Structural Data Manipulation

Reshaping and Refolding

Housekeeping

- Assignments page has been updated
    - This weeks assignment is recommended but optional (no grade)
- Week numbering has been changed. The challenge is figuring out what week to refer to:
    - The week of the lecture, *or* the week the assignment is due?
- Previously, it had been the week the assignment is due, meaning we would be covering material from “4” during Week 3
    - Now, Week 3 refers to Week 3
        - The assignment that starts on week 3 is still due week 4

Agenda: What do we want to do with data?

- Last time: Tidy Data, Data Coding, Data Formats, Recoding Data
- This time:
  - Filtering data
  - Merging Data
  - Collapsing, grouping, and summarizing data
  - Working with survey data: sampling & weighting
  - Reshaping data

Review

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

–Tolstoy

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

Tidy data

- 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.***

	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

tidyverse cheatsheets

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

Much more helpful than `?tidyr`

Data tidying with tidyr : : CHEATSHEET

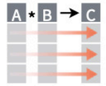
Tidy data is a way to organize tabular data in a consistent data structure across packages.
A table is tidy if:

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

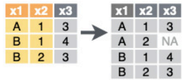
separate_longer_delim(data, cols, delim, .., width, keep_eampty**)** Separate each cell in a column into several rows.

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).

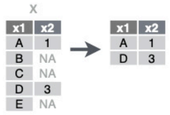
expand(data, …**)** Create a new tibble with all possible combinations of the values of the variables listed in … Drop other variables.
expand(mtcars, cyl, gear, carb)



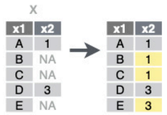
complete(data, …, fill = list()**)** Add missing possible combinations of values of variables listed in … Fill remaining variables with NA.
complete(mtcars, cyl, gear, carb)

Handle Missing Values

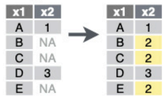
Drop or replace explicit missing values (NA).



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))

Reading data with `readr`

- `read_csv` (base: `read.csv`)
- Alternate data:
  - `library(foreign), library(haven)`

Piping: a new command structure

- 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

Recoding data with `dplyr`

```
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)

Structural Data Manipulation

- Filtering and Selecting Data
- Combining Datasets
- Collapsing Data
- Sampling Weights
- Reshaping Data

“Seeing” Structural Data

- The trick here is to be able to visualize your data
  - What are the rows?
  - What are the columns?
  - What is in them?
  - What should be in them?
  - How do they need to be re-arranged?

Filtering and Selecting Data

What if we have too much data?

- Large surveys often come to use with a profusion of data
- Which we don't always need!
- You should be careful about throwing away data forever
  - But this is why we **never destructively edit data**
  - And do **reproducible analysis**

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

The `filter()` command

- We use logical statements to keep and drop **observations** in tidyverse
  - If we want to keep every observation where year is 2020:
    - `df |> filter(year == 2020)`
  - If we want to drop every observation where year is 2020:
    - `df |> filter(year != 2020)`
  - We can also combine filters:
    - `df |> filter(year != 2020, turnout == 1)`

```
df |>  filter(year == 2020)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

```
df |>  filter(year == 2020)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

id	year	turnout	party
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

```
df |>  filter(year != 2020)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

```
df |>  filter(year != 2020)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A

```
df |>  filter(year != 2020, turnout == 1)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

```
df |>  filter(year != 2020, turnout == 1)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

id	year	turnout	party
1	2012	1	D
3	2012	1	R
5	2016	1	R
6	2016	1	I

The `select()` command

- We use logical statements to keep and drop **variables** in tidyverse
  - To keep just year and turnout:
  - `df |>  select(id, year, turnout)`
  - To drop party:
  - `df |>  select(-turnout)`

```
df |>  select(id, year, turnout)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

```
df |>  select(id, year, turnout)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

id	year	turnout
1	2012	1
2	2012	0
3	2012	1
4	2012	0
5	2016	1
6	2016	1
7	2016	0
8	2016	0
9	2020	0
10	2020	1
11	2020	1
12	2020	1

