# Lecture 3: Data Manipulation with `dplyr`

Harris Coding Camp – Accelerated Track

Summer 2022

# dplyr

# Data manipulation with `dplyr`

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

We 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

# 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()
```

# dplyr concepts

For each function:

- data in the first position
- `...` in the second position (i.e. allows for many changes at once)

As I show you examples, I'll work with variations of `txhousing` `txhousing_short`, `txhousing_narrow` and `txhousing_example`

# selecting columns with `select()`

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 |

# selecting columns with `select()`

Use case: 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
```

# think: `select()` extends `[, col_expressions]`

`select` provides a suite of short cuts for selecting columns.

- Notice that `select` can operate with column names while `[` requires Characters.

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

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

# selecting columns with `select()`

As with `[`, `-` says to exclude the columns listed in the vector.

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

# selecting columns with `select()`

Use case: I want a bunch of columns with similar names.

- There are several useful functions, see `?tidyselect::select_helpers`.
- (For more information see r4ds chapter 5.4)

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

select(txhousing_short, starts_with("city"))
```

```
## # A tibble: 6 x 1
##   city
##   <chr>
## 1 Abilene
## 2 Abilene
## 3 Abilene
## 4 Abilene
## 5 Abilene
```

# selecting columns with `select()`, helpers

Use case: You want to reorder your columns

- Notice we used a "select_helpers" function `everything()`.

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

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

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

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
```

# think: `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
```

# Recall: Relational operators return TRUE or FALSE

Before moving forward with `filter()`, let's review relational operators and logical operators

| 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 |

# Recall: Relational operators in practice

```
4 < 4
```

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

```
4 >= 4
```

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

```
4 == 4
```

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

```
4 != 4
```

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

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

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

# %in% operator

Tests if left-hand side is in right-hand side.

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

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

It operates *element-by-element* on the left-hand side!

# Review: logical operators combine TRUEs and FALSEs logically

| Operator | Name |
|---|---|
| ! | not |
| & | and |
| \| | or |

```
# not true
! TRUE
```

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

```
# are both x & y TRUE?
TRUE  &  FALSE
```

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

```
# is either x | y TRUE?
TRUE | FALSE
```

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

# Review: What do the following return?

```
! (4 > 3)
(5 > 1) & (5 > 2)
(4 > 10) | (20 > 3)
```

# Review: What do the following return?

Logical operators team up with relational operators.

- First, evaluate the relational operator
- Then, carry out the logic.

```
! (4 > 3)                # ! TRUE
(5 > 1) & (5 > 2)        # TRUE & TRUE
(4 > 10) | (20 > 3)      # FALSE | TRUE
```

# Recall: What do the following return?

```
! (4 > 3)                    # ! TRUE
```

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

```
(5 > 1) & (5 > 2)            # TRUE & TRUE
```

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

```
(4 > 10) | (20 > 3)          # FALSE | TRUE
```

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

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

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
```

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

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>
```

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

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
```

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

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
```

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

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!

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

Use %in%!

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

# Warning: == with two vectors is a bad idea.

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)
```

# Warning: == with two vectors is a bad idea.

Why not? Vector recycling.

