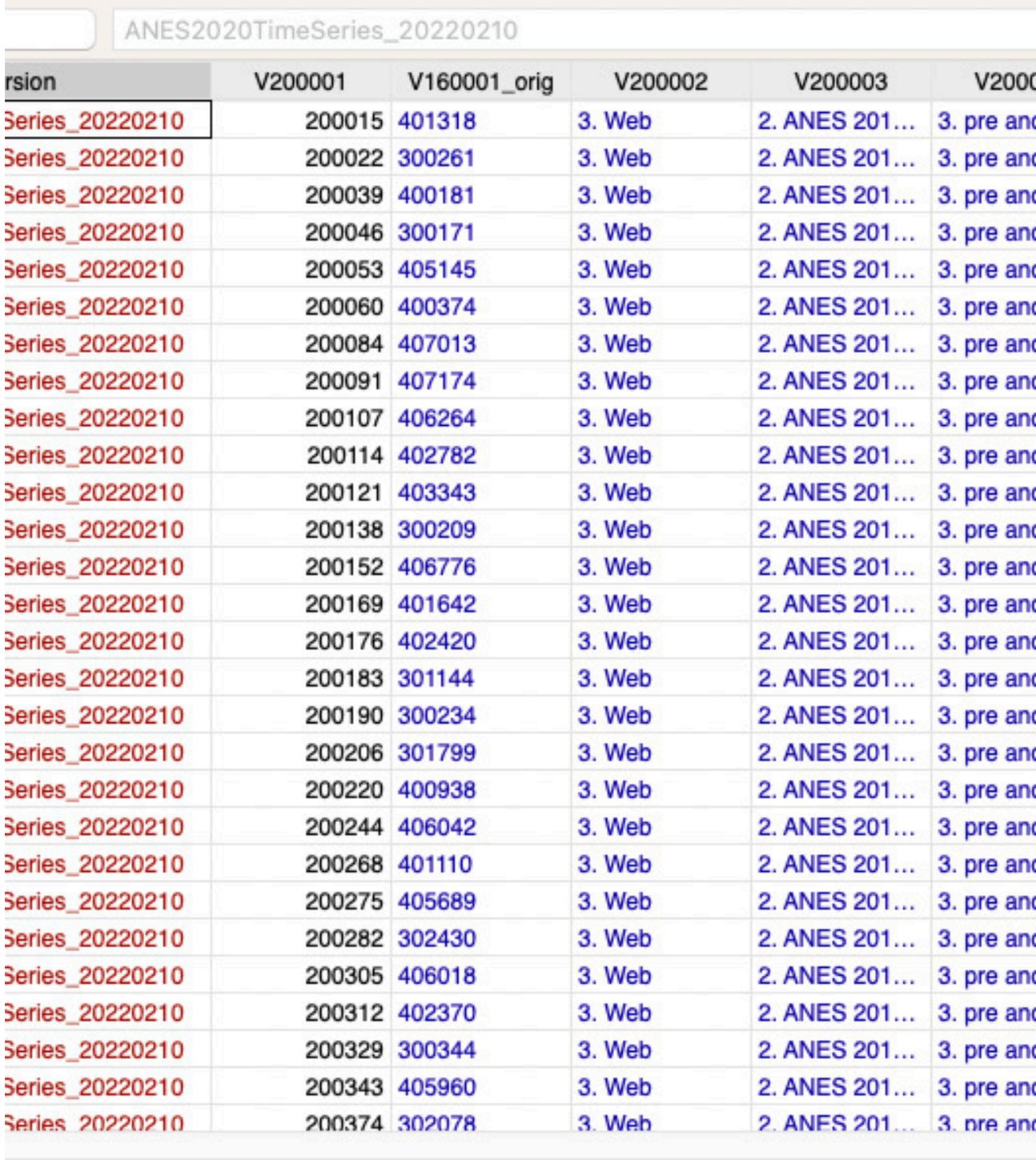
A Field Guide to Data

Storage and Retrieval

Agenda: What do we want to do with data?

- What is Data?
- Tidy Data
- Data Formats
- Curated and Found Data
- Recode Data
- (Begin) Understanding Missing Data


