Lecture 3: Data Manipulation with `dplyr`

Harris Coding Camp – Accelerated Track

Summer 2023

Brainstorm: What types of actions do you need to work with data sets?

Data manipulation with `dplyr`

The `dplyr` library provides a toolkit for data manipulation.

Today will cover:

- `select()` to pick columns
- `filter()` to get rows that meet a criteria
- `arrange()` to order the data
- `mutate()` to create new columns
- `summarize()` to summarize data

As I show you examples, I'll work with variations of `txhousing` a data set built-in to `dplyr`

tidyverse origins: dplyr

```
library(tidyverse)
## -- Attaching packages ---------------------------
## v ggplot2 3.3.0  v purrr   0.3.4
## v tibble  2.1.3  v dplyr   0.8.5
## v tidyr   1.0.2  v stringr 1.4.0
## v readr   1.3.1  v forcats 0.5.0
## -- Conflicts ------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

selecting **columns**

select()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	pressure
Alberto	1007
Alex	1009
Allison	1005
Ana	1013
Arlene	1010
Arthur	1010

You want to present a subset of your **columns**

```
select(txhousing_short, city, date, sales, listings)
```

```
## # A tibble: 6 x 4
##   city     date sales listings
##   <chr>   <dbl> <dbl>    <dbl>
## 1 Abilene 2000     72      701
## 2 Abilene 2000.    98      746
## 3 Abilene 2000.   130      784
## 4 Abilene 2000.    98      785
## 5 Abilene 2000.   141      794
## 6 Abilene 2000.   156      780
```

select() extends [, col_expressions]

```
identical(
  select(txhousing, city, date, sales, listings),
  txhousing[, c("city", "date", "sales", "listings")],
)
```

```
## [1] TRUE
```

Compare:

-`select` can operate with column names while `[` requires characters. -`select` doesn't require column names to be in a single vector

exclude columns with `select()` and -

- says to exclude the columns listed in the vector.

```
select(txhousing_short, -c(city, date, sales, listings, yea
```

```
## # A tibble: 6 x 4
##   month   volume median inventory
##   <int>    <dbl>  <dbl>     <dbl>
## 1     1  5380000  71400       6.3
## 2     2  6505000  58700       6.6
## 3     3  9285000  58100       6.8
## 4     4  9730000  68600       6.9
## 5     5 10590000  67300       6.8
## 6     6 13910000  66900       6.6
```

tidyverse provides helpers for pulling out columns

I want a bunch of columns with similar names.

- use `starts_with()`, `ends_with()`, `contains()`
- or `matches()` with regular expressions

```
# baseR requires more coding knowledge
# txhousing[,grep("^city", names(txhousing))]

select(txhousing_short, ends_with("e"))
```

```
## # A tibble: 6 x 2
##     volume  date
##      <dbl> <dbl>
## 1  5380000 2000
## 2  6505000 2000.
## 3  9285000 2000.
## 4  9730000 2000.
## 5 10590000 2000.
## 6 13910000 2000.
```

Use case: You want to reorder your columns

- Notice we used a "select_helpers" function `everything()`.
- See also `dplyr` function `relocate()`

```
select(txhousing_short,
       year, month, date, everything())
```

```
## # A tibble: 6 x 9
##    year month  date city    sales   volume median listi
##   <int> <int> <dbl> <chr>   <dbl>    <dbl>  <dbl>    <db
## 1  2000     1 2000  Abilene    72  5380000  71400       7
## 2  2000     2 2000. Abilene    98  6505000  58700       7
## 3  2000     3 2000. Abilene   130  9285000  58100       7
## 4  2000     4 2000. Abilene    98  9730000  68600       7
## 5  2000     5 2000. Abilene   141 10590000  67300       7
## 6  2000     6 2000. Abilene   156 13910000  66900       7
```

select helpers only work with `select()`

You may see this error[1]

```
‘contains()‘ must be used within a *selecting* function.
See <https://tidyselect.r-lib.org/reference/faq-selection-c
```

For similar functionality outside of “selecting”, see the `stringr` package

[1] check out `?tidyselect::select_helpers()` and R4DS section on `select()`

choose **rows** that match a condition

filter()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Ana	40	1013	1997-07-01

choose rows that match a condition with `filter()`

Get all the data from 2013

