Starting with R

Welcome to R

- So cruel to set you up for econometrics and take a detour
- But we're going to spend some time today focusing on getting ourselves set up to use R
- R is a programming language designed for use in statistics
- We will be using it throughout the course

Getting Set Up

- We can install R at R-project.org
- Then we can download and install RStudio at RStudio.com
- Then, R is installed and ready to go! Open up RStudio to get going
- Alternately, you can skip installing anything and use RStudio.cloud which runs entirely in the browser but is a bit slower and requires a (free) account
- **If you installed R a while ago, please reinstall it. I'd like everyone to have 4.1 or newer.**

The RStudio Panes

- Console
- Environment Pane
- Browser Pane
- Source Editor

Console

- Typically bottom-left
- This is where you can type in code and have it run immediately
- Or, when you run code from the Source Editor, it will show up here
- It will also show any output or errors

Console Example

- Let's copy/paste some code in there to run

```
#Generate 500 heads and tails
data ← sample(c("Heads","Tails"),500,replace=TRUE)
#Calculate the proportion of heads
mean(data=="Heads")
#This line should give an error - it didn't work!
data ← sample(c("Heads","Tails"),500,replace=BLUE)
#This line should give a warning
#It did SOMETHING but maybe not what you want
mean(data)
#This line won't give an error or a warning
#But it's not what we want!
mean(data=="heads")
```

What We Get Back

- We can see the code that we've run
- We can see the output of that code, if any
- We can see any errors or warnings (in red). Remember - errors mean it didn't work. Warnings mean it *maybe* didn't work.
- Just because there's no error or warning doesn't mean it DID work! Always think carefully
- Specific note: Warning: (package name) was built in R version (version number) just means that your R installation isn't fully updated. Usually not a problem, but you can update R at R-project.org to make this go away

Environment Tab

- Environment tab shows us all the objects we have in memory
- For example, we created the `data` object, so we can see that in Environment
- It shows us lots of handy information about that object too
- (we'll get to that later)
- You can erase everything with that little broom button (technically this does `rm(list=ls())`)

Browser Pane

- Bottom-right
- Lots of handy stuff here!
- Mostly, the *outcome* of what you do will be seen here
- Plots you make will show up here
- Some functions create tables or output that show up in Viewer
- Packages tab - avoid for loading, but the update button is nice!

Files Tab

- Basic file browser
- Handy for opening up files
- Can also help you set the working directory:
  - Go to folder
  - In menu bar, Session
  - Set Working Directory
  - To Files Pane Location

Help Tab

- This is where help files appear when you ask for them
- You can use the search bar here, or `help()` in the console
- Of course, plenty of materials also available for whatever on Google!

Source Pane

- You should be working with code FROM THIS PANE, not the console!
- Why? Replicability!
- Also, COMMENTS! USE THEM! PLEASE! `#` lets you write a comment.
- Switch between tabs like a browser

Running Code from the Source Pane

- Select a chunk of code and hit the "Run" button
- Click on a line of code and do Ctrl/Cmd-Enter to run just that line and advance to the next <- Super handy!
- Going one line at a time lets you check for errors more easily
- Let's try some!

```
data(mtcars)
mean(mtcars$mpg)
mean(mtcars$wt)
372+565
log(exp(1))
2^9
(1+1)^9
```

Autocomplete

- RStudio comes with autocomplete!
- Typing in the Source Pane or the Console, it will try to fill in things for you
  - Command names (shows the syntax of the function too!)
  - Object names from your environment
  - Variable names in your data
- Let's try redoing the code we just did, typing it out

Help

- Autocomplete is one way that RStudio tries to help you out
- The way that R helps you out is with the documentation
- When you start doing anything serious with a computer, like programming, the most important skills are:
  - Knowing to read documentation
  - Knowing to search the internet for help (always!)

help()