```
df |>  select(-turnout)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

```
df |>  select(-turnout)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I
11	2020	1	D
12	2020	1	R

id	year	party
1	2012	D
2	2012	D
3	2012	R
4	2012	R
5	2016	R
6	2016	I
7	2016	D
8	2016	N/A
9	2020	I
10	2020	I
11	2020	D
12	2020	R

Pipe commands can be combined and sequenced

```
df |>
    select(id, year, turnout) |>
    filter(year == 2020)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I

Pipe commands can be combined and sequenced

```
df |>
    select(id, year, turnout) |>
    filter(year == 2020)
```

id	year	turnout	party
1	2012	1	D
2	2012	0	D
3	2012	1	R
4	2012	0	R
5	2016	1	R
6	2016	1	I
7	2016	0	D
8	2016	0	N/A
9	2020	0	I
10	2020	1	I

id	year	turnout
9	2020	0
10	2020	1
11	2020	1
12	2020	1

Combining Datasets

Combining datasets: two approaches

- First: adding more *observations* with identical *variables*
  - Variables must be *exactly* identical! Labeling matters
  - Must be careful if a variable is missing
- This is known as **appending** or **stacking**

id	var1	var2	var3
1			
2			
3			
4			

id	var1	var2	var3
5			
6			
7			
8			

id	var1	var2	var3
1			
2			
3			
4			
5			
6			
7			
8			

Combining datasets: two approaches

- Second: adding more *variables* for the same *observations*
  - Observations must be *exactly* identical!
  - Same individuals, same states, etc
- This is known as **merging**

id	var1	var2	var3
1			
2			
3			
4			

id	var4	var5	var6
1			
2			
3			
4			

id	var1	var2	var3	var4	var5	var6
1						
2						
3						
4						

Appending: More observations for the same variables

- The `bind_rows()` command from `dplyr` takes care of this
- You can also use `rbind()` from base R, but it's less forgiving
  - `dplyr` checks for matching variables, will insert missing

```
library(dplyr)

# Stack two data frames (same variables)
combined <- bind_rows(df1, df2)

# You can stack many at once
combined <- bind_rows(df1, df2, df3)

# Keep track of data frames
dfs <- list(df1, df2, df3)
combined <- bind_rows(dfs, .id = "source")
```

Merging: More variables for the same observations

- This tends to be a more difficult case, as it is typically trickier to match observations than variables
  - If you're 100% positive that the data is sorted exactly right, with exactly the same cases, you could just slap it on, like you do when you're appending.
  - But this is not typically the case, and it's dangerous to assume that it is the case
- There are also a variety of different *types* of merges you may want to do

Merging: More variables for the same observations

- You can merge in many ways
  - 1:1 by observation *(the slap it together approach)*
  - 1:1 by case id
  - many:1 (*many people in the same state get the same value*)
  - 1:many (*vice versa*)
  - many:many

Merging in R

- In base R, we can merge with the `merge()` command:

```
# Merge two data frames by key

combined <- merge(data1, data2, by="id")
```

  - **id** is a variable common to both sets that serves as a *unique identifier*

  - **Unique identifier**: maps precisely, one to one, from an observation in one dataset to a single other observation in another dataset.

A new language: joining in `dplyr`

- A join is a term from the world of databases (like SQL)
  - Refers to “joining” two databases together

```
left_join(x, y, by = "id")   # keep all rows of x, add matches from y
right_join(x, y, by = "id")  # keep all rows of y, add matches from x
inner_join(x, y, by = "id")  # only matching rows (intersection)
full_join(x, y, by = "id")   # keep all rows from both (union)
```

  - **Keep track** of what you’re dropping and keeping!