ANES2020TimeSeries_20220210

rsion	V200001	V160001_orig	V200002	V200003	V2000
Series_20220210	200015	401318	3. Web	2. ANES 201…	3. pre and
Series_20220210	200022	300261	3. Web	2. ANES 201…	3. pre and
Series_20220210	200039	400181	3. Web	2. ANES 201…	3. pre and
Series_20220210	200046	300171	3. Web	2. ANES 201…	3. pre and
Series_20220210	200053	405145	3. Web	2. ANES 201…	3. pre and
Series_20220210	200060	400374	3. Web	2. ANES 201…	3. pre and
Series_20220210	200084	407013	3. Web	2. ANES 201…	3. pre and
Series_20220210	200091	407174	3. Web	2. ANES 201…	3. pre and
Series_20220210	200107	406264	3. Web	2. ANES 201…	3. pre and
Series_20220210	200114	402782	3. Web	2. ANES 201…	3. pre and
Series_20220210	200121	403343	3. Web	2. ANES 201…	3. pre and
Series_20220210	200138	300209	3. Web	2. ANES 201…	3. pre and
Series_20220210	200152	406776	3. Web	2. ANES 201…	3. pre and
Series_20220210	200169	401642	3. Web	2. ANES 201…	3. pre and
Series_20220210	200176	402420	3. Web	2. ANES 201…	3. pre and
Series_20220210	200183	301144	3. Web	2. ANES 201…	3. pre and
Series_20220210	200190	300234	3. Web	2. ANES 201…	3. pre and
Series_20220210	200206	301799	3. Web	2. ANES 201…	3. pre and
Series_20220210	200220	400938	3. Web	2. ANES 201…	3. pre and
Series_20220210	200244	406042	3. Web	2. ANES 201…	3. pre and
Series_20220210	200268	401110	3. Web	2. ANES 201…	3. pre and
Series_20220210	200275	405689	3. Web	2. ANES 201…	3. pre and
Series_20220210	200282	302430	3. Web	2. ANES 201…	3. pre and
Series_20220210	200305	406018	3. Web	2. ANES 201…	3. pre and
Series_20220210	200312	402370	3. Web	2. ANES 201…	3. pre and
Series_20220210	200329	300344	3. Web	2. ANES 201…	3. pre and
Series_20220210	200343	405960	3. Web	2. ANES 201…	3. pre and
Series_20220210	200374	302078	3. Web	2. ANES 201…	3. pre and

Data 101

“Happy families are all alike; every unhappy family is unhappy in its own way.”

–Tolstoy

(as related by Hadley Wickham, creator of the tidyverse)

How do we refer to things in data?

- A dataset is a collection of **values**, or numbers that refer to particular **measurements** of particular **variables** for particular **cases**
- Datasets collect these values in structured ways, allowing us to analyze them
- On the right: A dataset containing three **cases**, for two **treatment variables**, and the **value** each person took for each variable
    - **Unit of analysis:** the type of object, person, place, or group that is being measured

	treat_a	treat_b
John	-	2
Jane	16	11
Mary	3	1

How do we refer to things in data?

- Often, data is **coded:**
  - Particular **values** are represented by **numbers.**
  - Sometimes, these number values refer to “real” numbers (ie, 1 dollar)
  - Other times, these numbers are **codes** that refer to concepts (this is a *Likert scale)*:
    - (1) Strongly disagree
    - (2) Disagree
    - (3) Neither agree nor disagree
    - (4) Agree
    - (5) Strongly agree

Categorical data and coding schemes

- Sometimes, even if numbers exist, the thing they represent doesn't even go in order. For instance, religious identity could be coded:
  - (1) Judaism
  - (2) Christianity
  - (3) Islam
  - (4) Buddhism
  - (5) Hinduism
  - (6) Other

Recall Intro Stats:

- Variables have types:
  - *Categorical, Ordinal, Interval, Ratio*
- And scales:
  - *Continuous, Discrete*

What is *tidy* data?

- Data that is *happy*. It fits the following form:
  - Columns are (labeled) **variables**
  - Rows are (usually labeled) **cases**
  - Each type of **observational unit** forms a table
    - The **unit of analysis** is clear
- There are a lot of ways to structure data! You should try to make yours ***tidy.***

Examples

	treat_a	treat_b
John	-	2
Jane	16	11
Mary	3	1

Examples

	John	Jane	Mary
treat_a	-	16	3
treat_b	2	11	1

Examples

Name	treat	result
John	a	-
Jane	a	16
Mary	a	3
John	b	2
Jane	b	11
Mary	b	1

Which should we prefer?

	treat_a	treat_b
John	-	2
Jane	16	11
Mary	3	1

	John	Jane	Mary
treat_a	-	16	3
treat_b	2	11	1

name	treat	result
John	a	-
Jane	a	16
Mary	a	3
John	b	2
Jane	b	11
Mary	b	1

Which should we prefer?

- Columns are **variables**
  - Rows are **cases**
  - Each type of **observational unit** forms a table
    - The **unit of analysis** is clear

	John	Jane	Mary
treat_a	-	16	3
treat_b	2	11	1

name	treat	result
John	a	-
Jane	a	16
Mary	a	3
John	b	2
Jane	b	11
Mary	b	1

- These two are not **wrong**. They are also not **tidy.**

Introducing **tidyr**, **dplyr**, **ggplot2**, and the **tidyverse**

- A new paradigm for wrangling **tidy** data in R
  - **tidyr** & **dplyr**: data wrangling
  - **ggplot2**: graphing
  - **tidyverse**: whole package of support libraries, including tidyr, dplyr and ggplot2
- **`library.install("tidyverse");`** `library(tidyverse)`
- It's become so popular that, for many people, it's synonymous with using R

Why is the **tidyverse** so great?

- **Philosophy**: a more unified, predictable approach to data in R
    - R was originally a language for **statistical analysis**
    - Data wrangling was not the primary focus (and it shows)
- **Community**: developers genuinely committed to making R easier to use for practical data work, especially for beginners
    - Also, lots of integration with the Rstudio people
- **Commitment**: it has **lasted**. There have been a bunch of these “user friendly” kinds of things over the time R has existed (1993), but tidyverse has had staying power
- **Flexibility**: introductory for students; powerful enough for professionals. You don’t “outgrow” it

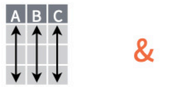
For instance: tidyverse cheatsheets

https://rstudio.github.io/cheatsheets/

Much more helpful than `?tidyr`

Data tidying with tidyr : : CHEATSHEET

Tidy data is a way to organize tabular data in a consistent data structure across packages.
A table is tidy if:

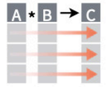


Each **variable** is in its own **column**

&

Each **observation**, or **case**, is in its own row

Access **variables** as **vectors**



Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

Tibbles are a table format provided by the **tibble** package. They inherit the data frame class, but have improved behaviors:

- **Subset** a new tibble with], a vector with [[and $.
- **No partial matching** when subsetting columns.
- **Display** concise views of the data on one screen.

options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf**)** Control default display settings.

