

summer_2021_qa

Ari Anisfeld

9/1/2021

Class 1: Reading files and 'dplyr'

Do now

Do now:

- ▶ Complete the intro poll at `bit.ly/acc_intro_poll`

After the poll

- ▶ Download lab_1 from the course webpage:
`harris-coding-lab.github.io`.

Notice

- ▶ Earlier we had a typo for the link to lab_0 on canvas, which is now fixed. Sorry for inconvenience.

Expectations

From you:

- ▶ do the work (i.e. watch video, try basics, do lab, bring questions)
- ▶ engage in course! (i.e. work with partners, answer questions, do polls)
- ▶ have R and RStudio installed!

From us:

- ▶ prepare engaging lesson materials
- ▶ address your questions
- ▶ help you be confident for core (confident \neq R expert)

From everyone:

- ▶ be nice to each other and create a growth-focused environment

Do the work

- ▶ Step 1. Videos
- ▶ Step 1a. Basics
- ▶ Step 2. QA
- ▶ Step 3. Lab

Not an expert

We cover:

- ▶ how to work with basic data structures (tibbles, vectors)
- ▶ how to read and manipulate data
- ▶ programmer logic (if statements, loops, functions)

We won't cover in depth:

- ▶ most statistical tools
- ▶ how to join data together
- ▶ how to convert data from long to wide (pivoting)
- ▶ how to deal with very messy data
- ▶ how to work with specific data types (e.g. dates, advanced strings)
- ▶ among other things like webscrapping, package development and so forth

Today's session

- ▶ Set up working directories
- ▶ Review some questions from QA
- ▶ Highlight key points and open up for live questions

Setting up working directory and coding environment

1. Do you have a folder on your computer for coding lab material?
If not, create one and make sure you know the path to the folder.
2. We recommend creating a `problem_set` folder inside your coding lab folder.
3. Make folder called `data` inside the `problem_set` folder.

Putting your files in place

4. Create a new R script. Save your script in the `problem_set` folder. From now on, when you start a script or Rmd save it there.
5. Download the first data set [here](#) and put the data in your data folder. Find the link in the lab pdf!

Tell R where to find files

- ▶ Local paths are like addresses on your computer. Use `getwd()` to see how your computer paths look.

```
# In lab0 we downloaded data form a URL which is an address on the inte
covid_data <-
  read_csv(
    "https://data.cdc.gov/api/views/qfhf-uhaa/rows.csv?"
  )

# Compare to a local path
wid_data <-
  read_xlsx(
    "~/coding-lab/harris-coding-lab.github.io/data/world_wealth_inequal
  )
```

6. Add a line to your script where you `setwd()` to the data folder.

Working with the files

7. Finally, we are using data in an excel format. We need the package `readxl` to process data of this type. In the console, run `install.packages("readxl")`.
8. Add code to load the `tidyverse`.
9. If you followed the set-up from above, you should be able to run the following code with no error.

```
wid_data <- read_xlsx("world_wealth_inequality.xlsx")
```

What to do when something is confusing?

- ▶ use ?
- ▶ test code in console. try to break it.
- ▶ ask teammates / try googling
- ▶ ask us!

If it's not “mission critical”, you can safely move on without full understanding. (Imagine learning a language and trying to figure out all the grammar and vocabulary at the same time!)

Question:

- ▶ What's the deal with `col_types = cols(Suppress = col_character())`?
- ▶ Do we need that “`accessType=DOWNLOAD&bom=true&format=true%20target=`” part?

```
covid_data <-  
  read_csv(  
    paste0("https://data.cdc.gov/api/views/qfhf-uhaa/rows.csv?",  
           "accessType=DOWNLOAD&bom=true&format=true%20target="),  
    col_types = cols(Suppress = col_character()))
```

- ▶ Note: In URLs after the ? you send meta information about your request.

Question: Can you explain pipes?

- ▶ Pipes `%>%` take the left hand side and put them into the first position on the right hand side.

```
storms %>% filter(year > 2010) %>% glimpse()

recent_storms <- filter(storms, year > 2010)
glimpse(recent_storms)
```

Notice

- ▶ `filter()` takes data in the first position and then an arbitrary number of filtering expressions.
- ▶ `glimpse()` takes data in the first position

Lesson 0: Intro to R, RStudio and the tidyverse

- ▶ navigate and use Rstudio's features
 - ▶ particularly, the console, the text editor and help
- ▶ assign objects to names with `<-`
- ▶ use functions by providing inputs and learn more with `?`
- ▶ `install.packages()` (once) and then load them with `library()` (each time you restart R)

Lesson 1: Key points: Reading files

- ▶ Tabular data is stored in a lot of different formats.
 - ▶ e.g. `.csv`, `.xlsx`, `.dta`
- ▶ Read tabular data of a given type with the proper function.
 - ▶ e.g. for `csvs` we have `read_csv()`
 - ▶ If you get a new type, Google “How to read xxx files into R tidyverse”.
- ▶ We need to be aware of the file path and can `setwd()`.
- ▶ We know there are useful tools built into the `read_xxx()` functions.
 - ▶ Though we just scratched the surface.

Lesson 1: Manipulating data with `dplyr()`

- ▶ Choose columns with `select()`.
- ▶ Choose rows based on a match criteria with `filter()`.
 - ▶ We were introduced to comparison operators like `==` and `%in%`.
- ▶ Make new columns with `mutate()`.
- ▶ Sort data with `arrange()` and `arrange(desc())` or `arrange(-x)`.
- ▶ Create summary statistics with `summarize()`.

Class 2: Vectors and data types

Course logistics:

- ▶ When should we start working on the final project?
 - ▶ Start looking for a dataset now.
 - ▶ Write code to read it into R and start investigating with `dplyr` verbs.
 - ▶ Ask simple questions that can be addressed with your current tools.

lab 1 solutions will be available on the course website.

Getting started with Rmarkdown (Rmd)

- ▶ What's an Rmd?
- ▶ How to make an Rmd
- ▶ How to work with an Rmd

Knitting: making the frustrating part less frustrating

- ▶ Install tinytex

```
install.packages("tinytex")  
tinytex::install_tinytex()
```

- ▶ Knit early and often.

When to use Rmds vs scripts?

Rmd

- ▶ Exploration of data
- ▶ Presentations and reports

script

- ▶ Projects with interrelated code (e.g. an R package)
- ▶ Working on a server that does not have Rstudio installed

Questions from QA

Question 1: - Why do I need the function `summarize` in the following bit of code?

```
michigan_population_total <-  
  midwest %>%  
    filter(state == "MI") %>%  
    summarize(total_pop = sum(poptotal))
```

- ▶ Why can't I just pipe directly into `sum`?

Question 2: Why do we need to use `pull()`?