```
filter(txhousing, year == 2013)
```

```
## # A tibble: 552 x 9
##    city     year month sales   volume median listings in
##    <chr>   <int> <int> <dbl>    <dbl>  <dbl>    <dbl>
##  1 Abilene  2013     1   114 15794494 125300      966
##  2 Abilene  2013     2   140 16552641  94400      943
##  3 Abilene  2013     3   164 19609711 102500      958
##  4 Abilene  2013     4   213 27261796 113700      948
##  5 Abilene  2013     5   225 31901380 130000      923
##  6 Abilene  2013     6   209 29454125 127300      960
##  7 Abilene  2013     7   218 32547446 140000      969
##  8 Abilene  2013     8   236 30777727 120000      976
##  9 Abilene  2013     9   195 26237106 127500      985
## 10 Abilene  2013    10   167 21781187 119000      993
## # ... with 542 more rows
```

filter() extends [row_expression,]

```
identical(
  filter(txhousing, year == 2013),
  txhousing[txhousing$year == 2013, ]
)
```

```
## [1] TRUE
```

Notice that `filter` can operate with column names while `[` requires that you use a vector.

`filter()` drops comparisons that result in `NA`

Compare:

```
df <- tibble(x = c(1, 2, NA))
filter(df, x > 1)
```

```
## # A tibble: 1 x 1
##       x
##   <dbl>
## 1     2
```

```
df[df$x > 1, ]
```

```
## # A tibble: 2 x 1
##       x
##   <dbl>
## 1     2
## 2    NA
```

When you think `filter()`, think comparison operator!

Recall: Comparison operators return TRUE or FALSE

Operator	Name
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to
%in%	matches something in

We've also seen `is.na()` to test for `NA`.

What does %in% do?

```
x <- c(1, 5, 3)

x %in% 5
```

```
## [1] FALSE  TRUE FALSE
```

```
x %in% c(1, 2, 3, 4)
```

```
## [1]  TRUE FALSE  TRUE
```

%in% operator is like a bunch of OR strung together

```
x <- c(1, 5, 3)

identical(
  # too much typing
  x == 1 | x == 2 | x == 3 | x == 4,

  x %in% c(1, 2, 3, 4)
)
```

```
## [1] TRUE
```

%in% operator is vectorized

Tests *element-by-element* whether items are in the right-side!

```
x <- c(1, 5, 3)
# 1 %in% c(1, 2, 3, 4)  TRUE
# 5 %in% c(1, 2, 3, 4)  FALSE
# 3 %in% c(1, 2, 3, 4)  FALSE
x %in% c(1, 2, 3, 4)
```

```
## [1]  TRUE FALSE  TRUE
```

Get all the data from 2013 and beyond for Houston.

- in `filter()` additional match criteria are treated like `and`

```
  filter(txhousing_narrow,
         year >= 2013,
         city == "Houston")
```

```
## # A tibble: 3 x 2
##   city     year
##   <chr>   <int>
## 1 Houston  2013
## 2 Houston  2014
## 3 Houston  2015
```

To do the same operation with [. . .

```
identical(
  filter(txhousing,
         year >= 2013,
         city == "Houston"),

txhousing[txhousing$year >= 2013 &
                    txhousing$city == "Houston", ]
)
```

```
## [1] TRUE
```

Why do we get 0 rows here?

Get all the data from 2013 and beyond for Houston and Austin

```
  filter(txhousing_narrow,
         year >= 2013,
         city == "Houston",
         city == "Austin")
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: city <chr>, year <int>
```

There's no rows where `city` is both Houston AND Austin!

We logically want data from Houston OR Austin

```
  filter(txhousing_narrow,
         year >= 2013,
         city == "Houston" | city == "Austin")
```

```
## # A tibble: 6 x 2
##   city     year
##   <chr>   <int>
## 1 Austin   2013
## 2 Austin   2014
## 3 Austin   2015
## 4 Houston  2013
## 5 Houston  2014
## 6 Houston  2015
```

At some point you will make this mistake!

```
  filter(txhousing_narrow,
         year >= 2013,
         city == "Houston" | "Austin")
```

```
Error in filter(txhousing, year >= 2013, city == "Houston"
Caused by error in `city == "Houston" | "Austin"`:
! operations are possible only for numeric, logical or comp
```

What if we want data from Houston, Austin OR Galveston

```
  filter(txhousing,
         year >= 2013,
         city == "Houston" | city == "Austin" | city == "Ga
```

There has to be an easier way!

Use %in%!

```
in_three_cities <-
  filter(txhousing,
         year >= 2013,
         city %in% c("Houston", "Dallas", "Austin"))
```

Why does it fail to produce the same result?

