

# Lecture 10

Base R // Tidyverse // Regression

---

Ivan Rudik  
AEM 4510

# Roadmap

- What is R?
- What is the tidyverse?
- How do we import and manipulate data?

Our goal is to take a hands on approach to learning how we actually **do** environmental economics

A good chunk of this lecture comes from Grant McDermott's **data science for economists** notes, Ed Rubin's **intro to econometrics**, and **RStudio education**

# RStudio Cloud

---

# Getting started

We will be using [rstudio.cloud](#) for our coding

# Getting started

We will be using **rstudio.cloud** for our coding

Why?

# Getting started

We will be using [rstudio.cloud](#) for our coding

Why?

You don't need to download/install anything

# Getting started

We will be using [rstudio.cloud](#) for our coding

Why?

You don't need to download/install anything

I can prepare the packages and code and make it easy to download

# Getting started

We will be using [rstudio.cloud](#) for our coding

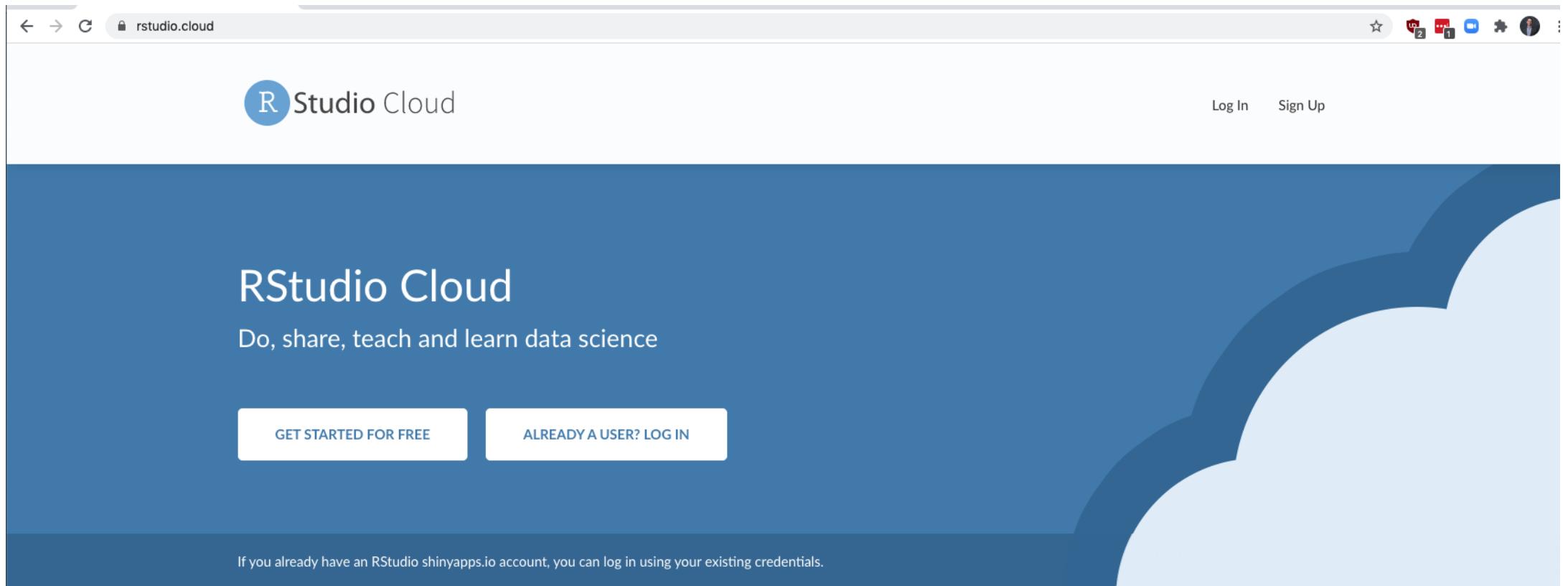
Why?

You don't need to download/install anything

I can prepare the packages and code and make it easy to download

Let's get everything going...