A minimal example

```
#Setup:
library(dplyr)
data1 <- data.frame(id = 1:3, age = c(25, 30, 35))
data2 <- data.frame(id = 2:4, income = c(40, 50, 60))
```

id	age
1	25
2	30
3	35

id	income
2	40
3	50
4	60

A minimal example

```
#Setup:
library(dplyr)
data1 <- data.frame(id = 1:3, age = c(25, 30, 35))
data2 <- data.frame(id = 2:4, income = c(40, 50, 60))
```

```
> left_join(data1, data2, by = "id")

  id age income
1  1  25     NA
2  2  30     40
3  3  35     50

> # Keeps id 1, 2, 3 (from df1), attaches income where
available
```

id	age	income
1	25	NA
2	30	40
3	35	50

A minimal example

```
#Setup:
library(dplyr)
data1 <- data.frame(id = 1:3, age = c(25, 30, 35))
data2 <- data.frame(id = 2:4, income = c(40, 50, 60))
```

```
> right_join(data1, data2, by = "id")

  id age income
1  2  30     40
2  3  35     50
3  4  NA     60

> # Keeps id 2, 3, 4 (from data2), attaches age where
available
```

id	age	income
2	30	40
3	35	50
4	NA	60

A minimal example

```r
#Setup:
library(dplyr)
data1 <- data.frame(id = 1:3, age = c(25, 30, 35))
data2 <- data.frame(id = 2:4, income = c(40, 50, 60))
```

```r
> inner_join(data1, data2, by = "id")

  id age income
1  2  30     40
2  3  35     50

> # Only id 2, 3 (present in both)
```

id	age	income
2	30	40
3	35	50

A minimal example

```
#Setup:
library(dplyr)
data1 <- data.frame(id = 1:3, age = c(25, 30, 35))
data2 <- data.frame(id = 2:4, income = c(40, 50, 60))
```

```
> full_join(data1, data2, by = "id")
```

```
  id age income
1  1  25     NA
2  2  30     40
3  3  35     50
4  4  NA     60
```

id	age	income
1	25	NA
2	30	40
3	35	50
4	NA	60

```
> # Keeps id 1, 2, 3, 4 (fills NAs where no match)
```

many:1 joins and merges

- Let's say we have many **survey respondents** (people) in different **states**
  - And we have the election results from those states and want to add them for each person

id	cnty	var2	var3
1	A		
2	A		
3	B		
4	B		
5	B		

cnty	vote
A	45
B	55

id	cnty	var2	var3	vote
1	A			45
2	A			45
3	B			55
4	B			55
5	B			55

- This is a **many** to **one join**: **many** people joined to **one** county
- Also: students embedded in classrooms, all other kinds of hierarchy, etc

many:1 joins and merges

- Set up two data sets for the example

```
> people
# A tibble: 5 × 3
  person_id county_fips   age
      <int> <chr>        <dbl>
1         1 01001           25
2         2 01001           40
3         3 01003           31
4         4 01003           50
5         5 01003           62
```

```
> elections
# A tibble: 2 × 2
  county_fips two_party_vote
  <chr>                <dbl>
1 01001                   45
2 01003                   55
```

many:1 joins and merges

- This is a **left join**: we want to keep everything from the first (the “many”) and match it to whatever is available in the second

```
> left_join(people, elections, by = "county_fips")
# A tibble: 5 × 4
  person_id county_fips   age two_party_vote
      <int> <chr>       <dbl>          <dbl>
1         1 01001          25             45
2         2 01001          40             45
3         3 01003          31             55
4         4 01003          50             55
5         5 01003          62             55
```

Merges are *complicated* things!

- These examples make it look simple. But:
  - The merge key must be exact.
    - *(`0300` ≠ `300`, if stored as a string)*
    - ***`LOS_ANGELES` ≠ `Los Angeles`***
  - The merge key variable should be the same in both sets (county ≠ county_code)
    - Although you can match them explicitly: `by = c("county" = "county_fips")`
  - Missing data can propagate and multiply quickly
  - Data can be lost quickly if using the wrong join

Fixing key problems

- Standardize types before joins
  - You can use commands to covert between variable types (`as.numeric`; `as.character`)
  - You can also manipulate string variables with `library(stringr)`
  - `trimws() - trim white space in character strings`
  - `tolower()`, `toupper()` - lower or uppercase strings

What can I do?