```
eq_three_cities <-
      filter(txhousing,
             year >= 2013,
             city == c("Houston", "Dallas", "Austin"))
```

```
## Warning in city == c("Houston", "Dallas", "Austin"): lon
## a multiple of shorter object length
```

```
identical(in_three_cities, eq_three_cities)
```

```
## [1] FALSE
```

```
nrow(in_three_cities)
```

```
## [1] 93
```

```
nrow(eq_three_cities)
```

```
## [1] 30
```

Be wary of vector recycling.

== with vectors of different length is usually a bad idea.

```
ex <- tibble(id = 1:4,
       attribute = c("a", "a", "b", "b"))


ex
```

```
## # A tibble: 4 x 2
##      id attribute
##   <int> <chr>
## 1     1 a
## 2     2 a
## 3     3 b
## 4     4 b
```

```
filter(ex, attribute == c("a", "c"))
```

```
## # A tibble: 1 x 2
##      id attribute
```

Be wary of vector recycling.

```
# a == a
# a == c
# b == a
# b == c

filter(ex, attribute == c("a", "c"))
```

```
## # A tibble: 1 x 2
##       id attribute
##    <int> <chr>
## 1      1 a
```

Another win for %in%

```
# a %in% c(a, c)
# a %in% c(a, c)
# b %in% c(a, c)
# b %in% c(a, c)

filter(ex, attribute %in% c("a", "c"))
```

```
## # A tibble: 2 x 2
##      id attribute
##   <int> <chr>
## 1     1 a
## 2     2 a
```

`dplyr` concept: Data in, Data out

Notice that `filter()` and `select()`

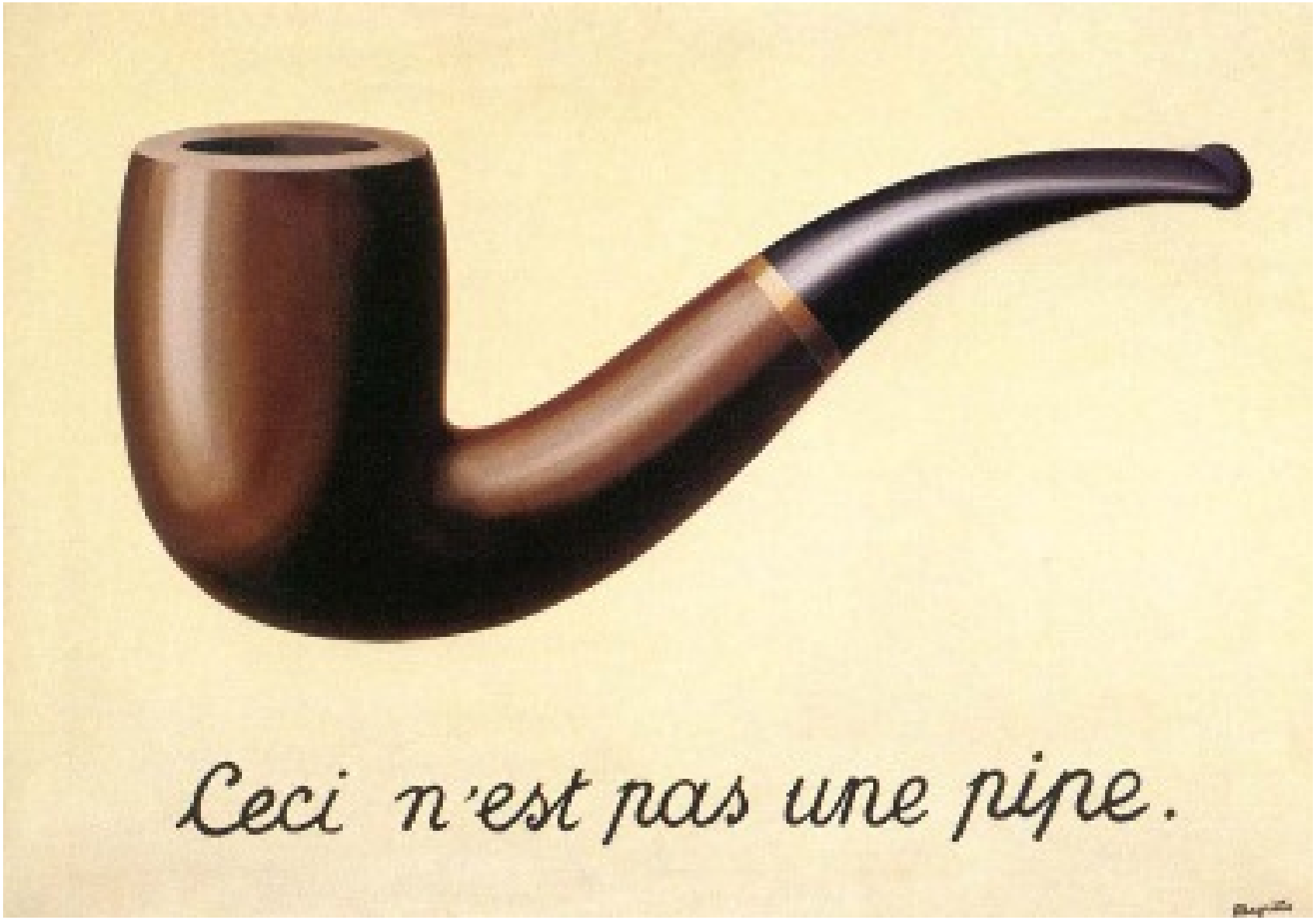
- data in the first position
- `...` in the second position (i.e. allows for arbitrary number of inputs)
- return data

`dplyr` functions take in a tibble and return a tibble.

- This allows us to chain together data-moves without creating clutter

But how do we chain together functions?

Introducing the pipe operator



Ceci est une |>

The pipe |> operator takes the left-hand side and makes it *input* in the right-hand side.

- by default, LHS is *first argument* of the RHS function.