View() or **glimpse()** View the entire data set.

CONSTRUCT A TIBBLE

tibble(…**)** Construct by columns.
tibble(x = 1:3, y = c("a", "b", "c"))

tribble(…**)** Construct by rows.
tribble(~x, ~y,
1, "a",
2, "b",
3, "c")

Both make this tibble

```
A tibble: 3 × 2
      x     y
  <int> <chr>
1     1     a
2     2     b
3     3     c
```

as_tibble(x, …**)** Convert a data frame to a tibble.

enframe(x, name = "name", value = "value"**)** Convert a named vector to a tibble. Also **deframe()**.

is_tibble(x**)** Test whether x is a tibble.

Reshape Data - Pivot data to reorganize values into a new layout.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

pivot_longer(data, cols, names_to = "name", values_to = "value", values_drop_na = FALSE**)**

"Lengthen" data by collapsing several columns into two. Column names move to a new names_to column and values to a new values_to column.

pivot_longer(table4a, cols = 2:3, names_to ="year", values_to = "cases")

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

pivot_wider(data, names_from = "name", values_from = "value"**)**

The inverse of pivot_longer(). "Widen" data by expanding two columns into several. One column provides the new column names, the other the values.

pivot_wider(table2, names_from = type, values_from = count)

Split Cells - Use these functions to split or combine cells into individual, isolated values.

table5

country	century	year
A	19	99
A	20	00
B	19	99
B	20	00

→

country	year
A	1999
A	2000
B	1999
B	2000

unite(data, col, …, sep = "_", remove = TRUE, na.rm = FALSE**)** Collapse cells across several columns into a single column.

unite(table5, century, year, col = "year", sep = "")

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174

separate_wider_delim(data, cols, delim, ..., names = NULL, names_sep = NULL, names_repair = "check unique", too_few, too_many, cols_remove = TRUE**)** Separate each cell in a column into several columns. Also **separate_wider_regex()** and **separate_wider_position()**.

separate(table3, rate, sep = "/", into = c("cases", "pop"))

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M

→

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M

separate_longer_delim(data, cols, delim, .., width, keep_eampty**)** Separate each cell in a column into several rows.

separate_longer_delim(table3, rate, sep = "/")

Expand Tables

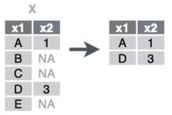
Create new combinations of variables or identify implicit missing values (combinations of variables not present in the data).

expand(data, …**)** Create a new tibble with all possible combinations of the values of the variables listed in … Drop other variables.
expand(mtcars, cyl, gear, carb)

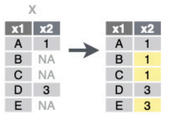
complete(data, …, fill = list()**)** Add missing possible combinations of values of variables listed in … Fill remaining variables with NA.
complete(mtcars, cyl, gear, carb)

Handle Missing Values

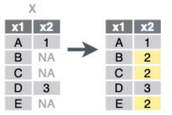
Drop or replace explicit missing values (NA).



drop_na(data, …**)** Drop rows containing NA's in … columns.
drop_na(x, x2)



fill(data, …, .direction = "down"**)** Fill in NA's in … columns using the next or previous value.
fill(x, x2)



replace_na(data, replace**)** Specify a value to replace NA in selected columns.
replace_na(x, list(x2 = 2))

posit®

OK, so, tell me more, tidyverse

- A new data.frame: the tibble
  - Behaves “better” and more predictably, esp with tidyverse tools
    - `data[, 1]` returns another tibble, rather than a vector
    - no partial matching for variable names - must be precise
    - concise data views
    - column names may be “peculiar” (have spaces)
- Still interoperates with most base R functions (mostly, statistical models)

Most things from base R carry over just fine

- We still assign to objects
- Can still pull out variables with $ (although not typical usage)
- Can still construct tibbles by hand
- Note the new information in the tibble display:
  - Type of variable, variable name, dataset size, etc.
  - Tibble also won't print whole screen: just first lines by default

```
> data<-tibble(x = 1:3,
   y = c("a", "b", "c"))
> data
# A tibble: 3 × 2
      x y
  <int> <chr>
1     1 a
2     2 b
3     3 c
```

Importing and Exporting Existing (Mostly Tidy) Data

What is a .csv file?

- .csv stands for comma-separated-value
- Most generic data comes in formats with a common data delimiter, which tells the program (and user) the difference between individual data entries
- CSVs are usually comma-delimited
- Files can also be tab-delimited, space-delimited, %@$*&@-delimited, etc.