- **Validate.** Before and after every join, check:
- 1. Number of rows (did they multiply unexpectedly?).
- 2. Number of unmatched cases (via `anti_join`).

```
left_join(people, elections, by = “county_fips") # our join from earlier
anti_join(people, elections, by = "county_fips") # rows in people with no match
anti_join(elections, people, by = "county_fips") # rows in elections with no match
```

- 3. Distribution of key variables (did you lose or duplicate counties?).
- 4. Variable name suffixes (are you keeping the right columns?).

Collapsing & Compressing Datasets

Collapse? Compress? Grouping? Summarizing?

- Sometimes, you want information *not* about the units of your data, but of a *higher-level* unit that your core units are grouped into
  - We often refer to this as your "*i*" *and "j"*
    - Respondents (i) nested within counties (j)
    - Students (i) nested within classrooms (j)
    - Employees (i) nested within nonprofits (j)
    - Bureaucrats (i) nested within departments (j)

Collapse? Compress? Grouping? Summarizing?

- Let's say we wanted to know how a class overall scored on a state assessment
- We need to take the mean of score **within** each class
  - And store it as a new dataset

id	class	score
1	A	80
2	A	70
3	B	90
4	B	80
5	B	70
6	C	95
7	C	85

class	score
A	75
B	80
C	90

We need a piping command

- To take the overall mean in dplyr, we:
  - Take the original data **and then** summarize it by mean:

```
scores |>
  summarise(mean_score = mean(score))
```

- To take a grouped mean: we take the original data **and then** group it **and then** summarize it:

```
scores |>
  group_by(class) |>
  summarise(mean_score = mean(score))
```

We need a piping command

- We can also take different, and more, summary statistics:
  - Mean, standard deviation, count
- And, of course, take numbers for different variables
  - Here, we take info for score as well as age

```
> scores |>
+   group_by(class) |>
+   summarise(
+     mean_score = mean(score),
+     sd_score   = sd(score),
+     N          = n(),
+     mean_age   = mean(age),
+   )
# A tibble: 3 × 5
  class mean_score sd_score     N mean_age
  <chr>      <dbl>    <dbl> <int>    <dbl>
1 A             75     7.07     2     18.5
2 B             80    10        3     21
3 C             90     7.07     2     21
>
```

We need a piping command

- You can also group by multiple levels:
  - students *inside* classes *inside* schools

```
> scores |>
+   group_by(class, school) |>
+   summarise(
+     mean_score = mean(score),
+     sd_score   = sd(score),
+     N          = n(),
+     mean_age   = mean(age),
+   )
```

Pitfalls

- Remember, if you want to save results, you always have to write to an object (`<-`)

- Missing data

  - Data is frequently missing - maybe a student missed test day

  - R often doesn't know what to do with missing data, and can do unpredictable things

  - If you have missing data, must declare what to do per line (`na.rm = T`)

Pitfalls

- Fundamentally, we have **changed the unit of analysis**
  - Went from a student unit to a class unit
- So statistics about the dataset will change!
  - mean(scores$score) = 81.42
  - mean(summary_data$mean_scores) = 81.66

Pitfalls

- You may wish to explicitly control what happens to grouping data
  - **Grouped tibbles** behave differently than **ungrouped tibbles**
    - For instance, mean summarizing a *grouped tibble* will summarize it by groups, but mean summarizing an *ungrouped tibble* will summarize the whole set
  - But **only** when using commands that recognize them as such
  - You can control this behavior with `.groups="keep"` or `.groups="drop"`

For instance:

```
> sum_drop <- scores |>
+   group_by(class) |>
+   summarise(mean_score =
+     mean(score),
+     .groups="drop")

> sum_drop |>
+   summarise(mean =
+     mean(mean_score))
# A tibble: 1 × 1
   mean
  <dbl>
1  81.7
```

```
> sum_keep <- scores |>
+   group_by(class) |>
+   summarise(mean_score =
+     mean(score),
+     .groups="keep")

> sum_keep |>
+   summarise(mean =
+     mean(mean_score))
# A tibble: 3 × 2
  class  mean
  <chr> <dbl>
1 A        75
2 B        80
3 C        90
```

```
> mean(sum_drop$mean_score)
[1] 81.66667
```

```
> mean(sum_keep$mean_score)
[1] 81.66667
```