```
# a tibble is the first argument
select(txhousing, city, year, sales, volume)

txhousing |>
  select(city, year, sales, volume)
```

Read |> as "and then.

```
# Take data
txhousing |>
  # And then select city, year, month and median
  select(city, year, month, median) |>
  # And then filter where year is 2013
  filter(year == 2013) |>
  # And then show the head  (i.e. first 6 rows)
  head()
```

Chaining avoids intermediate data frames!

- Coming up with names is hard.
- Updating an object repeatedly leads to a *buggy* development process

```
txhousing |>
  select(city, year, month, median) |>
  filter(year == 2013) |>
  head(3)
```

```
## # A tibble: 3 x 4
##   city     year month median
##   <chr>   <int> <int>  <dbl>
## 1 Abilene  2013     1 125300
## 2 Abilene  2013     2  94400
## 3 Abilene  2013     3 102500
```

Updating an object repeatedly -> *buggy* code

```
txhousing <- read_csv(...)

# Code in a different chunk
txhousing <- txhousing |>
  mutate(important_new_col =  ..code..)

# Code in yet a different chunk
txhousing <- txhousing |>
  filter(important_new_col < 10)
```

Error in `filter()`: . . .

! object 'important_new_col' not found

Treat coding like writing

- start with a rough draft, but polish as you go.
- put code that defines/manipulates an object close together
  - like writing a tight paragraph
  - the name is the topic sentence

```
# Sometimes reading data is slow, so I have this habit
txhousing_raw <- read_csv(...)

# Now only reference to with name txhousing
# all in one chunk
txhousing <- txhousing_raw |>
  mutate(important_new_col =  ..code..) |>
  filter(important_new_col < 10)
```

%>% is also a pipe.

- %>% was the original pipe in R
  - in `magrittr` package
  - still loads when with `library(tidyverse)`
- |> was added to baseR in version 4.1

%>% has more bells and whistles, which can be a liability

sort rows

arrange()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date
Ana	40	1013	1997-07-01
Alex	45	1009	1998-07-30
Arthur	45	1010	1996-06-21
Arlene	50	1010	1999-06-13
Allison	65	1005	1995-06-04
Alberto	110	1007	2000-08-12

sort rows with arrange()

```
identical(
  # base R
  txhousing[order(txhousing$year), ],

  # dplyr
  arrange(txhousing, year)
)
```

```
## [1] TRUE
```

sort rows with `arrange()`

To sort in `desc()`ending order.