Advantages to CSV files

- Almost any statistical program, including R, can read and write them, using any separator
- This makes them incredibly useful to transport files: if in doubt, just ask someone to send you a CSV (or just download a CSV) and R will be a happy camper.
- File size is nice and small
- Standardized, clear, etc

Disadvantages to CSV files

- Everything else.
    - There's a reason many (most) packages don't use them as a default.
        - *(Although there's also a reason they're the de facto standard for interoperability)*
- No data labels at all: makes you dependent on the data codebook
- Subject to occasional errors. What happens, for instance, if you type in a "value" of 3,739?

Fixed-format files

- Rather than having a set delimiter (like a comma) these files have a fixed size for every column
- Disadvantages and advantages much the same
- Both delimited files and fixed-format files are ASCII (American Standard Code for Information Interchange) files - essentially, plain text files with no labels
    - Sometimes, you'll get data in ASCII format that comes with a "data dictionary", which is a way to programmatically write in variable names and values to your ASCII data.

Reading native or platform independent data into R

- Native reading: Rdata (many objects), Rds (one object)
  - `data <- load(“survey.rdata”)`
  - `data <- readRDS(“survey.rds")`
- .csv and other delimiters
  - `data <- read.table("survey.dat", header=TRUE, sep= " “)`
  - `data <- read.csv("survey.csv", header=TRUE)`
- Fixed format
  - `data <- read.fwf("survey.txt", header=TRUE)`

Reading native or platform independent data into R

- The **tidyverse** has their own reading commands (from **readr**) that natively read in tibbles.
  - Typically, this just means replacing the period with an underscore:
- .csv and other delimiters
  - `data <- read_table("survey.dat", header=TRUE, sep= " ")`
  - `data <- read_csv("survey.csv", header=TRUE)`
- Fixed format
  - `data <- read_fwf("survey.txt", header=TRUE)`

Platform/software specific files

- Examples: .dta (Stata) files, .Rdata and .rds (R) files, .por and .sav (SPSS) files, .sas (SAS) files, .xlsx .xls (Excel) files, etc
- Advantages:
  - Support special/specific operations in software packages.
  - Often have nice features - labeling, highlighting, embedded functions, etc
- Disadvantages:
  - **Closed.** Only officially supported by those vendors!

How do I read in different file types?

- In general, there are two ways to go about transporting data
  - When you have a copy of the originating program
  - When you don't have a copy of the originating program
- When you have a copy of the original program (SPSS, SAS, MATLAB, Stata, EViews, who knows what else . . .)
  - You can the originating program to export to a format R can read natively (often .csv, sometimes .Rdata or .rds)

Should I do it this way?

- **Probably not!**
  - Remember: you want to **programmatically start from your raw data and move the entire way through in script files.** (#Reproducibility)
    - I am a pragmatist, though: sometimes things go smoother this way, and it is worth keeping around as a tool in your kit
- This way also requires you to have a copy of the originating program, which you may not (probably don't?) have

The better way: use libraries in R

- The **foreign** package: a useful Swiss army knife for older data
  - `library(foreign)`
    - `dataSPSS <- read.spss("C:\mydata/survey.save", to.data.frame=TRUE)`
    - `dataStata <- read.dta("survey.dta")`
    - `dataSAS <- read.xport("C:/mydata/survey")`
- **foreign** also writes files: `write.foreign()`
- **foreign** also supports some other formats - read the help file!

The better way: use libraries in R

- Foreign doesn't support all types of newer data, though
  - **haven:** similar to foreign, reads a variety of filetypes. Part of the **tidyverse**
    - Usage is easy to visually see: `read_dta`, as opposed to `read.dta`
  - **readstata13:** Stata, in particular, changes its file format all the time, and this package wrangles the complexity for you (not just Stata 13)
    - *You can also save data in Stata in the old format*

readr

- The tidyverse' native file read package doesn't just duplicate existing base R; it also expands to reading Excel and google sheets files:
  - `read_excel()`
    - `read_xls()`, `read_xlsx()`
  - `read_sheets()`
    - Can be pointed to a URL

A Note on Variable kinds

- Remember, there are three primary data types in R you'll likely run across:
  - **Numeric** Variables - numbers; can be stored at varying levels of precision
  - **String** Variables - text
    - Often stored as *factors*, where "Orange" and "Blue", for instance, are two *levels* of the factor, and may have some numeric representation under them
  - **Boolean** Variables - true/false
- Sometimes, variables read in improperly
  - `as.numeric()` - attempts to coerce variables improperly stored into numerics.
  - Many tidyverse functions to enable managing these as well

When Data Isn’t Yet Tidy

In this life, there are two types of data

- Curated Data
  - Data that a human collected and intentionally put together for the purposes of statistical analysis
  - Tends to cleaner (not always **tidy**, but cleaner). Still needs manipulation
- Found Data
  - Data that was produced in some fashion - often programmatically - but not intentionally for analysis. Often for record-keeping
  - Frequently **not tidy at all.** Needs more cleaning and manipulation to make it analyzable

Found data

- For example:
  - Tweets; Facebook posts
  - Traffic camera feeds
  - GPS traces
  - Retail transaction logs
  - Administrative data (billing records, etc)
- From a **research design** standpoint, this found data is both potentially very useful but also very risky
  - Messy, incomplete, biased
  - Requires significant manipulation to be usable
  - Must always keep its limitations in mind