All together

```
scores |>
 group_by(class) |>
 summarise(
   mean_score = mean(score, na.rm=T),
   sd_score   = sd(score, na.rm=T),
   N          = n(),
   mean_age   = mean(age, na.rm=T),
   .groups="drop"
 )
```

Sampling Weights

A woefully short crash course on survey sampling

- Ideally, we have a random, **representative** survey
    - Each member of the population had an equal chance of being drawn into the sample
    - Randomness ensures equality between population and sample
- Representative samples ensure we can use critical tools from statistics (like the central limit theorem) to allow for **inference**
    - For instance: **“I am 95% confident the true population mean falls within three percentage points of the sample mean”**

A woefully short crash course on survey sampling

- In practice, surveys are frequently **not representative** of their populations
  - Sometimes, this is **unintentional**: laws of probability tell us that, in general on average, random sampling generates an unbiased sample.
    - In any one particular sample, though, you could just happen to draw a biased sample (sometimes you flip 20 heads in a row, too)
  - Other times, this is **intentional**
    - We may want to oversample small populations in an overall sample to enable effective inference on that subgroup

Correcting under and oversamples

- If you have nonrepresentative sample, you may be able to return it to some kind of representativeness through **weighting** your sample observations
    - Essentially: we tell our software that some samples had a higher probability of being drawn than others
- For instance, if we oversampled Group A, by doubling their chances of selection, we could *downweight* them in the survey relative to Group B
- Alternately, if we have a sample that is randomly drawn but we know over-represents certain individuals over others, we can weight known statistics in our sample to match known parameters in the population

Weights & raking procedures

- These **survey weights** are typically included in large public surveys
    - They can also be created through a **raking procedure** in data you collect yourself
- When they exist, weights should be used to gain better representative data

Example data

- Let's say we have a survey of 12 people, asking whether they turned out to vote, their education, and age
  - Educ: mode of BA+
  - Age: mean of 35.08
  - Turnout: "mean" of .66
- Does this seem "right"?

id	educ	age	turnout
1	HS	55	0
2	HS	48	1
3	SomeCol	40	0
4	BAplus	30	1
5	BAplus	28	1
6	BAplus	27	1
7	SomeCol	38	0
8	BAplus	29	1
9	BAplus	33	1
10	BAplus	31	0
11	SomeCol	36	1
12	BAplus	26	1

Summary Statistics

- It shouldn't! *but the survey provides a weight.*
    - Visual inspection confirms intuition is correct
- But we still need to analyze it

id	educ	age	turnout	pwt
1	HS	55	0	3.0
2	HS	48	1	3.0
3	SomeCol	40	0	1.5
4	BAplus	30	1	0.5
5	BAplus	28	1	0.5
6	BAplus	27	1	0.5
7	SomeCol	38	0	1.5
8	BAplus	29	1	0.5
9	BAplus	33	1	0.5
10	BAplus	31	0	0.5
11	SomeCol	36	1	1.5
12	BAplus	26	1	0.5

Weights in R & `srvyr`

- `srvyr` provides a tidy-centric way to weight your data
- Let's lock it in
- We start by *declaring* the survey design to R
  - `des <- as_survey_design(dat, weights = pwt)`

```
> dat
# A tibble: 12 × 5
      id   age turnout educ           pwt
   <int> <dbl>   <dbl> <fct>        <dbl>
 1     1    22       1 HS             2.2
 2     2    27       0 HS             2.2
 3     3    31       1 SomeCollege    1.2
 4     4    35       1 BAplus         0.6
 5     5    41       1 BAplus         0.6
 6     6    45       0 BAplus         0.6
 7     7    50       1 SomeCollege    1.2
 8     8    54       0 BAplus         0.6
 9     9    60       1 BAplus         0.6
10    10    33       0 BAplus         0.6
11    11    38       1 SomeCollege    1.2
12    12    29       1 BAplus         0.6
```