- You can get the documentation on most R objects using the `help()` function
- `help(mean)`, for example, will show you:
  - What the function is
  - The "syntax" for the function and the order the arguments go in
  - The available options for the function
  - Other, related functions, like `weighted.mean`
  - Ideally, some examples of proper use
- Not just for functions/commands - some data sets will work too! Try `help(mtcars)`

Packages

- R runs on user-contributed packages that contain functions you can use
- Packages stored on CRAN can be installed with `install.packages('packagename')`
- Once a package is installed you don't need to install it again (except to update it)
- But every time you open R you'll need to load it in again with `library(packagename)` if you want to use its functions
- *Please don't* include package installation in your code itself; this will make you re-install the package every time you run!

```
# If we haven't installed it yet
# install.packages('vtable')
library(vtable)
vtable(iris)
```

RMarkdown

- Go File $\rightarrow$ New File $\rightarrow$ RMarkdown to create a new RMarkdown document
- A text document with some basic layout, for example `hashtags` for sectioning
- Include formatted code with backticks
- Triple backticks for not-inline code chunk. Let's look!

Working in R

- Everything in R is an object (can contain data and code)
- We can only really do three things in R:
  - Create objects with `←` or `=`
  - Send objects through functions to manipulate them
  - Look at objects

Objects

- Let's create a basic object

```
a ← 1
```

- We've taken `1` and stored it inside the `a` object
- Now if we just type `a` by itself, it will show us the `1` we stored inside
- This is a numeric object. We could also have `'strings'` or logicals: `TRUE` or `FALSE` or factors

Objects

- We can manipulate objects

```
a + 1
```

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

- "create a new object that takes `a` (1) and adds 1 to it (`1+1=2`)
- Notice that `a` itself doesn't change until we *reassign it*

```
a
```

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

```
a ← a + 1
a
```

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

Vectors

- A vector is a collection of objects of the same type
- While lots of functions create vectors, we can also make them ourselves with `c()` (concatenate)

```
my_vector ← c(1,8,2,4,3)
```

- We can refer to a certain element of a vector with square brackets `[]` with a single number or a range `start:end`

```
my_vector[3:4]
```

```
## [1] 2 4
```

- Or using another vector of logicals (`TRUE` and `FALSE`) to pick elements (handy when we get to data!)

```
my_vector[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 1 2 4
```

Data Frames

- A `data.frame` is what we'll be working with most of the time.
- It's a collection of vectors of the same length
- We can create them ourselves, e.g. `data.frame(a = c(1,5,3,2), b = c('a','d','b','f'))`, but often we will read in a file or use `data()` to get a data set

```
data(mtcars)
mtcars
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
```

Data Frames

- We can get a vector back out of the data frame with `$` or `[[]]`

```
mtcars$cyl[1:5]
```

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

```
mtcars[['cyl']][1:5]
```

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

Data Frames

- Which we might want to do to send it to a function like `mean` that takes a vector!

```
mean(mtcars$cyl)
```

```
## [1] 6.1875
```

dplyr

- In this course, we'll be working with data largely using the **dplyr** package, which is a part of the **tidyverse**
- **dplyr** is a collection of verbs for manipulating data, all strung together with the pipe `%>%` (which technically **dplyr** just borrows from **magrittr**)
- **dplyr** has lots of functions in it. Today we'll cover just five: `pull()`, `filter()`, `%>%`, `select()`, and `mutate()`.
- For all of these, don't forget that if you want to take the data frame you've changed and *keep* that change, you have to re-assign the object with `←` or `=`!
- If you already know a different way of doing things in R, I **strongly recommend** using the **dplyr** commands, if only for this class, as it will make it much easier to follow along with the material, and it's good to learn to pick up new things (and also in my experience you guys end up making things waaaay harder on yourselves by avoiding the switch)

pull()