# Getting started: login



The screenshot shows the RStudio Cloud homepage. At the top, there's a navigation bar with icons for back, forward, refresh, and search, followed by the URL 'rstudio.cloud'. On the right side of the bar are various browser extension icons. Below the bar, the RStudio Cloud logo is on the left, and 'Log In' and 'Sign Up' buttons are on the right. The main section has a blue background with a white cloud graphic on the right. It features the text 'RStudio Cloud' and 'Do, share, teach and learn data science'. Below this are two buttons: 'GET STARTED FOR FREE' and 'ALREADY A USER? LOG IN'. At the bottom of this section, a note says 'If you already have an RStudio shinyapps.io account, you can log in using your existing credentials.'.

## Data science without the hardware hassles

RStudio Cloud is a lightweight, cloud-based solution that allows anyone to do, share, teach and learn data science online.

- Analyze your data using the RStudio IDE, directly from your browser.

\$ AVAILABLE PRICING PLANS

🕒 RSTUDIO CLOUD GUIDE

🌐 RSTUDIO.COM

# Getting started: new project

The screenshot shows the RStudio Cloud interface for the space **AEM 4510** (Ivan Rudik). The left sidebar includes sections for **Spaces** (Your Workspace, AEM 4510, AEM 6510, New Space), **Learn** (Guide, What's New, Primers, Cheat Sheets), and **Help**. The main content area displays the **All Projects** page with a project titled **In-10** by **Ivan Rudik**, created on Mar 29, 2021 at 10:22 AM. The interface includes filters for **List** (All projects) and **Sort** (By name), and a toolbar with **New Project** (New Project, New Project from Git Repository), **Delete**, **Move**, and **Export**.

# Getting started: wait for deployment

The screenshot shows the RStudio Cloud interface at the URL [rstudio.cloud/project/1795327](https://rstudio.cloud/project/1795327). The user is in the 'Your Workspace / aem6510' project. The left sidebar includes sections for Spaces (Your Workspace, New Space), Learn (Guide, What's New, Primers, Cheat Sheets), Help (Current System Status, RStudio Community), and Info (Plans & Pricing, Terms and Conditions). A central message bubble indicates 'Deploying Project' with a progress bar.