R's primary data structures: Vectors vs. tibbles

- ▶ Why do we need different data structures?
- ▶ How are vectors and tibbles related?

Why do we need different data structures?

In theory, we can do all our work with vectors.

```
names_vec <- c("Ari", "Qiwei", "Jay", "Thomas")  
surnames_vec <- c("Anisfeld", "Lin", "Zaleski", "Whamond")  
position_vec <- c("Instructor", "TA", "TA", "TA")
```

why might I want a tibble?

R's primary data structures: Vectors vs. tibbles

Tibbles encapsulate vectors

```
names_vec <- c("Ari", "Qiwei", "Jay", "Thomas")
surnames_vec <- c("Anisfeld", "Lin", "Zaleski", "Whamond")
positions_vec <- c("Instructor", "TA", "TA", "TA")

my_tibble <- tibble(names = names_vec,
                    surnames = surnames_vec,
                    positions = positions_vec)
```

- ▶ Keep data tidy -> rows are a single observation or record.
- ▶ Keep meta-data (column names)
- ▶ Keep track of relationships between vectors
- ▶ Can hold various data types

R's primary data structures: Vectors vs. tibbles

- ▶ vectors are simpler
- ▶ have a single data type
- ▶ some functions expect vectors or make more sense on them.

Reviewing automatic type coercion

Type coercion is done automatically when R knows how. Usually, simpler types can be coerced to more complex types.

► `logical < integer < double < character`.

```
# paste0() is a function that combines  
# two chr vectors into a single vector  
paste0("str", "ing")
```

```
## [1] "string"
```

```
paste0(1L, "ing")
```

```
## [1] "1ing"
```

1L is an `int`, but R will coerce it into a `chr` in this context.

Automatic coercion

Logicals are coercible to numeric or character. This is very useful!

Determine the rule for how R treats TRUE and FALSE in math.

```
TRUE + 4
```

```
FALSE + 4
```

```
sum(c(FALSE, FALSE, FALSE, FALSE))
```

```
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

Automatic coercion

```
TRUE + 4
```

```
## [1] 5
```

```
FALSE + 4
```

```
## [1] 4
```

```
sum(c(FALSE, FALSE, FALSE, FALSE))
```

```
## [1] 0
```

```
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

```
## [1] 0.6
```

Exercise

1: Use R to calculate the sum

$$\sum_{n=0}^{10} \frac{1}{2^n} = \frac{1}{2^0} + \frac{1}{2^1} + \dots + \frac{1}{2^{10}}$$
$$\sum_{n=0}^{10} \frac{1}{2^n} = 1 + 0.5 + \dots + 0.00098$$

1. Use vectorized math to create a vector with the correct numbers
2. Use a built-in function to add up all the numbers in the vector.

Bonus What happens to the sum as you increase n ?

2: Use `paste0()` to convert `v1` and `v2` into "hello!"

```
v1 <- c("h", "l", "o")  
v2 <- c("e", "l", " !")
```

```
## [1] "h el lo  !"
```

Key points: Class 2 vectors and data types

vectors and vectorized coding

- ▶ Vectors are the fundamental way to store data in R
- ▶ We can operate on vectors element-by-element without loops
 - ▶ `dplyr` verbs rely on this!
- ▶ We introduced built-in functions to build vectors and do operations on vectors.

data types

- ▶ (Atomic) Vectors have a single data type
 - ▶ most often: `logical`, `integer`, `double`, or `character`
- ▶ Certain operations expect a certain data type and will try to coerce the data if it can.
 - ▶ coercion can lead to unexpected behavior such as making `NAs`.

Over weekend: Attempt lab 2. **For Tuesday:** Watch video about control flow + try basics.

Class 3: Control flow

Outline

- ▶ lingering questions about Rmds
- ▶ `ifelse()` questions

Reviewing the anatomy of an Rmd:

Write text in the document

```
# ^^^ start an R chunk '''{r}  
# sometimes {} have meta information  
  
# R code goes in a chunk  
ex <- seq(1, 12)  
  
# and output prints below  
print(ex)
```

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

A meta examples: {r, echo = FALSE}

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

Another meta examples: {r, eval = FALSE}

```
print(ex)
```

Naming chunks

Chunk names help you debug

```
# code
```

Two chunks cannot have the same name

```
# If I change the name to "example" we will get an error.
```

Quick hitters

Q: Why don't we get LaTeX with \$ in this example?

```
$$  
  \sqrt{p}  
$$
```

Quick hitters

Q: Why don't we get LaTeX with \$ in this example? A: In a code chunk, knitr expects R code! So a dollar sign is interpreted to be referring to a named entity in an object like `data$column_name`.

So put LaTeX in the “text” area of an Rmd!

$$\sqrt{p}$$

Quick hitters

Q: What does this error tell you?

```
setwd("~/Documents/coding")
```

Error in setwd("~/Documents/coding") : cannot change working directory

Quick hitters

Q: What does this error tell you? A: Usually, it means your directory name is wrong somehow!

```
setwd("~/Documents/coding_lab/")
```

Note: “~/” does not work on Windows!

An aside on relative paths

You can refer to directories **relative** to your current directory using `.` and `..`.

- ▶ `.` means current directory.
- ▶ `..` means “go back” the directory path.

an example

File structure:

- ▶ coding_lab datafile.csv
 - ▶ problem_sets my_rmd.Rmd

You could access the data with `read_csv("../datafile.csv")` in a chunk.

What's the difference between & and &&?

- ▶ Test your hypothesis with additional examples in the console?

```
c(TRUE, TRUE) & c(TRUE, FALSE)
```

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

```
c(TRUE, TRUE) && c(TRUE, FALSE)
```

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

- ▶ Which plays nicely with `ifelse()`?
- ▶ Which plays nicely with `if()`?

What's the difference between `|` and `||`?

Similarly OR has a vectorized `|` and singleton `||` version.

- ▶ generally, you can get by with `|` and `&!`
- ▶ when you are working on code for general use (like writing a package) there will be times when you want to ensure

Exercise

We want to make a new column called `famous_storm` that is 1 for “Katrina” and “Rita” and 0 otherwise.

This code fails.

```
# bad code :(
storms %>%
  mutate(famous_storm =
    ifelse(name == "Katrina" | "Rita", 1, 0))

storms %>%
  mutate(famous_storm =
    ifelse(name == "Katrina" | name == "Rita", 1, 0))

storms %>%
  mutate(famous_storm =
    ifelse(name %in% c("Katrina", "Rita", "Amy", "Bo"))
```

Example: Creating a simulation dataset

You want to understand the impact of discrimination on gifted education.

```
discrimination_simulation <-  
  tibble(group = rep(c(1, 2), 5000),  
         prob_tested = runif(10000),  
         iq = rnorm(10000))
```

- ▶ Students in group 1 get tested with probability 60 percent
- ▶ Students in group 2 get tested with probability 10 percent
- ▶ Students get gifted education if $iq > 1$ and they're tested

Key points: control flow with `if` and contingent column creation with `ifelse`

- ▶ Use `ifelse()` with `mutate()` to create new columns contingently.
 - ▶ `ifelse()` is vectorized so can operate on a logical vector to produce new results
- ▶ Understand how logical operators (i.e. `!`, `|`, `&`) work together with `ifelse` and conditional operators.
- ▶ Use `if()` (and `else`) to control whether an action is completed outside of a data context.