- Pull just takes a vector back out of a data set, just like `$` or `[[]]`
- So why use it? It plays well with the pipe (upcoming)
- Also, it's flexible. It takes a variable name, a variable name as a string, or a column number

```
mtcars$cyl
pull(mtcars, cyl)
pull(mtcars, 'cyl')
pull(mtcars, 2)
mtcars %>% pull(cyl)
```

filter()

- `filter()` picks just some of the *rows* of your data
- Important for analyzing a subset of your data!
- Give it a logical statement that's `TRUE` for the rows you want. `==` checks for equality!

```
filter(mtcars, cyl == 4 & am == 1)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

The pipe

- The pipe `%>%` is important for making your code readable, and minimizing balanced-parentheses errors
- It takes whatever is on its left and makes it the first argument of the function on the right
- So whatever object you're working with you take, ship it along to the next function, process, then ship along again, then ship along again! Like a conveyer belt
- Notice that all **dplyr** functions take the data frame as the first argument, making it easy to chain them
- "Ships along" anything, including vectors or single numbers, not just data frames! Track what the object being shipped is in each step.

The pipe

- See how clean it can make the code!

```
mean(mtcars[mtcars$am == 1,]$cyl, na.rm = TRUE)
```

```
## [1] 5.076923
```

vs.

```
mtcars %>%
  filter(am == 1) %>%
  pull(cyl) %>%
  mean(na.rm = TRUE)
```

```
## [1] 5.076923
```

select()

- `select()` is like `filter()`, but instead of picking rows it picks columns
- Unlike `pull()` it doesn't give you a vector - it gives you back a data frame with fewer columns

```
mtcars %>%
  select(mpg, cyl) %>%
  summary()
```

```
##       mpg             cyl        
##  Min.   :10.40   Min.   :4.000  
##  1st Qu.:15.43   1st Qu.:4.000  
##  Median :19.20   Median :6.000  
##  Mean   :20.09   Mean   :6.188  
##  3rd Qu.:22.80   3rd Qu.:8.000  
##  Max.   :33.90   Max.   :8.000
```

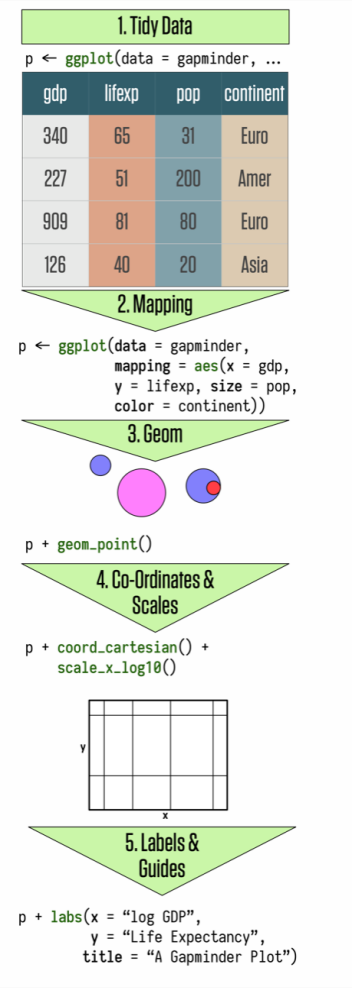
mutate()

- `mutate()` creates a new column using the old ones. You can assign multiple columns at once!
- The syntax is `NewVariableName = FunctionOfOldVariables`
- Don't forget to save the object!

```
mtcars ← mtcars %>%
  mutate(high_mpg = mpg > median(mpg))
mtcars %>%
  pull(high_mpg) %>%
  table()
```