R Studio Cloud

Your Workspace / aem6510

Spaces

Your Workspace

New Space

Learn

Guide

What's New

Primers

Cheat Sheets

Help

Current System Status

RStudio Community

Info

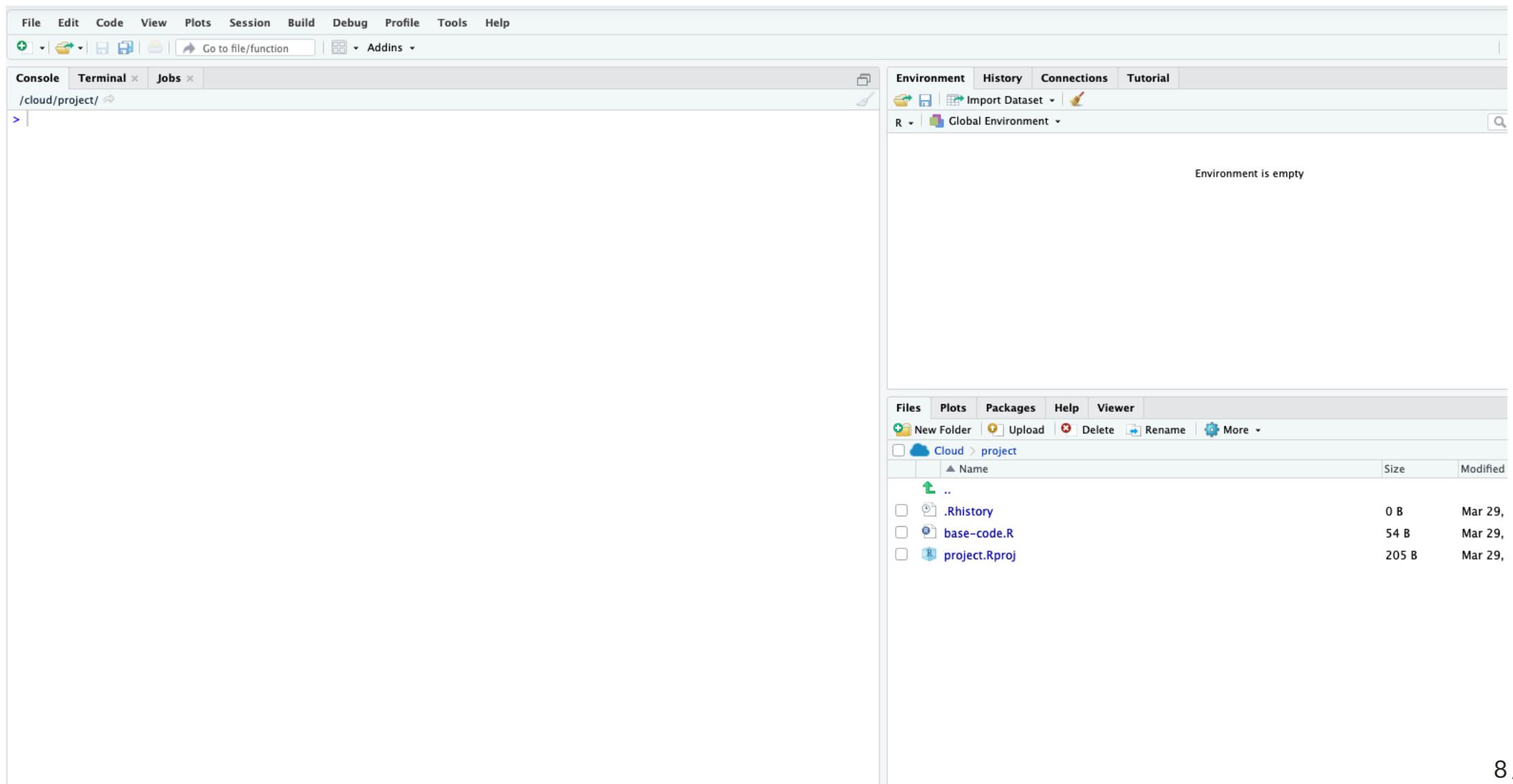
Plans & Pricing

Terms and Conditions

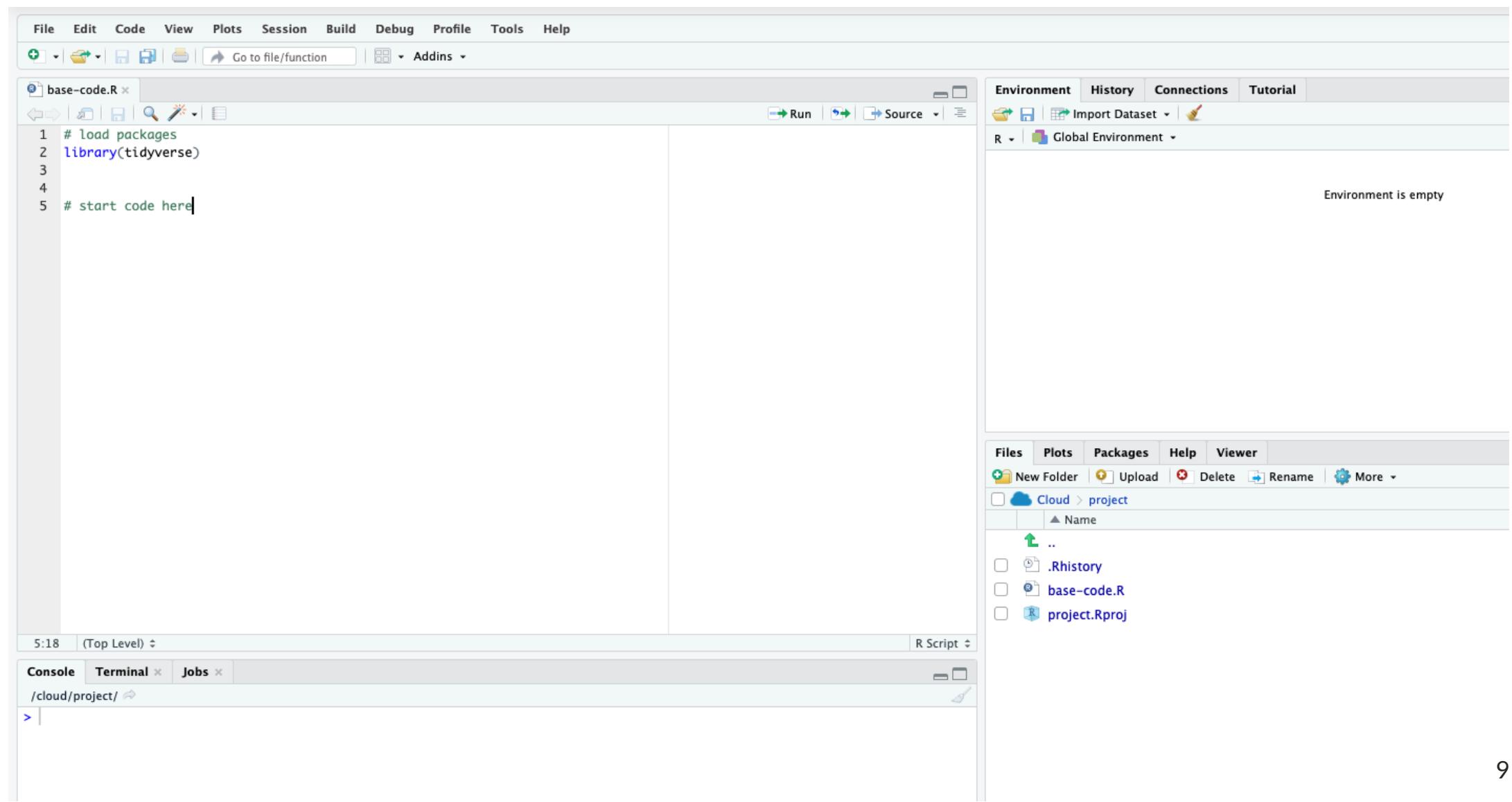
Deploying Project

7 / 174

# Click on base-code in bottom-right



# Code script open!



# Now we're set

Now we're all set with our coding environment

# Now we're set

Now we're all set with our coding environment

You can write code in the top window and save it as a file

# Now we're set

Now we're all set with our coding environment

You can write code in the top window and save it as a file

Or you can just enter it in the console in the bottom if you don't want to save it

# Now we're set

Now we're all set with our coding environment

You can write code in the top window and save it as a file

Or you can just enter it in the console in the bottom if you don't want to save it

Highlight code in the top window and press cmd+enter to run those highlighted lines

# Quick intro to R

---

# Introduction to base R

---

# Basic arithmetic

R is a powerful calculator and recognizes all of the standard arithmetic operators:

```
1+2 # add / subtraction
```

```
## [1] 3
```

```
5/2 # divide
```

```
## [1] 2.5
```

```
2^3 # exponentiate
```

```
## [1] 8
```

```
2+4*1^3 # standard order of precedence (`*` before `+`, etc.)
```

# Logic

R also comes equipped with a full set of logical operators and Booleans

```
1 > 2
```

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

```
(1 > 2) & (1 > 0.5) # "&" is the "and" operator
```

