ABW
                               1997 2986175079.
9 Aruba
                ABW
                               1998 3045659222.
10 Aruba
                               1999 3083365758.
                ABW
# i 8,768 more rows
```

Done! This is a much friendlier format to work with. For example, we can now do summaries:

```
gdp_long |>
  summarize(mean_gdp_ppp = mean(wdi_gdp_ppp, na.rm = T), .by = country_name)
```

A tibble: 266 x 2 country_name mean_gdp_ppp <chr> <dbl> 3.38e 9 1 Aruba 2 Africa Eastern and Southern 1.61e12 3 Afghanistan 5.56e10 4 Africa Western and Central 1.15e12 5 Angola 1.38e11 6 Albania 2.56e10 7 Andorra NaN 4.22e12 8 Arab World

i 256 more rows

9 United Arab Emirates

i Exercise

10 Argentina

Convert back gdp_long to a wide format using pivot_wider(). Check out the help file using ?pivot_wider. Your code:

4.29e11

8.06e11

4.5 Merging datasets

It is extremely common to want to integrate data from multiple sources. Combining information from two datasets is called *merging* or *joining*.

To do this, we need ID variables in common between the two data sets. Using our QOG and WDI datasets, these variables will be country code (which in this case is shared between the two datasets) and year.



Standardized unit codes (like country codes) are extremely useful when merging data. It's harder than expected for a computer to realize that "Bolivia (Plurinational State of)" and "Bolivia" refer to the same unit. By default, these units will not be matched.²

Okay, now to the merging. Imagine we want to add information about GDP to our QOG main dataset. To do so, we can use the left_join() function, from the tidyverse's dplyr package:

```
qog_plus <- left_join(qog, # left data frame, which serves as a "base"
                         gdp_long, # right data frame, from which to draw new columns
                         by = c("ccodealp" = "country_code", # can define name equivalencies!
                                "vear"))
  qog_plus |>
    # select variables for clarity
    select(cname, ccodealp, year, wdi_pop, wdi_gdp_ppp)
# A tibble: 1,085 x 5
   cname
                       ccodealp
                                 year wdi_pop wdi_gdp_ppp
   <chr>
                       <chr>>
                                 <dbl>
                                         <dbl>
                                                      <dbl>
 1 Antigua and Barbuda ATG
                                         63328
                                  1990
                                                966660878.
2 Antigua and Barbuda ATG
                                  1991
                                         63634
                                                987701012.
3 Antigua and Barbuda ATG
                                  1992
                                         64659
                                                999143284.
4 Antigua and Barbuda ATG
                                         65834 1051896837.
                                  1993
5 Antigua and Barbuda ATG
                                  1994
                                         67072 1122128908.
6 Antigua and Barbuda ATG
                                  1995
                                         68398 1073208718.
7 Antigua and Barbuda ATG
                                  1996
                                         69798 1144088355.
8 Antigua and Barbuda ATG
                                  1997
                                         71218 1206688391.
9 Antigua and Barbuda ATG
                                  1998
                                         72572 1263778328.
10 Antigua and Barbuda ATG
                                  1999
                                         73821 1310634399.
# i 1,075 more rows
```

²There are R packages to deal with these complications. **fuzzyjoin** matches units by their approximate distance, using some clever algorithms. **countrycode** allows one to standardize country names and country codes across different conventions.

🕊 Tip

Most of the time, you'll want to do a left_join(), which is great for adding new information to a "base" dataset, without dropping information from the latter. In limited situations, other types of joins can be helpful. To learn more about them, you can read Jenny Bryan's excellent tutorial on dplyr joins.

i Exercise

There is a dataset on country's CO2 emissions, again from the World Bank (2023), in "data/wdi_co2.csv". Load the dataset into R and add a new variable with its information, wdi_co2, to our qog_plus data frame. Finally, compute the average values of CO2 emissions per capita, by country. Tip: this exercise requires you to do many steps—plan ahead before you start coding! Your code:

4.6 Plotting extensions: trend graphs, facets, and customization

i Exercise

Draw a scatterplot with time in the x-axis and democracy scores in the y-axis. Your code:

How can we visualize trends effectively? One alternative is to use a trend graph. Let's start by computing the yearly averages for democracy in the whole region:

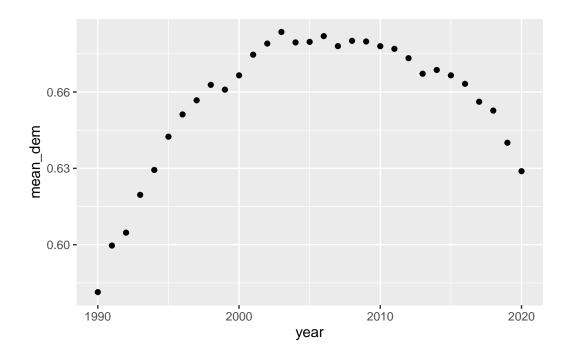
```
dem_yearly <- qog |>
  summarize(mean_dem = mean(vdem_polyarchy, na.rm = T), .by = year)
dem_yearly
```

A tibble: 31 x 2 year mean_dem <dbl> <dbl> 1 1990 0.581 2 1991 0.600 1992 0.605 4 1993 0.620 5 1994 0.629 6 1995 0.642 7 1996 0.651 8 1997 0.657