We also introduced `Rmds` and saw how to knit the `Rmd` to `html` or `pdf`.

Up next: prepare lab 3 for tomorrow. **Friday** watch video for class 4 on using `group_by` to do grouped analysis.

Key points: `if()` versus `ifelse()`

	<code>if() / else()</code>
Used to conditionally evaluate code	yes
Vectorized?	no, only uses first element
Handles NA	no, error missing value where T
baseR	yes

- *Takeaway:* When we are focusing on data analysis use `ifelse()` (or `if_else()`).

¹there's a tidyverse `if_else()` that works slightly differently

Quick hitters

Class 2 basics

- ▶ Q Why did I get double when your code shows the type is an int?

```
typeof(seq(1, 12, 1))
```

```
## [1] "double"
```

Quick hitters

Class 2 basics

- ▶ Q Why did I get double when your code shows the type is an int?
- ▶ A In base R, data types are not always predictable.

```
typeof(seq(1, 12))
```

```
## [1] "integer"
```

```
typeof(seq(1, 12, 1))
```

```
## [1] "double"
```

Tidyverse functions tend to be more careful to avoid this sort of behavior.

Horoscope game:

Make a game where you ask the user to enter their birth month and you tell them their fortune.

- ▶ Give people born in December, January or February a “cold” fortune
- ▶ Give people born in June through September a “warm” fortune
- ▶ Give people born in November a great fortune
- ▶ Give everyone else an okay fortune

e.g. `birth_month <- 2` the code should return something like I see penguins in your forecast.

Class 4: Grouped analysis

Today's class

- ▶ Review if statements and conditional expressions
- ▶ Review some base R
- ▶ Take questions about grouped analysis

For the curious: "`[\\r]?\\n`" is a “regular expression” which tells `separate()` to look for "`\\n`" or "`\\r\\n`" and separate the data wherever it finds those strings within other strings.

conditional expressions and missing data

- ▶ `NA | TRUE` returns `TRUE`. Why does `FALSE | NA` return `NA`?

Hint: Fill out this table:

x	y	x OR y
FALSE	FALSE	
FALSE	TRUE	
TRUE	FALSE	
TRUE	TRUE	

conditional expressions and missing data

- ▶ `NA | TRUE` returns `TRUE`. Why does `FALSE | NA` return `NA`?

x	y	x OR y
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

conditional expressions and missing data

- ▶ `NA | TRUE` returns `TRUE`. Why does `FALSE | NA` return `NA`?

x	y	x OR y
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

conditional expressions and missing data

- ▶ `NA | TRUE` returns `TRUE`. Why does `FALSE | NA` return `NA`?

`OR` returns `TRUE` if **any** value is `TRUE`. Thus, when we have `NA | TRUE`, then we logically know that the result is `TRUE`!

`FALSE | NA` depends on the missing value, so R cannot evaluate the expression.

conditional expressions and missing data

- ▶ TRUE & NA returns NA. Why does FALSE & NA return FALSE?

x	y	x AND y
FALSE	FALSE	
FALSE	TRUE	
TRUE	FALSE	
TRUE	TRUE	

conditional expressions and missing data

- ▶ TRUE & NA returns NA. Why does FALSE & NA return FALSE?

x	y	x AND y
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

AND requires **all** TRUE so R knows FALSE & NA is FALSE!

Quick hitter

Q: What's the difference between `ifelse()` and `if_else()`

A: `tidyverse::if_else()` mimics `base::ifelse()` with two major differences

Qh: What's the difference between `ifelse()` and `if_else()`

1. `tidyverse::if_else()` has a built-in way to replace missing data.

```
ex <- c(1, NA, 0, NA, 1)
if_else(ex == 1, "Yes", "No", missing = "Eh")
```

```
## [1] "Yes" "Eh"  "No"  "Eh"  "Yes"
```

Qh: What's the difference between `ifelse()` and `if_else()`

The same behavior requires nesting with `base::ifelse()`

```
ifelse(is.na(ex),  
      "Eh",  
      ifelse(ex == 1, "Yes", "No"))
```

```
## [1] "Yes" "Eh"  "No"  "Eh"  "Yes"
```


Qh: What's the difference between `ifelse()` and `if_else()`

2. `tidyverse::if_else()` checks for type matching

```
if_else(ex == 1, 1, "No")
```

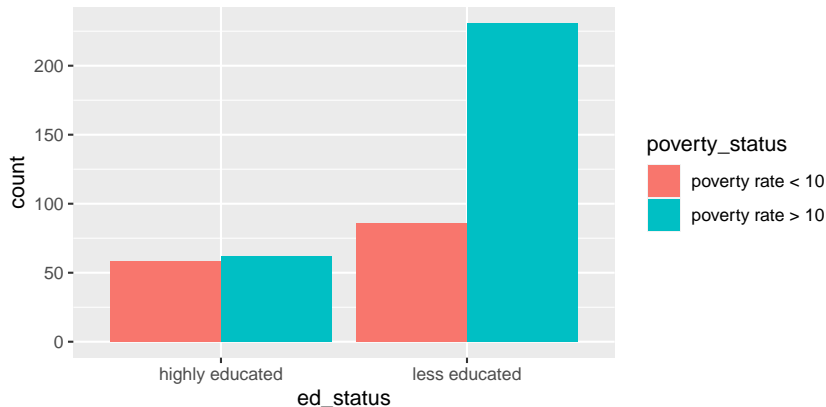
Error: false must be a double vector, not a character vector.

Practice

1. Use `mutate()` and `ifelse()` with `midwest` data to create binary variables
 - ▶ `ed_status` = a city is “highly educated” if over 20 percent of residents have college education (`percollege`)
 - ▶ `poverty_status` = distinguish between cities with poverty rates above and below 10 percent.
2. Assign your intermediate data set to a name.
3. Calculate the proportion “highly educated” and proportion “poverty rate < 10” for counties in Ohio.
4. Calculate the proportion “highly educated” and proportion “poverty rate < 10” for counties in Illinois.