```
identical(
  # base R
  txhousing[order(-txhousing$year), ],

  # dplyr
  txhousing %>% arrange(desc(year))
)
```

```
## [1] TRUE
```

sort rows with arrange()

- arrange can take multiple columns

```
txhousing_example %>%
  arrange(year, month, desc(volume)) %>%
  head()
```

```
## # A tibble: 6 x 5
##   city     year month sales    volume
##   <chr>   <int> <int> <dbl>     <dbl>
## 1 Houston  2000     1  2653 381805283
## 2 Dallas   2000     1  2286 375389865
## 3 Dallas   2000     2  3247 555124812
## 4 Houston  2000     2  3687 536456803
## 5 Houston  2000     3  4733 709112659
## 6 Dallas   2000     3  4244 702148377
```

create columns

mutate()

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

storm	wind	pressure	date	ratio	inverse
Alberto	110	1007	2000-08-12	9.15	0.11
Alex	45	1009	1998-07-30	22.42	0.04
Allison	65	1005	1995-06-04	15.46	0.06
Ana	40	1013	1997-07-01	25.32	0.04
Arlene	50	1010	1999-06-13	20.20	0.05
Arthur	45	1010	1996-06-21	22.44	0.04

mutate(.data, ...) works like other dplyr verbs

```
# the data in the first position
txhousing_example |>
  # after that ... create columns like so
  mutate( volume_millions = volume / 1e6)  |>
  head()
```

```
## # A tibble: 6 x 6
##   city    year month sales    volume volume_millions
##   <chr>  <int> <int> <dbl>     <dbl>           <dbl>
## 1 Dallas  2000     1  2286 375389865            375.
## 2 Dallas  2000     2  3247 555124812            555.
## 3 Dallas  2000     3  4244 702148377            702.
## 4 Dallas  2000     4  3977 667331427            667.
## 5 Dallas  2000     5  4545 783197806            783.
## 6 Dallas  2000     6  4738 846254912            846.
```

creating columns with `mutate()`

When we mutate, you can create new columns.

- *Right-hand side*: the name of a new column.
- *Left-hand side*: code that creates a vector
  - no quotes and no reference to the data's name

```
txhousing_example |>
  mutate(volume_millions = volume / 1e6) |>
  head()
```

```
## # A tibble: 6 x 6
##   city    year month sales    volume volume_millions
##   <chr>  <int> <int> <dbl>     <dbl>           <dbl>
## 1 Dallas  2000     1  2286 375389865            375.
## 2 Dallas  2000     2  3247 555124812            555.
## 3 Dallas  2000     3  4244 702148377            702.
## 4 Dallas  2000     4  3977 667331427            667.
## 5 Dallas  2000     5  4545 783197806            783.
## 6 Dallas  2000     6  4738 846254912            846.
```

Compare to base R $ <-

```
mutate(txhousing,
       volume_millions = volume / 1e6)

txhousing$volume_millions <-
  txhousing$volume / 1e6
```

- baseR operates on vectors directly, requires assignment <-

dplyr functions know that names refer to columns

```
identical(
# BAD: extracting the column (not as nice)
txhousing_example |>
  mutate(volume_millions = txhousing_example$volume / 1e6),

# GOOD: referring to the column by name!
txhousing_example |>
  mutate(volume_millions = volume / 1e6)
)
```

```
## [1] TRUE
```

dplyr verbs allow many updates at once

- with `mutate()`, create multiple columns
- use information from a newly created column
  - code evaluated in order from top to bottom.

```
txhousing_example |>
  mutate(mean_price = volume / sales,
         sqrt_mean_price = sqrt(mean_price)) %>%
  head()
```

```
## # A tibble: 6 x 7
##   city    year month sales    volume mean_price sqrt_mea
##   <chr>  <int> <int> <dbl>     <dbl>      <dbl>
## 1 Dallas  2000     1  2286 375389865    164213.
## 2 Dallas  2000     2  3247 555124812    170965.
## 3 Dallas  2000     3  4244 702148377    165445.
## 4 Dallas  2000     4  3977 667331427    167798.
## 5 Dallas  2000     5  4545 783197806    172321.
## 6 Dallas  2000     6  4738 846254912    178610.
```

The change is not permanent

Until you assign the output tibble to a name!

```
texas_housing_again <-
txhousing_example |>
  mutate(mean_price = volume / sales,
         sqrt_mean_price = sqrt(mean_price))
```

You try it.

If you load `tidyverse`, you can access the `midwest` data

What dplyr function would you need to ...

- choose the columns `county`, `state`, `poptotal`, `popdensity`
- get the counties with population over a million
- reorder the columns by population total
- round the popdensity to the nearest whole number

You try it

- `select()` the columns `county`, `state`, `poptotal`, `popdensity`
- `filter()` the counties with population over a million
- `arrange()` the columns by population total
- `mutate()` to round the `popdensity` to the nearest whole number
- AND `mutate()` to round the `population` totals to the nearest 1000

if you finish early: Try to write it in base R