```
## .
## FALSE  TRUE
##    17    15
```

Data vizualization with `ggplot`



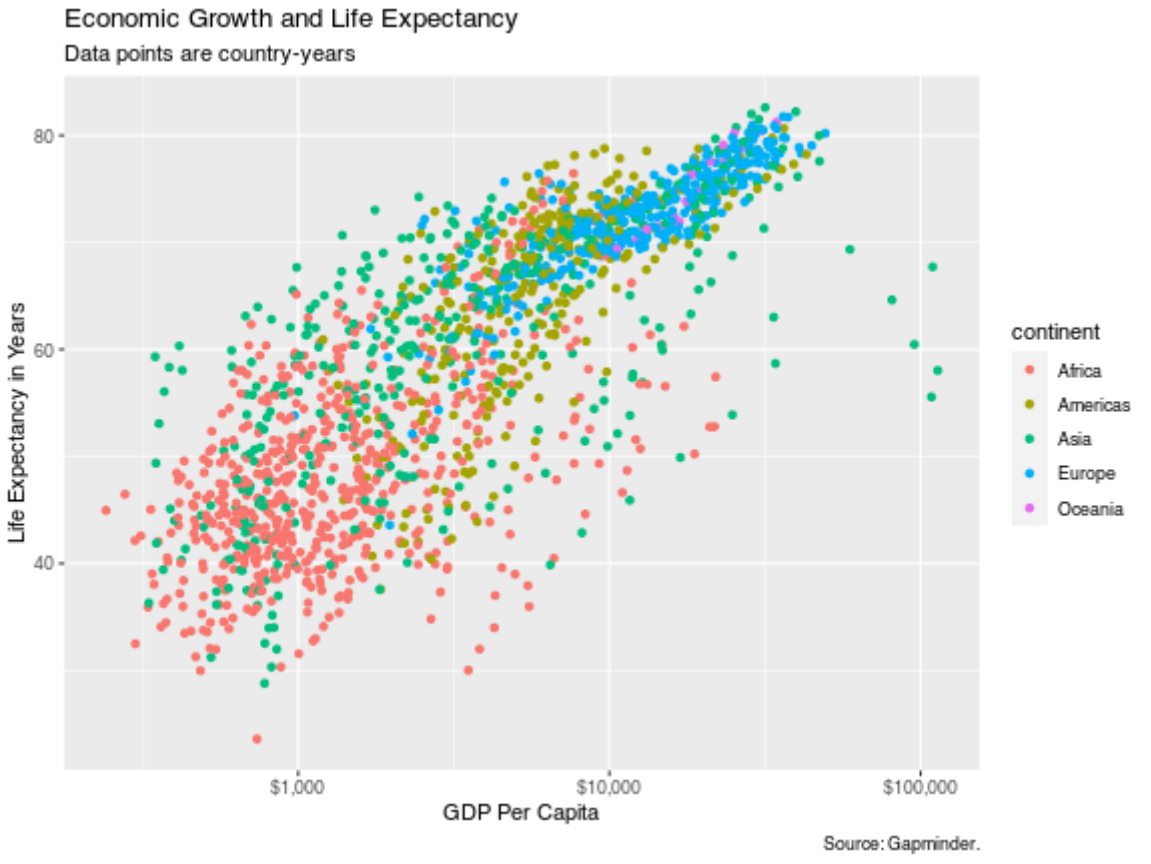
ggplot

- You begin every plot by telling the `ggplot()` function what your data is
- Then how the variables in this data logically map onto the plot's aesthetics (`aesthetic mappings` or just `aesthetics`)
- Then say what general sort of plot you want (called `geom`):
    - scatterplot: `geom_point()`