Expected output

```
midwest_binarized %>%  
  ggplot(aes(x = ed_status, fill = poverty_status)) +  
  geom_bar(position = "dodge")
```



Practice

- ▶ Calculate the proportion “highly educated” and proportion “poverty rate < 10” for counties in Ohio.

```
## # A tibble: 1 x 3
##   state prop_highly_educated prop_low_poverty
##   <chr>                <int>          <int>
## 1 OH                      18             29
```

- ▶ Calculate the proportion “highly educated” and proportion “poverty rate < 10” for counties in IL.

```
## # A tibble: 1 x 3
##   state prop_highly_educated prop_low_poverty
##   <chr>                <int>          <int>
## 1 IL                      28             27
```

Solutions

```
midwest_binarized <-  
midwest %>%  
  mutate(poverty_status = ifelse(percbelowpoverty < 10, "poor", "not poor"),  
         ed_status = ifelse(percollege > 20, "highly educated", "not highly educated"))  
  
midwest_binarized %>%  
  filter(state == "IL") %>%  
  summarize(state = first(state),  
            prop_highly_educated = sum(ed_status == "highly educated"),  
            prop_low_poverty = sum(percbelowpoverty < 10),  
            )
```

Is there a better way? What if I want the information for all the states?

Solutions

Use `group_by()` and `summarize()`!

```
midwest_binarized %>%  
  group_by(state) %>%  
  summarize(  
    prop_highly_educated = sum(ed_status == "highly  
    prop_low_poverty = sum(percbelowpoverty < 10)  
  )
```

```
## # A tibble: 5 x 3  
##   state prop_highly_educated prop_low_poverty  
##   <chr>           <int>           <int>  
## 1 IL             28             27  
## 2 IN             18             46  
## 3 MI             27             16  
## 4 OH             18             29  
## 5 WI             29             26
```

Quick hitter: When do we need `ungroup()`?

Recall, `group_by()` adds information about groups “silently” without changing the data

```
midwest_grouped <-  
midwest_binarized %>%  
  group_by(state, ed_status, poverty_status) %>%  
  select(poptotal)
```

Adding missing grouping variables: 'state', 'ed_status'

```
midwest_grouped %>%  glimpse()
```

```
## Rows: 437
```

```
## Columns: 4
```

```
## Groups: state, ed_status, poverty_status [20]
```

```
## $ state      <chr> "IL", "IL", "IL", "IL", "IL", "IL"
```

```
## $ ed_status  <chr> "less educated", "less educated"
```

```
## $ poverty_status <chr> "poverty rate > 10", "poverty rat
```

```
## $ poptotal   <int> 66090, 10626, 14991, 30806, 5836
```

Quick hitter: When do we need `ungroup()`?

If we want to do analysis where groups get in the way, we need `ungroup()`.

```
midwest_grouped %>%  
  ungroup() %>%  
  glimpse()
```

```
## Rows: 437  
## Columns: 4  
## $ state      <chr> "IL", "IL", "IL", "IL", "IL", "IL"  
## $ ed_status  <chr> "less educated", "less educated",  
## $ poverty_status <chr> "poverty rate > 10", "poverty rat  
## $ poptotal   <int> 66090, 10626, 14991, 30806, 5836,
```


Quick hitter: When do we need `ungroup()`?

If we want to do analysis where groups get in the way, we need `ungroup()`.

```
midwest_grouped %>%  
  summarize(biggest_county = max(poptotal))
```

```
## 'summarise()' has grouped output by 'state', 'ed_status'
```

```
## # A tibble: 20 x 4
```

```
## # Groups:   state, ed_status [10]
```

##	state	ed_status	poverty_status	biggest_county
##	<chr>	<chr>	<chr>	<int>
##	1 IL	highly educated	poverty rate < 10	78166
##	2 IL	highly educated	poverty rate > 10	510506
##	3 IL	less educated	poverty rate < 10	4805
##	4 IL	less educated	poverty rate > 10	24923
##	5 IN	highly educated	poverty rate < 10	30083
##	6 IN	highly educated	poverty rate > 10	79715
##	7 IN	less educated	poverty rate < 10	15619

Quick hitter: When do we need `ungroup()`?

If we want to do analysis where groups get in the way, we need `ungroup()`.

```
midwest_grouped %>%  
  ungroup() %>%  
  summarize(biggest_county = max(poptotal))
```

```
## # A tibble: 1 x 1  
##   biggest_county  
##           <int>  
## 1           5105067
```

Sure, but why would I group that data in the first place?

Perhaps you did a `group_by()` + `summarize()` with multiple groups.

- ▶ The default is to strip the “last” grouping variable

```
midwest_grouped %>%  
  summarize(avg_pop = mean(poptotal)) %>%  
  glimpse()
```

```
## 'summarise()' has grouped output by 'state', 'ed_status'
```

```
## Rows: 20
```

```
## Columns: 4
```

```
## Groups: state, ed_status [10]
```

```
## $ state      <chr> "IL", "IL", "IL", "IL", "IN", "IN"
```

```
## $ ed_status  <chr> "highly educated", "highly educated"
```

```
## $ poverty_status <chr> "poverty rate < 10", "poverty rate < 10"
```

```
## $ avg_pop    <dbl> 234318.64, 390871.41, 25303.62, 390871.41, 25303.62, 390871.41, 25303.62, 390871.41, 25303.62, 390871.41
```

BaseR: what's the deal with subsetting?

How many ways can you pull out the even numbers from `vec`?

```
vec <- 1:10
```

BaseR: what's the deal with subsetting?

How many ways can you pull out the even numbers from `vec`?

```
vec <- 1:10  
vec[c(2, 4, 6, 8, 10)]
```

```
## [1]  2  4  6  8 10
```

```
vec[rep(c(FALSE, TRUE), 5)]
```

```
## [1]  2  4  6  8 10
```

```
vec[seq(2, 10, 2)]
```

```
## [1]  2  4  6  8 10
```

```
vec[vec %% 2 == 0]
```

```
## [1]  2  4  6  8 10
```

Notice `filter()` works like the final option!

BaseR: what's the deal with subsetting?

Now what if vec is a column in a tibble?

```
data <- tibble(vec_col = 1:10, random_col = "A")
```

BaseR: what's the deal with subsetting?

Now what if vec is a column in a tibble?

```
data <- tibble(vec_col = 1:10, random_col = "A")

# equivalent to pull() (with filter)
data$vec_col[data$vec_col %% 2 == 0] # nicest way
data[["vec_col"]][c(2, 4, 6, 8, 10)]
data[rep(c(FALSE, TRUE), 5), ]["vec_col"]

# equivalent to select() (with filter)
data[rep(c(FALSE, TRUE), 5), "vec_col" ]
data[seq(2, 10, 2), "vec_col"]

data %>%
  filter(vec_col %% 2 == 0) %>%
  select(vec_col)
```

BaseR: what's the deal with subsetting?

Using `[` with vectors is natural.