```
9 1998 0.663
10 1999 0.661
# i 21 more rows
```

Now we can plot them with a scatterplot:

```
ggplot(dem_yearly, aes(x = year, y = mean_dem)) +
  geom_point()
```



We can add geom_line() to connect the dots:

```
ggplot(dem_yearly, aes(x = year, y = mean_dem)) +
  geom_point() +
  geom_line()
```



We can, of course, remove to points to only keep the line:

```
ggplot(dem_yearly, aes(x = year, y = mean_dem)) +
   geom_line()
```



What if we want to plot trends for different countries? We can use the group and color aesthetic mappings (no need to do a summary here! data is already at the country-year level):

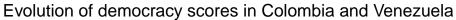
```
# filter to only get Colombia and Venezuela
dem_yearly_countries <- qog |>
   filter(ccodealp %in% c("COL", "VEN"))

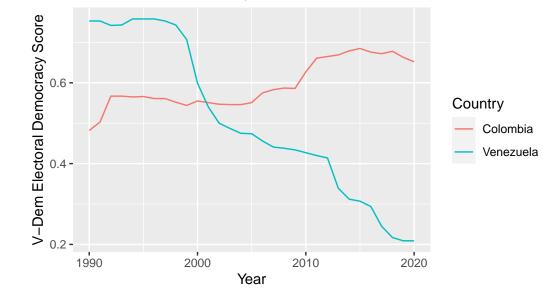
ggplot(dem_yearly_countries, aes(x = year, y = vdem_polyarchy, color = cname)) +
   geom_line()
```



Remember that we can use the labs() function to add labels:

```
ggplot(dem_yearly_countries, aes(x = year, y = vdem_polyarchy, color = cname)) +
   geom_line() +
   labs(x = "Year", y = "V-Dem Electoral Democracy Score", color = "Country",
        title = "Evolution of democracy scores in Colombia and Venezuela",
        caption = "Source: V-Dem (Coppedge et al., 2022) in QOG dataset.")
```





Source: V-Dem (Coppedge et al., 2022) in QOG dataset.

Another way to display these trends is by using *facets*, which divide a plot into small boxes according to a categorical variable (no need to add color here):

```
ggplot(dem_yearly_countries, aes(x = year, y = vdem_polyarchy)) +
   geom_line() +
   facet_wrap(~cname)
```



Facets are particularly useful for many categories (where the number of distinguishable colors reaches its limit):

```
ggplot(qog |> filter(region == "South America"),
    aes(x = year, y = vdem_polyarchy)) +
    geom_line() +
    facet_wrap(~cname)
```



With facets, one can control whether each facet picks its own scales or if all facets share the same scale. For example, let's plot the populations of Canada and the US:

```
ggplot(qog |> filter(cname %in% c("Canada", "United States")),
    aes(x = year, y = wdi_pop)) +
    geom_line() +
    facet_wrap(~cname)
```

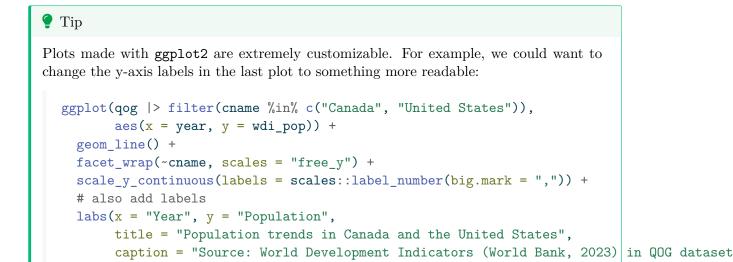


The scales are so disparate that unifying them yields a plot that's hard to interpret. But if we're interested in within-country trends, we can let each facet have its own scale with the scales = argument (with can be "fixed", "free_x", "free_y", or "free"):

```
ggplot(qog |> filter(cname %in% c("Canada", "United States")),
    aes(x = year, y = wdi_pop)) +
    geom_line() +
    facet_wrap(~cname, scales = "free_y")
```



This ability to visualize within time trends also makes facets appealing in many situations.



Population trends in Canada and the United States



Source: World Development Indicators (World Bank, 2023) in QOG dataset.

While it's impossible for us to review all the customization options you might need, a fantastic reference is the "ggplot2: Elegant Graphics for Data Analysis" book by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen.

i Exercise

Using your merged dataset from the previous section, plot the trajectories of C02 per capita emissions for the US and Haiti. Use adequate scales.

5 Functions

5.1 Basics

5.1.1 What is a function?

Informally, a function is anything that takes input(s) and gives one defined output. There are always three main parts:

- The input (x values, or each value in the domain)
- The relationship of interest
- The output (y values, or a unique value in the range)

Note

" $f(x) = \dots$ is the classic notation for writing a function, but we can also use" $y = \dots$ ". This is because y is a function of x, so y = f(x).

Let's take a look at an example and break down the structure:

$$f(x) = 3x + 4$$

- x is the *input* (some value) that the function takes.
- For any x, we multiply by three and add 4, which is the *relationship*.
- Finally, f(x) or y is the unique result, or the *output*.

The most common name to give a function is, predictably, "f", but we can have other names such as "g" or "h". The choice is yours.

Important

When reading out loud, we say "[name of function] of x equals [relationship]. For example, $f(x) = x^2$ is referred to as for x equals x squared."

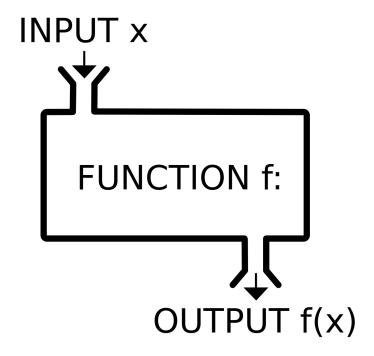
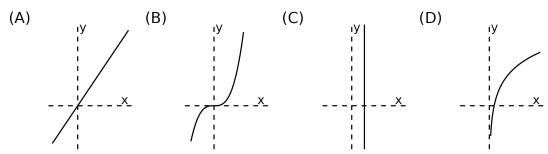


Figure 5.1: Function machine. Source: Bill Bailey on Wikimedia Commons.

5.1.2 Vertical line test

i Exercise

When graphed, vertical lines cannot touch functions at more than one point. Why? Which of the following represent functions?



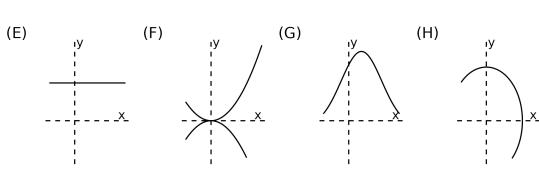


Figure 5.2: Vertical line test: examples.

5.2 Functions in R

Often we need to create our own functions in R. To build them: we use the keyword function alongside the following syntax: function_name <- function(argumentnames) { operation }

- function_name: the name of the function, that will be stored as an object in the R environment. Make the name concise and memorable!
- function(argumentnames): the inputs of the function.
- { operation }: a set of commands that are run in a predefined order every time we call the function.

For example, we can create a function that multiplies a number by 2:

```
mult_by_two <- function(x)\{x * 2\} mult_by_two(x = 5) \text{ # we can also omit the argument name } (x =)
```

[1] 10

If the function body works for vectors, our custom function will do too:

```
mult_by_two(1:10)
[1] 2 4 6 8 10 12 14 16 18 20
```

We can also automate more complicated tasks such as calculating the area of a circle from its radius:

```
calc_circle_area <- function(r){
   pi * r ^ 2
}
calc_circle_area(r = 3)</pre>
```

[1] 28.27433

i Exercise

Create a function that calculates the area of a circle *from its diameter*. So your_function(d = 6) should yield the same result as the example above. Your code:

Functions can take more than one argument/input. In a silly example, let's generalize our first function:

```
mult_by <- function(x, mult){x * mult}
mult_by(x = 1:5, mult = 10)</pre>
```

[1] 10 20 30 40 50

```
mult_by(1:5, mult = 10)
[1] 10 20 30 40 50
  mult_by(1:5, 10)
[1] 10 20 30 40 50
To graph a function, we'll use our friend ggplot2 and stat_function():
  library(tidyverse)
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr
        1.1.2
                   v readr
                             2.1.4
v forcats 1.0.0
                   v stringr 1.5.0
v ggplot2 3.4.2 v tibble 3.2.1
v lubridate 1.9.2 v tidyr
                                1.3.0
           1.0.2
v purrr
-- Conflicts ------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()
                 masks stats::lag()
i Use the conflicted package (<a href="http://conflicted.r-lib.org/">http://conflicted.r-lib.org/</a>) to force all conflicts to become
  ggplot() +
    stat_function(fun = mult_by_two,
                 xlim = c(-5, 5)) # domain over which we will plot the function
```



User-defined functions have endless possibilities! We encourage you to get creative and try to automate new tasks when possible, especially if they are repetitive.



5.3 Common types of functions

5.3.1 Linear functions

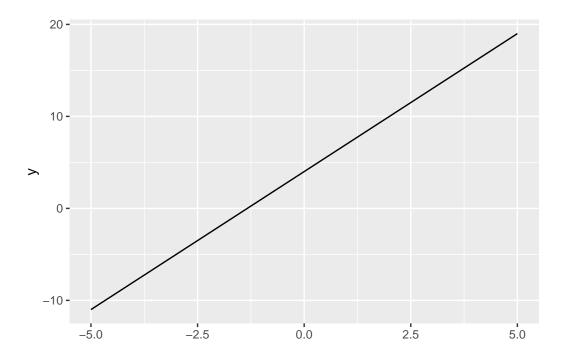
$$y = mx + b$$

Linear functions are those whose graph is a straight line (in two dimensions).

- m is the slope, or the rate of change (common interpretation: for every one unit increase in x, y increases m units).
- b is the y intercept, or the constant term (the value of y when x = 0).

Below is a graph of the function y = 3x + 4:

```
ggplot() + stat_function(fun = function(x){3 * x + 4}, # we don't need to create an object xlim = c(-5, 5))
```



5.3.2 Quadratic functions

$$y = ax^2 + bx + c$$

Quadratic functions take "U" forms. If a is positive, it is a regular "U" shape. If a is negative, it is an "inverted U" shape.

Note that x^2 always returns positive values.

Below is a graph of the function $y = x^2$:



i Exercise

Social scientists commonly use linear or quadratic functions as theoretical simplifications of social phenomena. Can you give any examples?

i Exercise

Graph the function $y=x^2+2x-10$, i.e., a quadratic function with $a=1,\,b=2,$ and c=-10.

Next, try switching up these values and the xlim = argument. How do they each alter the function (and plot)?

5.3.3 Cubic functions

$$y = ax^3 + bx^2 + cx + d$$

These lines (generally) have two curves (inflection points).

Below is a graph of the function $y = x^3$:



i Exercise

We'll briefly introduce Desmos, an online graphing calculator. Use Desmos to graph the following function $y = 1x^3 + 1x^2 + 1x + 1$. What happens when you change the a, b, c, and d parameters?

5.3.4 Polynomial functions

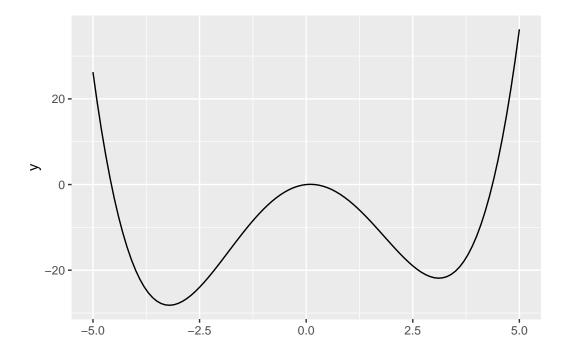
$$y=ax^n+bx^{n-1}+\ldots+c$$

These functions have (a maximum of) n-1 changes in direction (turning points). They also have (a maximum of) n x-intercepts.

High-order polynomials can be made arbitrarily precise!

Below is a graph of the function $y = \frac{1}{4}x^4 - 5x^2 + x$.

```
ggplot() +
stat_function(fun = function(x){1/4 * x ^4 - 5 * x ^2 + x},
xlim = c(-5, 5))
```



5.3.5 Exponential functions

$$y = ab^x$$

Here our input (x), is the exponent.

Below is a graph of the function $y = 2^x$:

```
ggplot() +
stat_function(fun = function(x){2 ^ x},
xlim = c(-5, 5))
```



i Exercise

Exponential growth appears quite frequently social science theories. Which variables can be theorized to have exponential growth over time?

5.4 Logarithms and exponents

5.4.1 Logarithms

Logarithms are the opposite/inverse of exponents. They ask how many times you must raise the base to get x.

So $log_a(b)=x$ is asking "a raised to what power x gives b?" For example, $log_3(81)=4$ because $3^4 = 81.$



⚠ Warning

Logarithms are undefined if the base is ≤ 0 (at least in the real numbers).

5.4.2 Relationships

If,

$$log_a x = b$$

then,

$$a^{log_a x} = a^b$$

and

$$x = a^b$$

5.4.3 Basic rules

$$\frac{\log_x n}{\log_x m} = \log_m n$$

$$\log_x(ab) = \log_x a + \log_x b$$

$$\log_x\left(\frac{a}{b}\right) = \log_x a - \log_x b$$

$$\log_x a^b = b \log_x a$$

$$\log_x 1 = 0$$

$$log_x x = 1$$

$$m^{\log_m(a)} = a$$

5.4.4 Natural logarithms

- We most often use natural logarithms for our purposes.
- This means $log_e(x)$, which is usually written as ln(x).

! Important

 $e \approx 2.7183$.

• ln(x) and its exponent opposite, e^x , have nice properties when we perform calculus.

5.4.5 Illustration of e

Imagine you invest \$1 in a bank and receive 100% interest for one year, and the bank pays you back once a year:

$$(1+1)^1 = 2$$

.

When it pays you twice a year with compound interest:

$$(1+1/2)^2 = 2.25$$

If it pays you three times a year:

$$(1+1/3)^3 = 2.37...$$

What will happen when the bank pays you once a month? Once a day?

$$(1+\frac{1}{n})^n$$

However, there is limit to what you can get.

$$\lim_{n \to \infty} (1 + \frac{1}{n})^n = 2.7183... = e$$

For any interest rate k and number of times the bank pays you t:

$$\lim_{n\to\infty}(1+\frac{k}{n})^{nt}=e^{kt}$$

e is important for defining exponential growth. Since $ln(e^x) = x$, the natural logarithm helps us turn exponential functions into linear ones.

i Exercise

Solve the problems below, simplifying as much as you can.

$$log_{10}(1000)$$

$$log_2(\frac{8}{32})$$

$$10^{log_{10}(300)}$$

 $ln(e^2)$ ln(5e)

5.4.6 Logarithms in R

By default, R's log() function computes natural logarithms:

```
log(100)
```

[1] 4.60517

We can change this behavior with the base = argument:

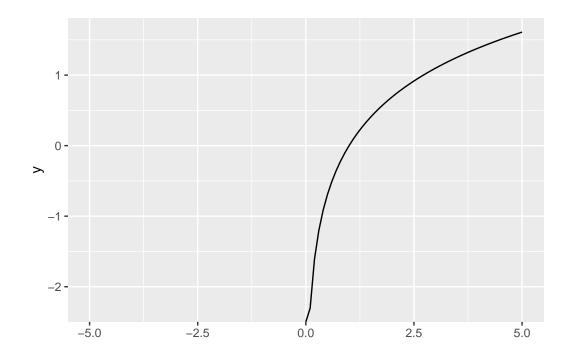
```
log(100, base = 10)
```

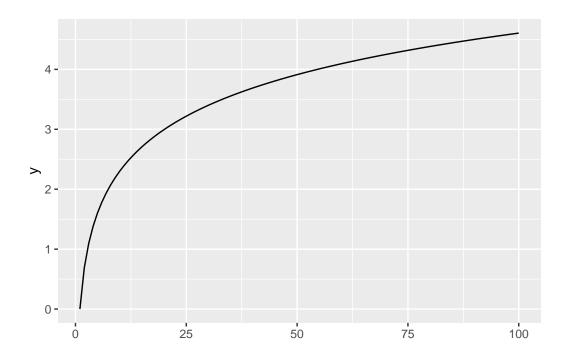
[1] 2

We can also plot logarithms. Remember that $ln(x) \forall x < 0$ is undefined (at least in the real numbers), and ggplot2 displays a nice warning letting us know!

Warning in log(x): NaNs produced

Warning: Removed 50 rows containing missing values (`geom_function()`).





5.5 Composite functions (functions of functions)

Functions can take other functions as inputs, e.g., f(g(x)). This means that the outside function takes the output of the inside function as its input.

Say we have the exterior function $f(x) = x^2$ and the interior function g(x) = x - 3. Then if we want f(g(x)), we would subtract 3 from any input, and then square the result.

• We write this as $(x-3)^2$, not $x^2-3!$

R can handle this just fine:

```
f <- function(x){x ^ 2}
g <- function(x){x - 3}

f(g(5))</pre>
```

[1] 4

Here we can also use pipes to make this code more readable (imagine if we were chaining multiple functions...). Remember that pipes can be inserted with the Cmd/Ctrl + Shift + M shortcut.

```
# compute g(5), THEN f() of that g(5) |> f()
```

[1] 4

i Exercise

Compute g(f(5)) using the definitions above. First do it manually, and then check your answer with R.

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.
- Bank, World. 2023. "World Bank Open Data." https://data.worldbank.org/.
- 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/.
- Robinson, David. 2020. Fuzzyjoin: Join Tables Together on Inexact Matching. https://github.com/dgrtwo/fuzzyjoin.
- 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 Agricultural Research Service." https://fdc.nal.usda.gov/.
- Wickham, Hadley, Danielle Navarro, and Thomas Lin Pedersen. 2023. *Ggplot2: Elegant Graphics for Data Analysis*. 3rd ed. https://ggplot2-book.org/.