JSON data: an example

- On the right: JSON data
- Below: the same data, tidy

Student	ID	Major	Midterm	Final
Alice	1	PP	88	92
Ben	2	Econ	76	81
Carla	3	Soc	90	95


```
{
 "course": "MAX 200 - Research Design and Analysis”,
 "semester": "Fall 2025",
 "students": [
  {
   "id": 1,
   "name": "Alice Johnson",
   "major": "Public Policy",
   "grades": {
    "midterm": 88,
    "final": 92
   }
  },
  {
   "id": 2,
   "name": "Ben Smith",
   "major": "Economics",
   "grades": {
    "midterm": 76,
    "final": 81
   }
  },
  {
   "id": 3,
   "name": "Carla Rivera",
   "major": "Sociology",
   "grades": {
    "midterm": 90,
    "final": 95
   }
  }
 ]
}
```

Recoding Tidy Data

Working with Curated Tidy Data

- If you already have tidy data, you'll still have things that you want to do with it:
  - Editing coding *within* variables
  - Reshaping/restructuring the rectangular data file *itself* in some way
  - *Adding* more data to the existing set

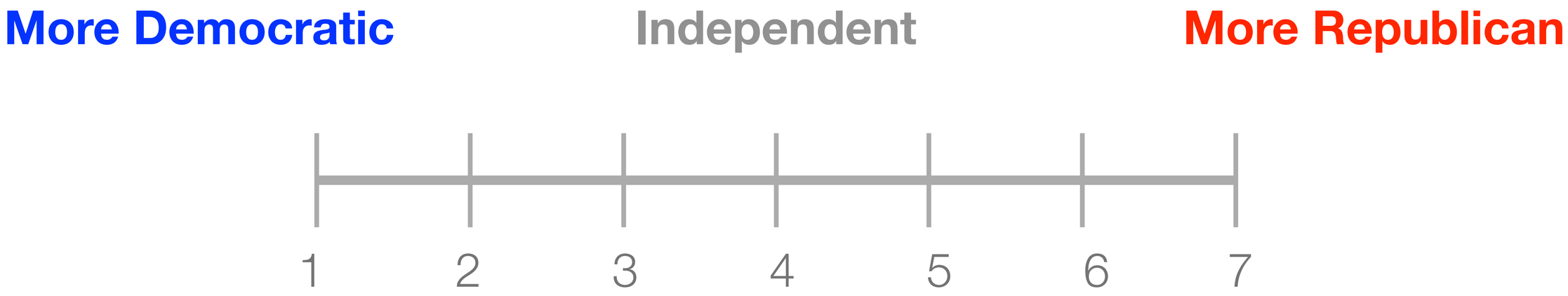
An typical “mostly” tidy data example: Surveys

- Surveys are an incredibly common data type
    - Essentially, you ask people a lot of questions - typically closed form - and respondents answer
- Clear, and usually rectangular and tidy, but still have their challenges
    - Variable coding choices
    - Missing data (nonresponse)
    - Various option choices and constraints, etc

An typical “mostly” tidy data survey recoding example

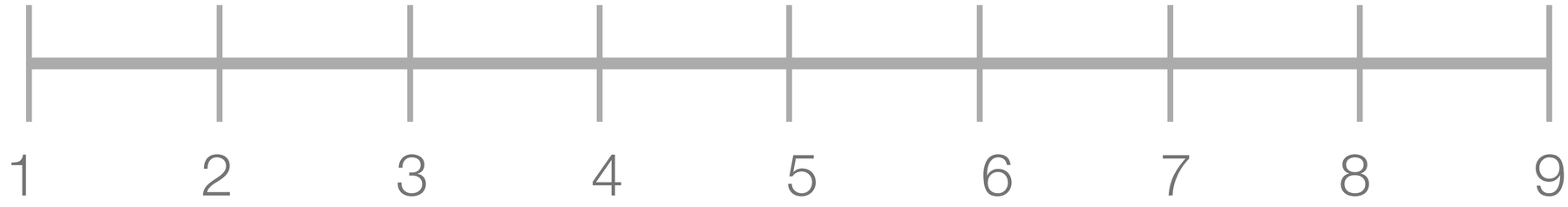
- Partisanship in the United States
- Two stage question:
  - Stage 1: “Would you say that you are a Democrat, Republican, or Independent”?
    - Stage 2A [if Dem/Rep]: “Would you say that you are a Strong or Weak Democrat/Republican?”
    - Stage 2B [if Ind]: “As an independent, would you say you lean towards the Democrats, toward the Republicans, or are truly in the middle?”

What we end up with (theoretically)