```
## # A tibble: 4 x 4
##   county    state poptotal popdensity
##   <chr>     <chr>    <dbl>      <dbl>
## 1 COOK      IL     5105000      88018
## 2 WAYNE     MI     2112000      60334
## 3 CUYAHOGA  OH     1412000      54313
## 4 OAKLAND   MI     1084000      19702
```

solution

```
midwest %>%
  select(county, state, poptotal, popdensity) %>%
  filter(poptotal > 1e6) %>%
  arrange(desc(poptotal)) %>%
  mutate(popdensity = round(popdensity),
         poptotal = poptotal - poptotal %% 1000
  # alternatively:
  #     poptotal = round(poptotal, -3)
  )
```

solution in base R

```
out <- midwest[midwest$poptotal > 1e6,
        c("county", "state", "poptotal", "popdensity")]

out$popdensity <- round(out$popdensity)
out$poptotal <- out$poptotal - out$poptotal %% 1000

out[order(out$poptotal, decreasing = TRUE), ]
```

```
## # A tibble: 4 x 4
##   county    state poptotal popdensity
##   <chr>     <chr>    <dbl>      <dbl>
## 1 COOK      IL     5105000      88018
## 2 WAYNE     MI     2111000      60334
## 3 CUYAHOGA  OH     1412000      54313
## 4 OAKLAND   MI     1083000      19702
```

summarize data with `summarize()`

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

median
22.5

Calculate total volume of sales in Texas from 2014.

```
txhousing %>%
  filter(year == 2014) %>%
  summarize(total_volume = sum(volume))
```

```
## # A tibble: 1 x 1
##   total_volume
##          <dbl>
## 1   84760948831
```

```
# take sum of txhousing and subset so year is 2014 and col
sum(txhousing[txhousing$year == 2014,"volume"])
```

```
## [1] 84760948831
```

```
# take the volume column and subset values where year is 2
sum(txhousing$volume[txhousing$year == 2014])
```

```
## [1] 84760948831
```

Calculate the mean and median number of sales in Texas's three largest cities.

```
txhousing |>
  filter(city %in%
           c("Houston", "Dallas", "San Antonio")) |>
  summarize(median_n_sales = median(sales),
            mean_n_sales = mean(sales))
```

```
## # A tibble: 1 x 2
##   median_n_sales mean_n_sales
##            <dbl>        <dbl>
## 1           3996        3890.
```

summarize data with `summarize()`

There are many useful functions that go with summarize. Try `?summarize` for more.

```
txhousing %>%
  summarize(n_obs = n(),
            n_cities = n_distinct(city))
```

```
## # A tibble: 1 x 2
##   n_obs n_cities
##   <int>    <int>
## 1  8602       46
```

Alert: `summarize()` without summarizing

Weird behavior:

```
# in older versions of dplyr this gives an error
# Error: Column `mean_price` must be length 1 (a summary v

txhousing %>%
  summarize(mean_price = volume / sales) %>%
  head()
```

piping dplyr verbs together

dplyrverbs can be piped together in any order you want, although different orders can give you different results, so be careful!

```
txhousing |>
  select(city, year, month, sales, volume) |>
  mutate(log_mean_price = log(volume / sales)) |>
  filter(year == 2013) |>
  summarize(log_mean_price_2013 = mean(log_mean_price,
                                       na.rm = TRUE))

# Won't give you the same result as
# txhousing %>%
#  select(city, year, month, sales, volume) %>%
#  mutate(log_mean_price = log(volume / sales)) %>%
#  summarize(log_mean_price = mean(log_mean_price, na.rm = TRUE)) %>%
#  filter(year == 2013)

# Actually this code will give you an error, try it!
```

Recap: manipulating data with `dplyr`

We learned

- how to employ the Big 5 `dplyr` verbs
  - `select()` to pick columns
  - `arrange()` to order the data
  - `mutate()` to create new columns
  - `filter()` to get rows that meet a criteria
  - `summarize()` to summarize data
- how to use relation operators, binary operators for math and logical operators in `dplyr` contexts

Next steps:

Lab:

- Today lab: practice with `dplyr` verbs (and base R manipulation)
- Tomorrow lab: more practice in data manipulation

Touchstones: I can comfortably manipulate data[2]

Next lecture:

- Using `if` and `ifelse`
- We'll have a completely low stakes quiz to help surmise how we're doing

[2] i.e. adjust or add columns to data, subset it in various ways, sort it as needs be and make summary tables.