    - boxplot: `geom_bar()`
    - bar chart: `geom_boxplot()`
- All these pieces are combined using the symbol "+"
- You can find more details on https://socviz.co/makeplot.html#how-ggplot-works

Example ggplot

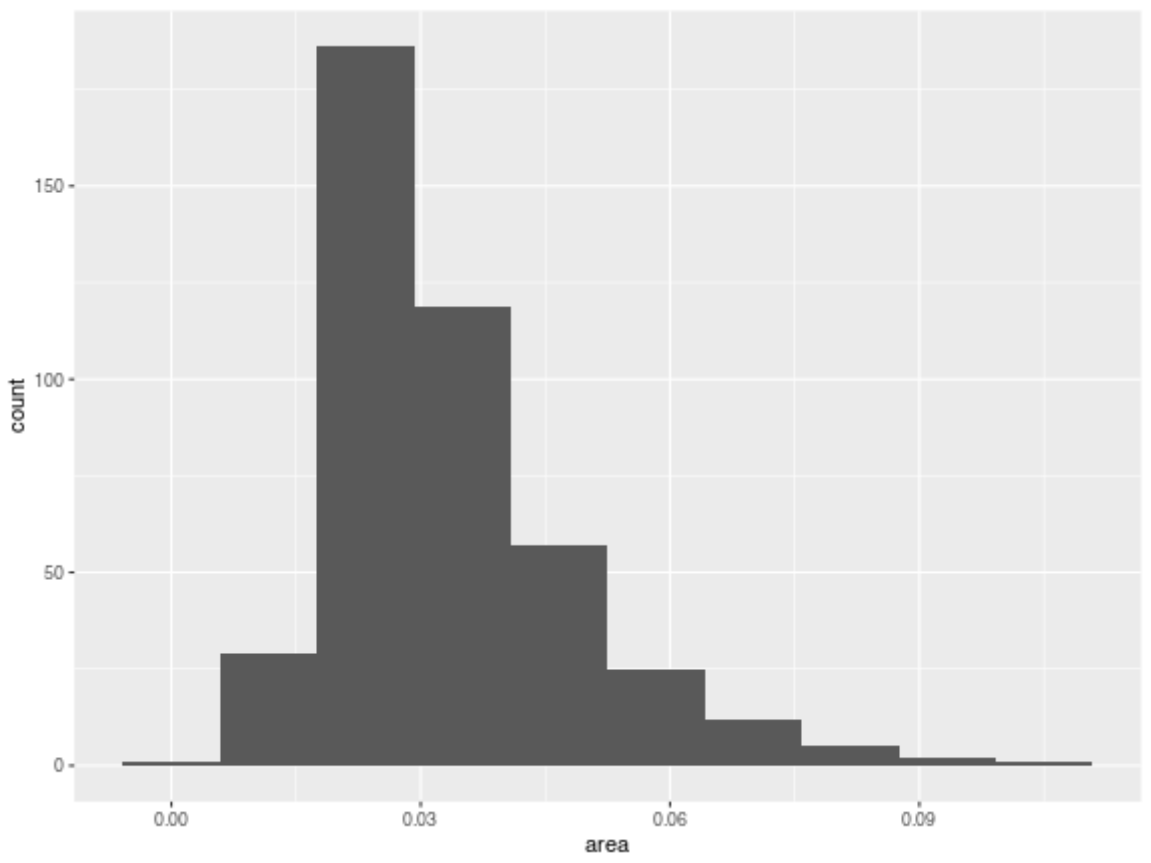
```
#install packages if needed
#install.packages("gapminder") # data we are using
#install.packages("ggplot2")
#load the packages
library (ggplot2)
library(gapminder)