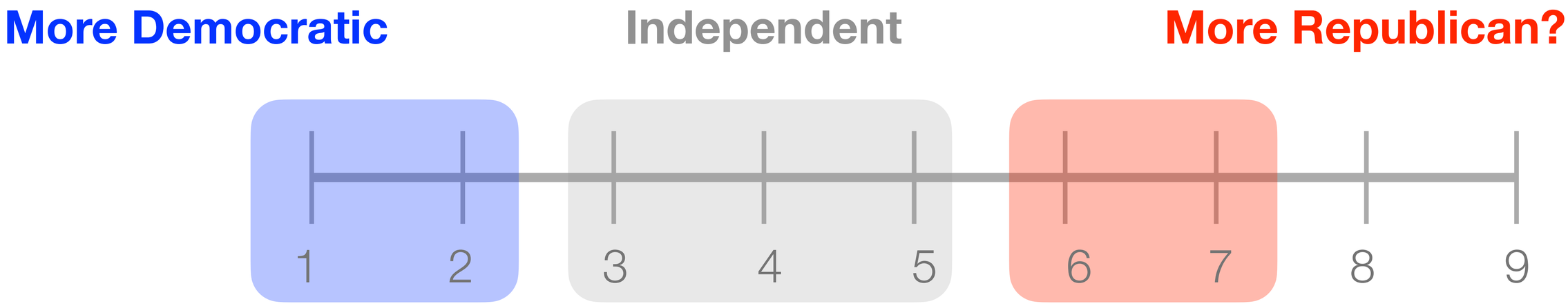
What we actually end up with

More Democratic **Independent** **More Republican?**



8 = I don’t want to answer
9 = I don’t know

Furthermore, remember we actually started out with three groups

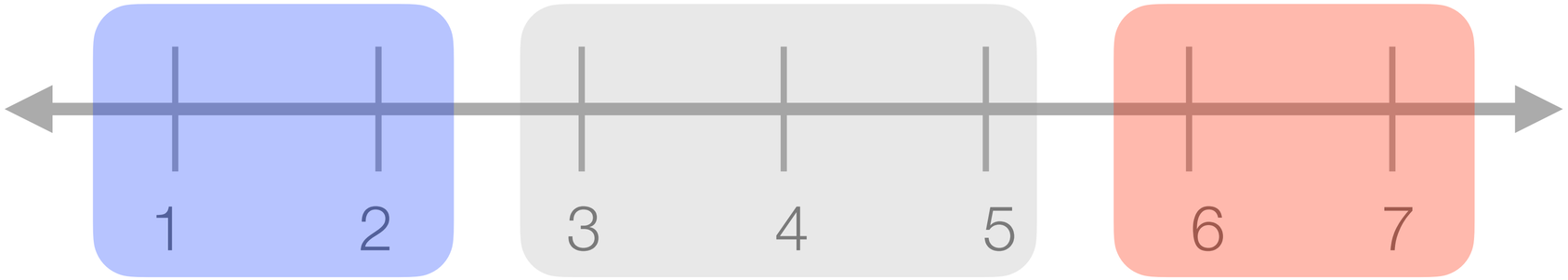


8 = I don’t want to answer
9 = I don’t know

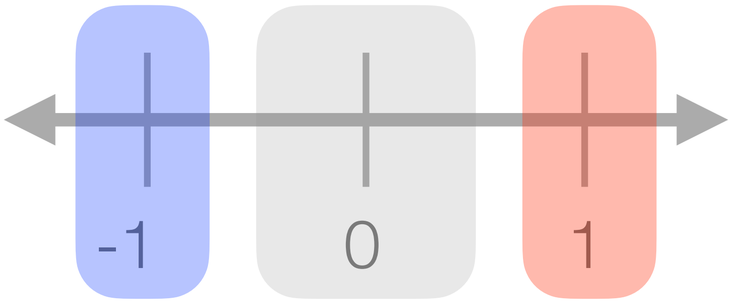
So we actually have some work to do here, even in this “tidy” data

- At minimum: eliminate the 8 and 9
- Often:
  - Maybe we want the original three point scale, rather than the 1-7?
  - Maybe a new three point scale, with only true independents being “independent”?
  - Recenter the scale so that independents are 0?

More Democratic **Independent** **More Republican**



8 = I don't want to answer
99 = I don't know



N/A = I don't want to answer
N/A = I don't know

To do this, let's break our tasks up in chunk

1. Create a new variable that will be the "recode" (*allows us to compare our old variable with our new)*

2. Record -1 into new variable whenever original variable is 1 or 2

3. Record 1 into new variable whenever original variable is 6 or 7

4. Record 0 into new variable whenever original variable is 3, 4 or 5

5. Turn all other variables into "NA"

6. Check our work

Step One: Create a new variable that will be the “recode” (allows us to compare our old variable with our new)

- Let’s imagine we have a dataset “data” with a variable “pid” in it. PID is recorded as 1-7, plus missing values 8 and 9

obs	pid
1	1
2	6
3	3
4	7
5	8
6	9
7	2
8	2
9	3
10	4
11	4
12	7
13	1
14	5

Step One: Create a new variable that will be the “recode” (allows us to compare our old variable with our new)

- First, let’s create a new variable with “NA” in it.

```
data$pid_r <- NA
```

- “NA” is “missing data” for R
- *Why do this?*

obs	pid	pid_r
1	1	*NA*
2	6	*NA*
3	3	*NA*
4	7	*NA*
5	8	*NA*
6	9	*NA*
7	2	*NA*
8	2	*NA*
9	3	*NA*
10	4	*NA*
11	4	*NA*
12	7	*NA*
13	1	*NA*
14	5	*NA*

Step Two: Record -1 into new variable whenever original variable is 1 or 2

- Remember, we can use the brackets in base R to select observations. So this:
  - `[data$pid == 4]`
- When applied to a vector that looks like this:
  - `[0,1,2,3,4,5,7,8,9]`
- Will return:
  - `[F,F,F,F,T,F,F,F,F,F]`

obs	pid	pid_r
1	1	*NA*
2	6	*NA*
3	3	*NA*
4	7	*NA*
5	8	*NA*
6	9	*NA*
7	2	*NA*
8	2	*NA*
9	3	*NA*
10	4	*NA*
11	4	*NA*
12	7	*NA*
13	1	*NA*
14	5	*NA*