Weights in R & `srvyr`

- We start by *declaring* the survey design to R and saving it as a new object

```
des <- as_survey_design(dat, weights = pwt)
```

- Now we can use summary commands that will weight our responses properly

```
des |>
  summarise(
    mean_age      = survey_mean(age),
    turnout_rate = survey_mean(turnout)
  )
```

Weights in R & `srvyr`

- We start by *declaring* the survey design to R and saving it as a new object

```
des <- as_survey_design(dat, weights = pwt)
```

- Now we can use summary commands that will weight our responses properly

```
des |>
  summarise(
    mean_age      = survey_mean(age),
    turnout_rate = survey_mean(turnout)
  )
```

“Corrected” weights

- Once we “correct” for the oversample of highly educated individuals
  - We can see the best prediction is an age of 41.6 and a turnout of 53.6
  - Rather than 35.1 and 0.667

```
> des |>
+   summarise(
+     mean_age      = survey_mean(age),
+     turnout_rate = survey_mean(turnout)
+   )
# A tibble: 1 × 4
  mean_age mean_age_se turnout_rate turnout_rate_se
     <dbl>       <dbl>         <dbl>           <dbl>
1     41.6        3.64         0.536           0.193

> des |>
+   summarise(
+     mean_age      = mean(age),
+     turnout_rate = mean(turnout)
+   )
# A tibble: 1 × 2
  mean_age turnout_rate
     <dbl>         <dbl>
1     35.1         0.667
```

The power of piping and the tidyverse

- Let's combine the last two exercises
  - Let's say we have a survey, collected across three years
  - And each respondent's age
  - But we also know our survey is not perfectly representative

id	year	educ	age	pwt
1	2012	BAplus	29	0.6
2	2012	BAplus	31	0.6
3	2012	BAplus	33	0.6
4	2012	HS	58	3.0
5	2016	BAplus	27	0.6
6	2016	BAplus	30	0.6
7	2016	BAplus	32	0.6
8	2016	SomeColl	41	1.5
9	2020	BAplus	28	0.6
10	2020	BAplus	34	0.6
11	2020	SomeColl	39	1.5
12	2020	HS	56	3.0

A weighted summary dataset by year

- First, we declare it a survey

```
> des <- as_survey_design(dat, weights = pwt)
```

- Then, we summarize it, grouping and taking the survey component into account

```
> des %>%
+   group_by(year) %>%
+   summarise(
+     mean_age_w = survey_mean(age, vartype = c("se","ci")), # add se/ci
+     N_pop      = survey_total(vartype = "se"),  # weighted sample size
+     .groups = "drop"
+   )
```

A weighted summary dataset by year

- Finally, we compare

```
> des_sum_weighted
# A tibble: 3 × 7
   year mean_age_w mean_age_w_se mean_age_w_low mean_age_w_upp N_pop N_pop_se
  <dbl>      <dbl>          <dbl>          <dbl>          <dbl> <dbl>    <dbl>
1  2012       47.9           7.64           31.1           64.7   4.8     2.98
2  2016       34.8           3.46           27.2           42.4   3.3     1.63
3  2020       46.3           6.20           32.6           59.9   5.7     3.18

> des_sum_unweighted
# A tibble: 3 × 3
   year mean_age_w N_pop
  <dbl>      <dbl> <int>
1  2012       37.8     4
2  2016       32.5     4
3  2020       39.2     4
```

Reshape

Reshape: when you have mostly tidy data

- Recall: we want data to be tidy

	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

Reshape: when you have mostly tidy data

- Recall: we want data to be tidy
    - Unfortunately, data frequently disappoints us, arriving in all sorts of different contortions
    - We also (rarely) may want to put data into a different (less tidy) shape
- Let’s start with some definitions

Wide data

- Each subject/unit has a single row, with variables having separate columns
  - Different measures, units, times, etc are stored across (wide)
  - **Pros**: good for human readability
  - **Cons**: sometimes not idea for storage and use

	treat_a	treat_b
John	-	2
Jane	16	11
Mary	3	1

Long data

- Each observation has a single row
  - One column stores the variable name; another stores the value
  - **Pros**: often convenient for storage
  - **Cons**: poor for human readability