#assign the plot to an object called p
p←ggplot(data=gapminder, aes(x=gdpPercap,y=lifeExp,color=continent))+
          # the date and variables we use
  geom_point()+ # type of plot
  scale_x_log10(labels = scales::dollar) + # change the units of x variable
  labs(x = "GDP Per Capita", y = "Life Expectancy in Years",
       title = "Economic Growth and Life Expectancy",
       subtitle = "Data points are country-years"
```

Example ggplot



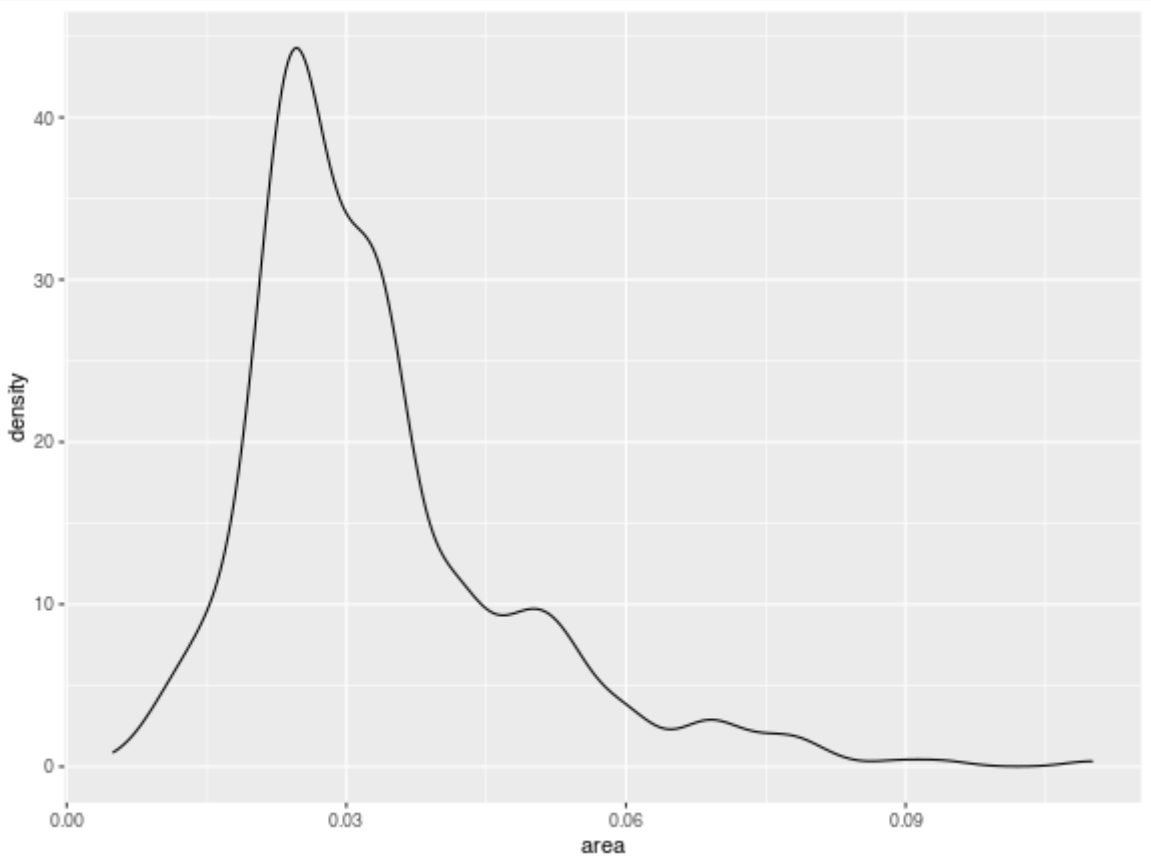
Histograms and density plots

```
p ← ggplot(data = midwest,
           mapping = aes(x = area))
p + geom_histogram(bins = 10)
```



Histograms and density plots

```
p ← ggplot(data = midwest,
           mapping = aes(x = area))
p + geom_density()
```



Histograms and density plots

```
p ← ggplot(data = midwest,
           mapping = aes(x = area, fill = state, color = state))
# We can use color (for the lines) and fill (for the body of the density curve) here

p + geom_density(alpha = 0.3)
```

Basics of R

- This has been a lot of information!
- This is mostly to show you what's there. To really learn it you'll have to get used to applying it yourself
- For that we will be using the **swirl** package which will walk us through using these commands!

Swirl

- Open up R and install Swirl with `install.packages('swirl')`
- Then, load up swirl with `library(swirl)`
- Download the Swirl course for this class (the first time you use it) with `install_course_github('NickCH-K','Econometrics')`
- Then start swirl with `swirl()` and pick Econometrics, and the R Basics lesson. Let's work through this!