- ▶ you can extract based on
 - ▶ indices
 - ▶ a vector of booleans (of equal length)
 - ▶ and names (if there are names)

When working with data, use tidyverse verbs to extract data.

- ▶ easier to remember words
- ▶ help avoid syntax errors and confusion between `[`, `[[` and `$`

When googling for help add “in R tidyverse” to your query. If you start doing numerical computing, look for a review of `[` and `[[`.

Wait did you say names for a vector?

Yep, we can have a named vector.

```
x <- c("a" = 1, "b" = 2)  
x["a"]
```

```
## a
```

```
## 1
```

Named vectors are not, Lists

Notice this is distinct from a list.

- ▶ The key difference is that lists can accept different data types as entries.

```
c("a" = c(1, 2), "b" = 2)
```

```
## a1 a2 b  
## 1 2 2
```

```
list("a" = c(1, 2), "b" = 2)
```

```
## $a  
## [1] 1 2  
##  
## $b  
## [1] 2
```

Tibbles are built on lists

- ▶ but tibbles enforce all entries to be the same length vectors.

```
my_tib <- tibble(a = c(1, 2),  
                 b = "A")
```

```
my_list <- list(a = c(1, 2),  
               b = c("A", "A"))
```

e.g.

```
my_tib[["b"]]
```

```
## [1] "A" "A"
```

```
my_list[["b"]]
```

```
## [1] "A" "A"
```

Key points: Grouped analysis with `group_by()`

- ▶ *groups* are a set of rows that belong together.
 - ▶ `group_by()` adds information about groups “silently” without changing the data
- ▶ Use `group_by()` with `summarize()` to create summary tables at group-level.
 - ▶ Use with functions that *reduce* data from a vector to a single value per group.
 - ▶ Expected output: a table with one row per group and one column per summary statistic (and one column per grouping column.)

Key points: Grouped analysis with `group_by()`

- ▶ we can also use `group_by()` to do grouped analysis with `mutate()`
 - ▶ you can use a “window function” like `lag()`
 - ▶ or add summary statistics to your main data set for further analysis
- ▶ `group_by()` also can impact `arrange()` and `filter()`

Up next: Grouping() watch video for class 5 on using loops.

Discussion

How do we make the data for this graph?

- ▶ What “groups” are required for the visualization?

Exercise

A student came during office hours and asked why `mean(percentile=="p90p100")` doesn't calculate the average wealth shares (value) for the top 10 percentile group.

```
wid_data %>%  
  filter(country %in% c("China", "India", "USA")) %>%  
  group_by(country) %>%  
  summarize(p90_mean = mean(percentile == "p90p100",  
                             na.rm = TRUE))
```

Class 5: Loops

Feedback

Thank you for filling out the feedback form (we have 30 so far!)

- ▶ “Would be great if you could post solutions to the labs at the end of each week.” **These are on the course website**
- ▶ “Please share qa code”. **These are (newly) on the course website**

Today's class

- ▶ `case_when()` question
- ▶ Review a few grouped analysis concepts
- ▶ Practice using loops

Quicker hitter

Q Why do you end `case_when()` with `TRUE ~ ...?`

```
x <- 1:35
case_when(
  x %% 35 == 0 ~ "fizz buzz",
  x %% 5 == 0 ~ "fizz",
  x %% 7 == 0 ~ "buzz",
  TRUE ~ as.character(x)
)
```

##	[1]	"1"	"2"	"3"	"4"	"fizz"
##	[7]	"buzz"	"8"	"9"	"fizz"	"11"
##	[13]	"13"	"buzz"	"fizz"	"16"	"17"
##	[19]	"19"	"fizz"	"buzz"	"22"	"23"
##	[25]	"fizz"	"26"	"27"	"buzz"	"29"
##	[31]	"31"	"32"	"33"	"34"	"fizz"

Quicker hitter

A If no cases match, NA is returned.

```
x <- 1:35  
case_when(  
  x %% 35 == 0 ~ "fizz buzz",  
  x %% 5 == 0 ~ "fizz",  
  x %% 7 == 0 ~ "buzz"  
)
```

##	[1]	NA	NA	NA	NA	"fizz buzz"
##	[7]	"buzz"	NA	NA	"fizz"	NA
##	[13]	NA	"buzz"	"fizz"	NA	NA
##	[19]	NA	"fizz"	"buzz"	NA	NA
##	[25]	"fizz"	NA	NA	"buzz"	NA
##	[31]	NA	NA	NA	NA	"fizz buzz"

Q: What does `group_by()` do?

The following command runs without error, but I'm not seeing any change in `grouped_data` compared to `traffic_data`.

- ▶ Both still have 4478 jobs of 9 variables.

Am I missing something?

```
grouped_data <-  
  traffic_data %>%  
    group_by(Race, Gender)
```

Two questions on grouped mutates.

Recall:

- ▶ `group_by() + summarize()` **collapses** the data to a row with summary values per group.
- ▶ `group_by() + mutate()` maintains the data set, but adds columns that may depend on group membership
 - ▶ e.g. `lag()`, `row_number()`

How does lag() work?

Find the “previous” (lag()) or “next” (lead()) values in a vector.

```
lag(c(1, 2, 3))
```

```
## [1] NA  1  2
```

How does lag() work?

Useful for comparing values behind of or ahead of the current values.

```
x <- 1:5  
tibble(behind = lag(x), x, ahead = lead(x))
```

```
## # A tibble: 5 x 3  
##   behind      x ahead  
##   <int> <int> <int>  
## 1     NA      1      2  
## 2      1      2      3  
## 3      2      3      4  
## 4      3      4      5  
## 5      4      5     NA
```


How does lag() work with grouped data?

► Why is this wrong?

```
x <- c(2, 3, 30, 20)
group <- c("A", "A", "B", "B")
time <- c(1, 2, 2, 1)
tibble(group, time, x) %>%
  mutate(previous = lag(x))
```

```
## # A tibble: 4 x 4
##   group  time      x previous
##   <chr> <dbl> <dbl>    <dbl>
## 1 A         1      2      NA
## 2 A         2      3       2
## 3 B         2     30       3
## 4 B         1     20      30
```

How does lag() work with grouped data?

You need to have the data in the correct

- ▶ order
- ▶ group

```
tibble(time, group, x) %>%  
  group_by(group) %>%  
  arrange(time) %>%  
  mutate(previous = lag(x))
```

```
## # A tibble: 4 x 4  
## # Groups:   group [2]  
##   time group      x previous  
##   <dbl> <chr> <dbl>     <dbl>  
## 1     1 1 A      2      NA  
## 2     1 1 B     20      NA  
## 3     2 2 A      3       2  
## 4     2 2 B     30     20
```

Practice

Use `txhousing`, a tidyverse dataset.