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

```
(1 > 2) | (1 > 0.5) # "|" is the "or" operator
```

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

# Logic

We can negate expressions with: !

This is helpful for filtering data

```
is.na(1:10)
```

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

```
!is.na(1:10)
```

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

NA means **not available** (i.e., missing)

# Logic

For value matching we can use: `%in%`

To see whether an object is contained within (i.e., matches one of) a list of items, use `%in%:`

```
4 %in% 1:10
```

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

```
4 %in% 5:10
```

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

# Logic

To evaluate whether two expressions are equal, we need to use **two** equal signs

```
1 = 1 # This doesn't work
```

```
## Error in 1 = 1: invalid (do_set) left-hand side to assignment
```

```
1 == 1 # This does.
```

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

```
1 != 2 # Note the single equal sign when combined with a negation.
```

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

# Logic

**Evaluation caveat:** What do you think will happen if we evaluate  $0.1 + 0.2 == 0.3$ ?

# Logic

**Evaluation caveat:** What do you think will happen if we evaluate `0.1 + 0.2 == 0.3`?

```
0.1 + 0.2 == 0.3
```

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

Uh-oh! What went wrong here?

# Logic

**Evaluation caveat:** What do you think will happen if we evaluate `0.1 + 0.2 == 0.3?`

```
0.1 + 0.2 == 0.3
```

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

Uh-oh! What went wrong here?