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

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

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

# Warning: == with two vectors is a bad idea.

Why not? 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
```

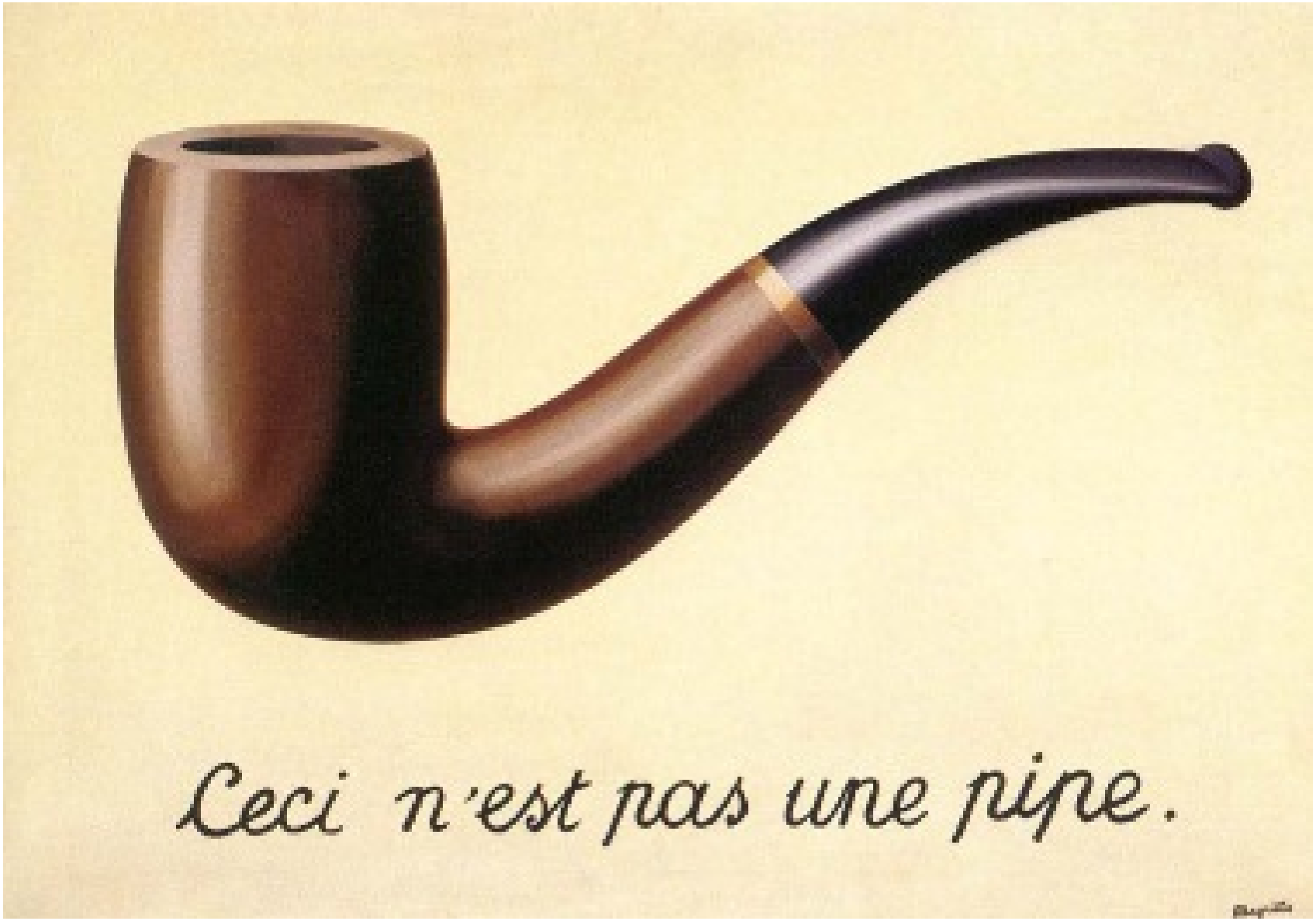
# Crisis averted

```
# 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
```

# Introducing the pipe operator



# Interlude: Ceci est une %>%

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

- by default, the left-hand side is the *first argument* of the right-hand side function.

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

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

# Ceci est une %>%

We can chain together tidyverse functions to avoid making so many 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()
```

```
## # A tibble: 6 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
## 4 Abilene  2013     4 113700
```

# Ceci est une %>%

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()
```

# sort rows with `arrange()`

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), ],

  # base R
  arrange(txhousing, year)
)
```

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

# sort rows with arrange()

To sort in desc()ending order.

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

  # base R
  txhousing %>% arrange(desc(year))
)
```

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

# creating columns with `mutate()`

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 |

# creating columns in Base R

We've hinted at how to add columns in base R.

What would you guess based on how we added values to lists or changed values in vectors?

# creating columns in Base R

1. Refer to a column with `$`.[1]
2. Assign a vector to the name

```
# convert the dollar amount to millions
txhousing$volume_millions <-
  round(txhousing$volume / 1000000, 1)
```

[1] or `[[`

# creating columns in Base R

Since we used an assignment operator. The change is permanent.