Step Two: Record -1 into new variable whenever original variable is 1 or 2

- We can use that Boolean T/F statement to write in new data.
  - `data$pid_recode[data$pid<3] <- -1`
- Break it apart. Always look inside brackets first. What is this asking?
  - `data$pid<3`
- Now move outside, and black box the inside (literally). When that operation is true, what happens?
  - `data$pid_recode[` [redacted] `] <- -1`

obs	pid	pid_r
1	1	*NA*
2	6	*NA*
3	3	*NA*
4	7	*NA*
5	8	*NA*
6	9	*NA*
7	2	*NA*
8	2	*NA*
9	3	*NA*
10	4	*NA*
11	4	*NA*
12	7	*NA*
13	1	*NA*
14	5	*NA*

Step Two: Record -1 into new variable whenever original variable is 1 or 2

- So, when:
  - `data$pid_recode[data$pid<3] <- -1`
- This happens:

- We're getting there!

obs	pid	pid_r
1	1	**-1**
2	6	*NA*
3	3	*NA*
4	7	*NA*
5	8	*NA*
6	9	*NA*
7	2	**-1**
8	2	**-1**
9	3	*NA*
10	4	*NA*
11	4	*NA*
12	7	*NA*
13	1	**-1**
14	5	*NA*

Step Three: Record 1 into new variable whenever original variable is 6 or 7

- We know how to do this one!
  - `data$pid_recode[data$pid>5] <- 1`
- Returns:

- **Whoops.**
  - Anything above 5 includes 8 & 9!

obs	pid	pid_r
1	1	-1
2	6	**1**
3	3	*NA*
4	7	**1**
5	8	**1**
6	9	**1**
7	2	-1
8	2	-1
9	3	*NA*
10	4	*NA*
11	4	*NA*
12	7	**1**
13	1	-1
14	5	*NA*

Step Three: Record 1 into new variable whenever original variable is 6 or 7

- We need to **constrain.** Not any number above 5; only above 5 but lower than 8
  - Need a more complex logic.
- **`data$pid_recode[data$pid < 8 & data$pid > 5] <- 1`**
  - Better.
    - Why?

obs	pid	pid_r
1	1	-1
2	6	**1**
3	3	*NA*
4	7	**1**
5	8	*NA*
6	9	*NA*
7	2	-1
8	2	-1
9	3	*NA*
10	4	*NA*
11	4	*NA*
12	7	**1**
13	1	-1
14	5	*NA*

Step Three: Record 1 into new variable whenever original variable is 6 or 7

- Think of the implicit boolean true/false
  - `data$pid < 8 & data$pid > 5`
- When PID is **under 8** and **over 5**
  - Then do something

obs	pid	bool
1	1	*F*
2	6	***T***
3	3	*F*
4	7	***T***
5	8	*F*
6	9	*F*
7	2	*F*
8	2	*F*
9	3	*F*
10	4	*F*
11	4	*F*
12	7	***T***
13	1	*F*
14	5	*F*

Step Four. Record 0 into new variable whenever original variable is 3, 4 or 5

- We (really) know how to do this one!
- **`data$pid_recode[data$pid < 5 & data$pid > 2] <- 0`**
- Returns:
- Now we're cooking with gas.

obs	pid	pid_r
1	1	-1
2	6	1
3	3	**0**
4	7	1
5	8	*NA*
6	9	*NA*
7	2	-1
8	2	-1
9	3	**0**
10	4	**0**
11	4	**0**
12	7	1
13	1	-1
14	5	**0**

Step Five. Turn all other variables into “NA”

- We already did this!
  - NA is our “base case”

obs	pid	pid_r
1	1	-1
2	6	1
3	3	**0**
4	7	1
5	8	*NA*
6	9	*NA*
7	2	-1
8	2	-1
9	3	**0**
10	4	**0**
11	4	**0**
12	7	1
13	1	-1
14	5	**0**

Step Six. Check our work

obs	pid	pid_r
1	1	-1
2	6	1
3	3	0
4	7	1
5	8	*NA*
6	9	*NA*
7	2	-1
8	2	-1
9	3	0
10	4	0
11	4	0
12	7	1
13	1	-1
14	5	0

- Here, we could just look at the data. (Looks good!)
- Typically, we won't have this ability, though, in a larger dataset. So, instead, we create a table, where we can double check that everything went where it was supposed to

```
>table(data$pid, data$pid_recode)

   -1 0 1
1   2 0 0
2   2 0 0
3   0 2 0
4   0 2 0
5   0 1 0
6   0 0 1
7   0 0 2
9   0 0 0
```

Step Six. Check our work

- Note that we still have our original variable!
  - So we can both check it against the new variable
  - And use it to create other new variables, if we wish

obs	pid	pid_r
1	1	-1
2	6	1
3	3	0
4	7	1
5	8	*NA*
6	9	*NA*
7	2	-1
8	2	-1
9	3	0
10	4	0
11	4	0
12	7	1
13	1	-1
14	5	0

Base R and the tidyverse

- Base R and the tidyverse think about data differently.
- Base R:
  - Conditionality, if/then, boolean true/false
  - Requires you to think in *nested programming structures*
- Tidyverse:
  - What if we want to think in normal sentences?