- ▶ Calculate the monthly change in number of sales for each city.
 - ▶ hint: use `lag()` in a grouped mutate!
- ▶ Calculate the annual change in total sales for each city.

Q.

Explain the rank() function(s). How is it different than order() and sort()?

```
x <- c(10, 0, 1, 2, 3, NA)
```

```
rank(x)
```

```
## [1] 5 1 2 3 4 6
```

```
order(x)
```

```
## [1] 2 3 4 5 1 6
```

```
sort(x)
```

```
## [1] 0 1 2 3 10
```

```
row_number(x)
```

```
## [1] 5 1 2 3 4 NA
```

A. Sort puts the data in order.

```
x
```

```
## [1] 10  0  1  2  3 NA
```

```
(the_order <- order(x))
```

```
## [1] 2 3 4 5 1 6
```

```
x[the_order]
```

```
## [1]  0  1  2  3 10 NA
```

```
sort(x, na.last = TRUE)
```

```
## [1]  0  1  2  3 10 NA
```

A. Rank ranks the data in place.

```
x
```

```
## [1] 10  0  1  2  3 NA
```

```
(the_rank <- rank(x))
```

```
## [1] 5 1 2 3 4 6
```

```
the_order
```

```
## [1] 2 3 4 5 1 6
```

tidyverse provides 6 ranking functions

```
# the help shows all 6!  
?row_number()
```

Explain how the ranking functions interact with a `group_by()` + `mutate()`.

For loops and iteration

- ▶ Iteration is useful when we are repeatedly calling the same block of code or function while changing one (or two) inputs.
- ▶ but if you can, use vectorized operations!

For loops

Compare and contrast the two code blocks

```
# block a
x <- -10:10
y <- integer(length = length(x))

for (i in seq_along(x)) {
  y[[i]] <- x[[i]]^2 + x[[i]] + 1
}
```

and

```
# block b
x <- -10:10
y <- c()

for (item in x) {
  y <- c(y, item^2 + item + 1)
}
```

loops

	block a	block b
iterates over	indexes	items
preallocates space?	yes	no
vectorized?	no	no

Vectorized? No.

Can you vectorize the code?

```
# block b
x <- -10:10
y <- c()

for (item in x) {
  y <- c(y, item^2 + item + 1)
}
```

Vectorized? yes.

```
x <- -10:10  
y <- x^2 + x + 1
```

Is preallocation a big deal?

I encapsulated our code into functions for testing.

```
preallocate_loop <- function(x){  
  y <- integer(length = length(x))  
  for (i in seq_along(x)) {  
    y[[i]] <- x[[i]]^2 + x[[i]] + 1  
  }  
  y  
}
```

```
build_loop <- function(x){  
  y <- c()  
  for (item in x) {  
    y <- c(y, item^2 + item + 1)  
  }  
  y  
}
```

For tiny data sets, it's not a huge deal.

```
x <- -10:10  
bench::mark(preallocate_loop(x), build_loop(x)) %>%  
  select(expression, min, median, mem_alloc)
```

```
## # A tibble: 2 x 4  
##   expression          min    median mem_alloc  
##   <bch:expr>      <bch:tm> <bch:tm> <bch:byt>  
## 1 preallocate_loop(x)  5.33us   7.71us  41.6KB  
## 2 build_loop(x)       7.54us  21.98us  29.5KB
```

For big data sets, it increasingly becomes a big deal.

```
# warning this test takes a while!  
x <- -1e4:1e4  
bench::mark(preallocate_loop(x), build_loop(x)) %>%  
  select(expression, min, median, mem_alloc)
```

```
## Warning: Some expressions had a GC in every iteration; s
```

```
## # A tibble: 2 x 4
```

##	expression	min	median	mem_alloc
##	<bch:expr>	<bch:tm>	<bch:tm>	<bch:byt>
## 1	preallocate_loop(x)	2.76ms	3.2ms	234.48KB
## 2	build_loop(x)	2.52s	2.52s	1.49GB

vectorized option

```
vectorized_code <- function(x) {  
  x^2 + x + 1  
}
```


vectorized option performs favorably

```
x <- -1e6:1e6
bench::mark(preallocate_loop(x),
             vectorized_code(x)) %>%
  select(expression, min, median, mem_alloc)
```

```
## Warning: Some expressions had a GC in every iteration; s
```

```
## # A tibble: 2 x 4
```

##	expression	min	median	mem_alloc
##	<bch:expr>	<bch:tm>	<bch:tm>	<bch:byt>
## 1	preallocate_loop(x)	669.9ms	669.9ms	22.9MB
## 2	vectorized_code(x)	20.1ms	56.2ms	22.9MB

mea culpa: the case for []

Last class, I encouraged you to not use [] for subsetting / indexing.

- ▶ But we need these when indexing into lists!

lists

lists are a **data structure** that can store different data types in the same object.

```
list(c(12, 1), 4, "f")
```

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

```
## [1] 12  1
```

```
##
```

```
## [[2]]
```

```
## [1] 4
```

```
##
```

```
## [[3]]
```

```
## [1] "f"
```

lists

- ▶ tibbles/data.frames are built on lists

```
(x <- list('int' = c(12L, 1L),  
          'dbl' = c(4.02, pi),  
          'char' = c("f", "f")))
```

```
## $int  
## [1] 12  1  
##  
## $dbl  
## [1] 4.020000 3.141593  
##  
## $char  
## [1] "f" "f"
```

Why [[]

`[]` pulls out the whatever is in the `i`th position of the list while `[` pulls out a subset of the list.

```
x[1]
```

```
## $int  
## [1] 12  1
```

```
x[[1]]
```

```
## [1] 12  1
```

Why []

When setting the *i*th stuff in a list we use [] so we can place any given object into the list.

```
# create an empty list  
output <- vector("list", 2)  
(output[1] <- c(1, 2))
```

```
## Warning in output[1] <- c(1, 2): number of items to replace is  
## replacement length
```

```
## [1] 1 2
```

```
(output[[1]] <- c(1, 2))
```

```
## [1] 1 2
```

Key points: for-loops

- ▶ If you can't vectorize, for loops work for iteration
 - ▶ Clearly define what you will iterate over (values or indices)
 - ▶ Preallocate space for your output (if you can)
 - ▶ The body of the for-loop has parametrized code based on thing your iterating over
 - ▶ Debug as you code by testing your understanding of what the for-loop should be doing (e.g. using `print()`)

Learn more in chapter 21 of r for data science:

<https://r4ds.had.co.nz/iteration.html>

Class 6: Functions

Today's class

- ▶ loop questions
- ▶ functions writing concepts
- ▶ primer for lab 6

Reminder: Please turn in your final projects on gradescope.

- ▶ Suggested deadline Sept 17 (tomorrow)
- ▶ Final deadline Sept 24.