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

JSON data: long data

- On the right: long JSON data
- Below: the same data, wider

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
   }
  }
 ]
}
```

“Wide” and “long” are also not perfect ideal types

- Reality is often more complicated

	Year	City	Golds	Silver	Bronze
USA	2021	Tokyo	39	41	33
USA	2024	Paris	40	44	42
China	2021	Tokyo	38	32	19
China	2024	Paris	40	27	24

- What is the “unit of analysis” here?

Reshaping

- `reshape()` in base R
- `pivot_longer()` in Tidyverse
- `pivot_wider()` in Tidyverse

- The key is identifying two things:
  - What your unit of analysis currently is
  - What you want your unit of analysis to be
- And then, does this change indicate wider, or longer, data?

A very simple example

Two students (Alice, Bob), two classes (Statistics, Policy Implementation)

```
df <- tibble(
  student = c("Alice", "Bob"),
  stat    = c(90, 80),
  imp = c(85, 88)
)

> df
# A tibble: 2 × 3
  student  stat   imp
  <chr>   <dbl> <dbl>
1 Alice      90    85
2 Bob        80    88
>
```

	Statistics	Policy Implementation
Alice	90	85
Bob	80	88

A simple example - wide to long

```
> df_long <- df |>
+   pivot_longer(cols = c(stat, imp),
+                names_to = "subject",
+                values_to = "score")

> df_long
# A tibble: 4 × 3
  student subject score
  <chr>   <chr>   <dbl>
1 Alice   stat       90
2 Alice   imp        85
3 Bob     stat       80
4 Bob     imp        88
```

	stat	imp
Alice	90	85
Bob	80	88

	subject	score
Alice	stat	90
Alice	imp	85
Bob	stat	80
Bob	imp	88

And back again - long to wide

```
> df_long |>
+   pivot_wider(names_from = subject,
+               values_from = score)

# A tibble: 2 × 3
  student  stat   imp
  <chr>   <dbl> <dbl>
1 Alice      90    85
2 Bob        80    88
```

	subject	score
Alice	stat	90
Alice	imp	85
Bob	stat	80
Bob	imp	88

	stat	imp
Alice	90	85
Bob	80	88

We can also take more complex examples

country	year	city	gold	silver	bronze
USA	2021	Tokyo	39	41	33
USA	2024	Paris	40	44	42
China	2021	Tokyo	38	32	19
China	2024	Paris	40	27	24

```
medals_wide <- medals |>
  pivot_wider(
    id_cols = Country,
    names_from = Year,
    values_from = c(Gold, Silver, Bronze),
    names_glue = "{.value}_{Year}"
  )
```

country	year	city	gold	silver	bronze
USA	2021	Tokyo	39	41	33
USA	2024	Paris	40	44	42
China	2021	Tokyo	38	32	19
China	2024	Paris	40	27	24

country	Gold_2021	Gold_2024	Silver_2021	Silver_2024	Bronze_2021	Bronze_2024
USA	39	40	41	44	33	42
China	38	40	32	27	19	24

```
> medals_wide <- medals %>%
+   pivot_wider(
+     id_cols = Country,
+     names_from = Year,
+     values_from = c(Gold, Silver, Bronze),
+     names_glue = "{.value}_{Year}"
+   )
>
> medals_wide
# A tibble: 2 × 7
  Country       Gold_2021 Gold_2024 Silver_2021 Silver_2024 Bronze_2021 Bronze_2024
  <chr>             <dbl>     <dbl>       <dbl>       <dbl>       <dbl>       <dbl>
1 United States        39        40          41          44          33          42
2 China                38        40          32          27          19          24
```

Wrapping Up

- This time:
  - A variety of data management and wrangling techniques
  - For “mostly” tidy data
- Next time
  - Workshop: working with survey data; codebooks
  - Next week: VISUALIZATION