Introducing the pipe

- Originally introduced in the tidyverse as:

  `%>%`

- Later, introduced to base R (and adopted by the tidyverse) as:

  `|>`

- The pipe is a fundamental concept of working with code in R.
  - It means "**take this, then do this**", or simply, **"and then"**
  - It allows us to read code (more) like a sentence, rather than within nested commands

So let's take our prior example

- Recoding party ID
  - In base R: match statements with true/false, then do **that** to **this**
  - In tidyverse: take **this**, then to **that** to it
- Logically, as a sentence:
  - Take this dataset, and then, take this variable from the dataset. Then make a new variable that looks like this relative to the old variable

Tidy commands

```
data |>
  mutate(pid_recode2 = case_when(
    pid %in% 1:2 ~ -1,
    pid %in% 3:5 ~ 0,
    pid %in% 6:7 ~ 1,
    pid == 9     ~ NA
  ))
```

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

start with the object "data"

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

then do the following to it

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

"mutate" = change the data values

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

make a new variable in data: pid_recode2

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

that looks like the following

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

When pid7 is in the range 1 to 2, make the new variable -1

Let’s map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

When pid7 is in the range 3 to 5, make the new variable 0

Let's map it out

```
data |>
   mutate(pid_recode2 = case_when(
      pid %in% 1:2 ~ -1,
      pid %in% 3:5 ~ 0,
      pid %in% 6:7 ~ 1,
      pid == 9     ~ NA
   ))
```

When pid7 is in the range 6 to 7, make the new variable 1

Let's map it out

```
data |>
    mutate(pid_recode2 = case_when(
        pid %in% 1:2 ~ -1,
        pid %in% 3:5 ~ 0,
        pid %in% 6:7 ~ 1,
        pid == 9     ~ NA
    ))
```

When pid7 is 9, make the new variable NA

Let's map it out

```
data |>
    mutate(pid_recode2 = case_when(
        pid %in% 1:2 ~ -1,
        pid %in% 3:5 ~ 0,
        pid %in% 6:7 ~ 1,
        pid == 8 | pid == 9 ~ NA
    ))
```

When pid7 is 8 or 9, make the new variable NA

If you run it first, without saving over the data, it shows a nice little way to check

---

Note, however, that if you actually want to save your new data object, you have to either overwrite the existing one or save a new one

```
> data |>
+   mutate(pid_recode2 = case_when(
+     pid %in% 1:2 ~ -1,
+     pid %in% 3:5 ~ 0,
+     pid %in% 6:7 ~ 1,
+     pid == 8 | pid == 9      ~ NA
+   ))
# A tibble: 14 × 3
     pid pid_recode pid_recode2
   <dbl>      <dbl>       <dbl>
 1     1         -1          -1
 2     6          1           1
 3     3          0           0
 4     7          1           1
 5     8         NA          NA
 6     9         NA          NA
 7     2         -1          -1
 8     2         -1          -1
 9     3          0           0
10     4          0           0
11     4          0           0
12     7          1           1
13     1         -1          -1
14     5          0           0
```

Saving the new data over the old

```
data <- data |>
  mutate(pid_recode2 = case_when(
    pid %in% 1:2 ~ -1,
    pid %in% 3:5 ~ 0,
    pid %in% 6:7 ~ 1,
    pid == 8 | pid == 9 ~ NA
  ))
```

(You still have the original variables, you're just writing the whole tibble over again)

Checking with tables

- `table(data$pid, data$pid_recode2)`

- `data |> count(pid, pid_recode2)`

```
# A tibble: 9 × 3
    pid pid_recode2     n
  <dbl>       <dbl> <int>
1     1          -1     2
2     2          -1     2
3     3           0     2
4     4           0     2
5     5           0     1
6     6           1     1
7     7           1     2
8     8          NA     1
9     9          NA     1
```

Let's Practice!

What does this code do?

```
data <- data |>
  mutate(relig_recode2 = case_when(
    relig %in% 1:3 ~ 1,
    relig %in% 3:5 ~ 0,
    pid == 8 | pid == 9 ~ NA
  ))
```

Let's Practice!

What does this code do?

```
data <- data |>
  mutate(relig_recode2 = case_when(
    relig %in% 1:3 ~ 1,
    relig %in% 4:6 ~ 0,
    relig == 8 | relig == 9 ~ NA
  ))
```

Religions 1-3 become 1; 4-6 become 0; 8/9 NA

(Abrahamic vs other religions)

Let's Practice!

What does this code do?

```
data <- data |>
  mutate(income_thirds = case_when(
    faminc %in% 1:50000 ~ 1,
    faminc %in% 50001:100000 ~ 2,
    faminc %in% 100001:10000000 ~ 3,
 ))
```

Let's Practice!

What does this code do?

```
data <- data |>
  mutate(income_thirds = case_when(
    faminc %in% 1:50000 ~ 1,
    faminc %in% 50001:100000 ~ 2,
    faminc %in% 100001:10000000 ~ 3,
 ))
```

Divide income into (rough) thirds

Base R vs Tidyverse

- When do I use which?
- A lot of it is preferences
  - Base R can be useful if you're just trying to do one or two things quickly
  - The tidyverse much more useful for large scale data recoding projects
- We're just getting into the tidyverse, though . . . much more to come

Wrapping up

- This time:
  - Data, data formats, beginning data manipulation
- Coming up:
  - Much more data manipulation!
  - Continuing statistical graphics and visualization