Q Can I pipe a for-loop into ggplot()?

```
# psuedo-code  
for (i in seq_along(x)) {  
  results[[i]] <- some_function(x[[i]])  
} %>%  
  ggplot(aes(x = x, y = y)) + ...
```

Reframe Q What is the output of a for-loop? Can it be a tibble?

- ▶ Each iteration there's an “intermediate output” which is the final line of code in the loop. (like a function).

```
x <- c(0, 1, 2)
result <- integer(length(x))

output <- for (i in seq_along(x)) {
  result[[i]] <- x[[i]]^2
  "last line inside loop"
} %>% print()
```

```
## [1] "last line inside loop"
## [1] "last line inside loop"
## [1] "last line inside loop"
```

- ▶ The “final output” is NULL

```
print(output)
```

```
## NULL
```

Q Can I pipe a for-loop into ggplot()?

If the last line inside the loop is a tibble, then yes (kind of).

- ▶ R doesn't display the output
- ▶ But we can assign it to a name

```
my_plot <- vector("list", 3)

for ( i in 1:3) {
  # do stuff
  # ...
  # end with
  txhousing
} %>%
  ggplot(aes(x = sales, y = volume)) +
  geom_point() -> my_plot[[i]]

my_plot[[3]]
```

- ▶ *Don't do it!* More natural to call ggplot w/i the loop or after collecting a dataset.

Q How to preallocate space for a tibble?

```
estimates <- tibble(n = integer(), sample_mean = double())

for (n in sample_sizes) {
  sample_mean <- mean(rnorm(n, mean = 0, sd = 5))
  estimates <- bind_rows(estimates,
                        c(n = n, sample_mean = sample_mean))
}
estimates
```

Named vectors behave like “rows” with a single data type

```
row_1 <- c(a = 1, b = "dog")  
row_2 <- c(a = 2, b = "cat")  
  
bind_rows(row_1, row_2)
```

```
## # A tibble: 2 x 2  
##   a      b  
##   <chr> <chr>  
## 1 1     dog  
## 2 2     cat
```

Named lists behave like “rows” with a various data type

```
row_1 <- list(a = 1, b = "dog")  
row_2 <- list(a = 2, b = "cat")  
  
bind_rows(row_1, row_2)
```

```
## # A tibble: 2 x 2  
##       a b  
##   <dbl> <chr>  
## 1     1 dog  
## 2     2 cat
```

We could also create one row tibbles

```
tibble_1 <- tibble(a = 1, b = "dog")  
tibble_2 <- tibble(a = 2, b = "cat")  
  
bind_rows(tibble_1, tibble_2)
```

```
## # A tibble: 2 x 2  
##       a b  
##   <dbl> <chr>  
## 1     1 dog  
## 2     2 cat
```


“rows” can be bound together with `bind_rows()`

```
row_3 <- list(a = 3, b = "dog")  
data <- bind_rows(row_1, row_2)  
(data <- bind_rows(data, row_3))
```

```
## # A tibble: 3 x 2  
##       a b  
##   <dbl> <chr>  
## 1     1 dog  
## 2     2 cat  
## 3     3 dog
```

“rows” can be bound together with `bind_rows()`

```
our_rows <- list(row_1, row_2, row_3)
(data <- bind_rows(our_rows))
```

```
## # A tibble: 3 x 2
##       a b
##   <dbl> <chr>
## 1     1 dog
## 2     2 cat
## 3     3 dog
```

Q How to preallocate space for a tibble?

The previous slide suggest we preallocate a list with one slot for each row.

```
sample_sizes <- seq(2, 10, 2) # c(2, 4, 6, 8, 10)
estimates <- vector("list", length(sample_sizes))
```

Q How to preallocate space for a tibble?

Then in the body of the loop we assign a “row” to the *i*th entry of `estimates`

```
for (i in seq_along(sample_sizes)) {  
  n <- sample_sizes[[i]]  
  sample_mean <- mean(rnorm(n, mean = 0, sd = 5))  
  estimates[[i]] <- c(n = n, sample_mean = sample_mean)  
}  
  
# estimates is a list of vectors until bind_rows(.)!  
estimates <- bind_rows(estimates)
```

notice this is the **only time** in coding lab we advocate using lists.

- ▶ use vectorized code / use tidyverse code

another loop example

Compute the mean of every column in `midwest`

another loop example

Compute the mean of every column in midwest.

- ▶ use `across()` (we won't cover this in detail)

```
midwest %>%  
  summarize(across(.fns = mean))
```

```
## # A tibble: 1 x 28  
##   PID county state   area poptotal popdensity popwhite  
##   <dbl> <dbl> <dbl>   <dbl>     <dbl>       <dbl>    <dbl>  
## 1 1437.     NA    NA 0.0332   96130.       3098.    81840  
## # ... with 19 more variables: popasian <dbl>, popother <dbl>,  
## #   percblack <dbl>, percamerindan <dbl>, percasian <dbl>,  
## #   popadults <dbl>, perchsd <dbl>, percollege <dbl>, pe  
## #   poppovertyknown <dbl>, percpovertyknown <dbl>, percb  
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>,  
## #   percelderlypoverty <dbl>, inmetro <dbl>, category <dbl>
```

another loop example

Compute the mean of every column in `midwest`

- ▶ What should we loop over?
- ▶ How do we calculate what we're looking for?

another loop example

Compute the mean of every column in `midwest`

```
mean(midwest[["area"]])
```

```
## [1] 0.03316934
```

```
# or
```

```
midwest %>%  
  pull(area) %>%  
  mean()
```

```
## [1] 0.03316934
```

```
# or
```

```
midwest %>%  
  summarise(area = mean(area))
```

```
## # A tibble: 1 x 1
```

```
##   area
```

```
##   <dbl>
```


another loop example

```
# preallocate output
col_names <- names(midwest)
midwest_means <- vector("double", ncol(midwest))

# determine what to loop over
for (i in seq_along(col_names)) {
  # body
  col_name <- col_names[[i]]
  midwest_means[[i]] <- mean(midwest[[col_name]])
}

tibble(names = col_names, means = midwest_means)
```

```
## # A tibble: 28 x 2
##   names          means
##   <chr>         <dbl>
## 1 PID          1437.
## 2 county       NA
```

function writing concept

- ▶ A function maps an input to an output.
“A user logs into our website. You respond with a greeting.”

fwc: input-output

Write the function!

input: name output: greeting

Recall

```
#psuedo code for a function  
function_name <- function(inputs) {  
  body  
}
```

fwc: input-output

```
greet_user <- function(name) {  
  paste0("Hello ", name, "!")  
}
```

```
greet_user("Qiwei")
```

```
## [1] "Hello Qiwei!"
```

fwc: when to write a function?