```
txhousing %>%
  select(starts_with("volume")) %>%
  head()
```

```
## # A tibble: 6 x 2
##     volume volume_millions
##      <dbl>           <dbl>
## 1  5380000             5.4
## 2  6505000             6.5
## 3  9285000             9.3
## 4  9730000             9.7
## 5 10590000            10.6
## 6 13910000            13.9
```

# All my x's live in Texas

Vectors of size 1 are recycled.

```
txhousing$state <- "Texas"

head(txhousing[, c("city", "state")])
```

```
## # A tibble: 6 x 2
##   city    state
##   <chr>   <chr>
## 1 Abilene Texas
## 2 Abilene Texas
## 3 Abilene Texas
## 4 Abilene Texas
## 5 Abilene Texas
## 6 Abilene Texas
```

# creating columns with `mutate()`

`mutate` works like other dplyr verbs

```
# the data in the first position
mutate(txhousing,
       # after that ... create columns like so
       volume_millions = volume / 1e6)

# same idea
txhousing$volume_millions <-
  txhousing$volume / 1e6
```

- As with other dplyr verbs, we can refer to the columns by name

# creating columns with `mutate()`

```
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
```

# creating columns with `mutate()`

```
txhousing_example %>%
  mutate(mean_price = volume / sales)  %>%
  head()
```

```
## # A tibble: 6 x 6
##   city     year month sales   volume mean_price
##   <chr>   <int> <int> <dbl>    <dbl>      <dbl>
## 1 Abilene  2000     1    72  5380000     74722.
## 2 Abilene  2000     2    98  6505000     66378.
## 3 Abilene  2000     3   130  9285000     71423.
## 4 Abilene  2000     4    98  9730000     99286.
## 5 Abilene  2000     5   141 10590000     75106.
## 6 Abilene  2000     6   156 13910000     89167.
```

# 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

```
txhousing_example %>%
  mutate(mean_price = volume / sales) %>%
  head()
```

```
## # A tibble: 6 x 6
##   city     year month sales   volume mean_price
##   <chr>   <int> <int> <dbl>    <dbl>      <dbl>
## 1 Abilene  2000     1    72  5380000     74722.
## 2 Abilene  2000     2    98  6505000     66378.
## 3 Abilene  2000     3   130  9285000     71423.
## 4 Abilene  2000     4    98  9730000     99286.
## 5 Abilene  2000     5   141 10590000     75106.
## 6 Abilene  2000     6   156 13910000     89167.
```

# creating columns with `mutate()`

You can create multiple columns at a single time and even use information from a newly created column as input.

```
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 Abilene  2000     1    72  5380000     74722.
## 2 Abilene  2000     2    98  6505000     66378.
## 3 Abilene  2000     3   130  9285000     71423.
## 4 Abilene  2000     4    98  9730000     99286.
## 5 Abilene  2000     5   141 10590000     75106.
## 6 Abilene  2000     6   156 13910000     89167.
```

# creating columns with `mutate()`

Of course, if you want to change the data, you need to assign the output to a name!

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

# Pop quiz

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)
  #      poptotal = poptotal - poptotal %% 1000
  )
```

# 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
```

# solution in base R

- You may also see some base R functions that are antecedents to dplyr
- (comparison of subset and filter on stackoverflow)[https://stackoverflow.com/questions/39882463/difference-between-subset-and-filter-from-dplyr]

```
# subset: filters and selects referring to names
out <- subset(midwest, poptotal > 1e6,
        c(county, state, poptotal, popdensity))


# within: mutate but with unfortunate syntax
out <- within(out,{
                popdensity <- round(popdensity)
                poptotal <- poptotal - poptotal %% 1000
              })

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

# 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 |

# summarize data with `summarize()`

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
```

# summarize data with `summarize()`

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 5 `dplyr` verbs of highest importance including
  - `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`

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