- ▶ If you repeat very similar code more than twice
- ▶ You can easily parameterize the similar code

```
txhousing %>%  
  filter(city == "Austin") %>%  
  mutate(date = year + month / 12) %>%  
  ggplot(aes(x = date, y = sales)) +  
  geom_line() +  
  labs(title = "Austin's sales volume")
```

fwc: don't repeat yourself

```
sales_trends <- function(city_name){  
  txhousing %>%  
    filter(city == city_name) %>%  
    mutate(date = year + month / 12) %>%  
    ggplot(aes(x = date, y = sales)) +  
    geom_line() +  
    labs(title = str_c(city_name, "'s sales volume"))  
}
```

fwc: don't repeat yourself

```
sales_trends("Austin")  
sales_trends("Houston")
```

fwc: don't repeat yourself (Advanced)

Extra slick: can we parameterize the y variable.

fwc: test your code as you write it

```
trends <- function(city_name, var){  
  txhousing %>%  
    filter(city == city_name) %>%  
    mutate(date = year + month / 12) %>%  
    ggplot(aes(x = date, y = var)) +  
    geom_line() +  
    labs(title = str_c(city_name, "'s ", var))  
}  
  
trends("Austin", sales, "sales volume")
```

fwc: don't repeat yourself (Advanced)

Extra slick: can we parameterize the y variable.

```
trends <- function(city_name, var, var_description){  
  txhousing %>%  
    filter(city == city_name) %>%  
    mutate(date = year + month / 12) %>%  
    ggplot(aes(x = date, y = {{ var }})) +  
    geom_line() +  
    labs(title = str_c(city_name,  
                        "'s ",  
                        var_description))  
}  
  
trends("Austin", sales, "sales volume")
```

{{ }}?

`{{ }}` “forwards” the name `var` from your global environment (where it’s undefined) to the tibble’s environment (where it’s a column name).

- ▶ The ideas here are somewhat advanced for coding camp

fwc: break big ideas into small ideas

- ▶ **problem** encapsulating 100s of lines of data analysis in a single function is a recipe for disaster.
- ▶ **solution** break the 100s of lines of analysis into coherent chunks (think high level about input-output!)

This is the heart of lab 6.

Background for the lab

Concepts: r/q/p/d functions

R has built-in functions for working with distributions.

	example	what it does?
r	<code>rnorm(n)</code>	generates a random sample of size n
p	<code>pnorm(q)</code>	returns CDF value at q
q	<code>qnorm(p)</code>	returns inverse CDF (the quantile) for a given probability
d	<code>dnorm(x)</code>	returns pdf value at x

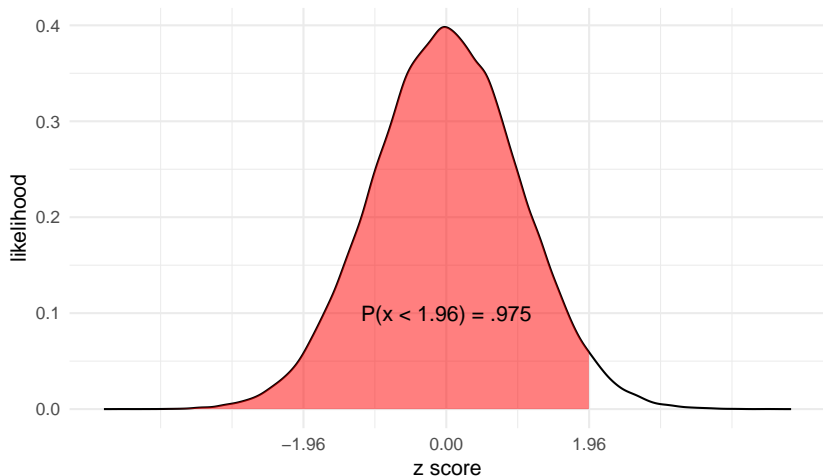
We should already be familiar with `r` functions like `rnorm()` and `runif()`.

- ▶ These concepts will be taught in stats. We don't expect you to learn them here. We can help you reason through the material enough to do the coding.

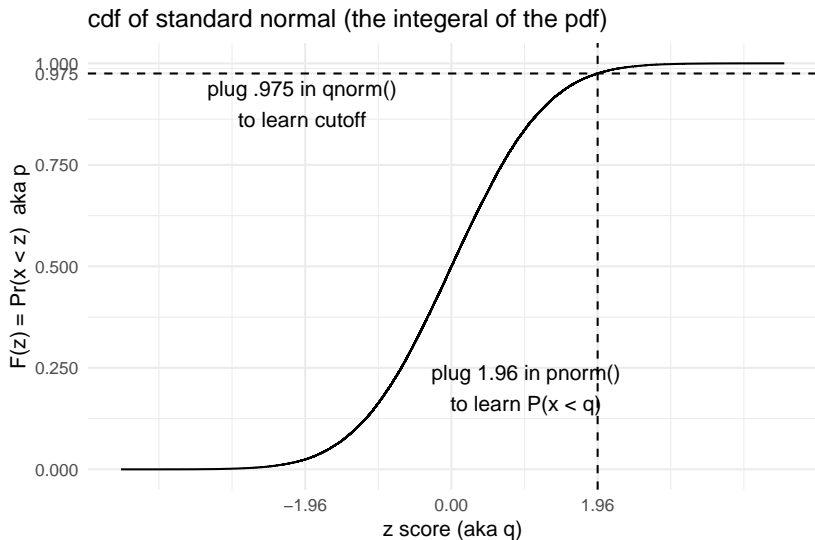
What are p and q?

pdf of standard normal

area under curve is the probability of being less than a cutoff



What are p and q?



What are p and q?

`pnorm` returns the probability we observe a value less than or equal to some value q .

```
pnorm(1.96)
```

```
## [1] 0.9750021
```

```
pnorm(0)
```

```
## [1] 0.5
```

`qnorm` returns the inverse of `pnorm`. Plug in the probability and get the cutoff.

```
qnorm(.975)
```

```
## [1] 1.959964
```

```
qnorm(.5)
```

```
## [1] 0
```

Monte Carlo experiments

Monte Carlo is a world gambling hub.

- ▶ Gamblers know that roulette wheels are not made perfectly.
- ▶ If you watch the wheel long enough and take notes you can figure out the empirical probability

Monte Carlo experiments

Statisticians use the same idea.

Esp. if we're not sure how to calculate something exactly, but have a model

- ▶ complicated interacting systems
- ▶ integrals or other objects without a closed form solution (or a difficult to compute closed form)

How to do a Monte Carlo Simulation

1. Generate random samples of data using a known process (e.g. `rnorm()`).
2. Make calculations based on the random sample.
3. Aggregate the results.

You will do this in the functions lab!

Thank you

It has been a blast.