**Problem:** Computers represent numbers as binary (i.e., base 2) floating-points. More [here](#)

- Fast and memory efficient, but can lead to unexpected behavior
- Similar to how standard decimals can't precisely capture certain fractions

# Logic

**Evaluation caveat:** What do you think will happen if we evaluate `0.1 + 0.2 == 0.3?`

```
0.1 + 0.2 == 0.3
```

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

Uh-oh! What went wrong here?

**Problem:** Computers represent numbers as binary (i.e., base 2) floating-points. More [here](#)

- Fast and memory efficient, but can lead to unexpected behavior
- Similar to how standard decimals can't precisely capture certain fractions

# Assignment

In R, we can use either `<-` or `=` to handle assignment

# Assignment

In R, we can use either `<-` or `=` to handle assignment

Assignment with `=`

`<-` is normally read aloud as "gets". You can think of it as a (left-facing) arrow saying *assign in this direction*.

```
a = 10 + 5  
a
```

```
## [1] 15
```

# Assignment

Assignment with `=`

You can also use `=` for assignment.

```
b = 10 + 10  
b
```

```
## [1] 20
```

# Which assignment operator should you use?

The proper one to use is <- , which can be inserted using the keyboard shortcut Alt/Option + -

It doesn't really matter for our purposes, other languages use =

I will use =

**Bottom line:** Use whichever you prefer, just be consistent

# Help

For more information on a (named) function or object in R, consult the "help" documentation using ?

For example:

```
?plot
```

# Vignettes

For some packages, `vignette()` will provide a detailed intro

```
# Try this:  
vignette("dplyr")
```

Vignettes are a great way to learn how and when to use a package

# Comments

Comments in R code are demarcated by `#`

Use comments to document your logic in `.R` scripts and within `.Rmd` code chunks

```
# THIS IS A CODE SECTION ----  
# this is a comment  
cornell = "big red"
```

# Comments

Comments in R code are demarcated by #

Use comments to document your logic in .R scripts and within .Rmd code chunks

```
# THIS IS A CODE SECTION ----  
# this is a comment  
cornell = "big red"
```

Comments should be concise and used only when necessary (unlike the comments above)

# Comments

```
# THIS IS A CODE SECTION ----  
# this is a comment  
cornell = "big red"
```

Using at least four trailing dashes ( ---- ) creates a code section, which simplifies navigation and code folding

- Also works with trailing equals ( ===== ) or pound signs ( ##### )

# Comments

```
# THIS IS A CODE SECTION ----  
# this is a comment  
cornell = "big red"
```

Using at least four trailing dashes ( ---- ) creates a code section, which simplifies navigation and code folding

- Also works with trailing equals ( ===== ) or pound signs ( ##### )

**Keyboard shortcut:** use `Ctrl/Cmd+/` or `Ctrl/Cmd+Shift+C` in RStudio to (un)comment whole sections of highlighted code

# Object-oriented programming in R

---

# Object-oriented programming

In R:

"Everything is an object and everything has a name."

"Everything is an object"

# What are objects?

There are many different *types* (or *classes*) of objects

Here are some objects that we'll be working with regularly:

- vectors
- matrices
- data frames
- lists
- functions

# Data frames

The most important object we will be working with is the **data frame**

You can think of it basically as an Excel spreadsheet

```
# Create a small data frame called "d"  
d = data.frame(x = 1:2, y = 3:4)  
d
```

```
##   x y  
## 1 1 3  
## 2 2 4
```

# Data frames

The most important object we will be working with is the **data frame**

You can think of it basically as an Excel spreadsheet

```
# Create a small data frame called "d"  
d = data.frame(x = 1:2, y = 3:4)  
d
```

```
##   x y  
## 1 1 3  
## 2 2 4
```

This is essentially just a table with columns named `x` and `y`

# Data frames

The most important object we will be working with is the **data frame**

You can think of it basically as an Excel spreadsheet

```
# Create a small data frame called "d"  
d = data.frame(x = 1:2, y = 3:4)  
d
```

```
##   x y  
## 1 1 3  
## 2 2 4
```

This is essentially just a table with columns named `x` and `y`

Each row is an observation telling us the values of both `x` and `y`

# Aside: built-in data frames

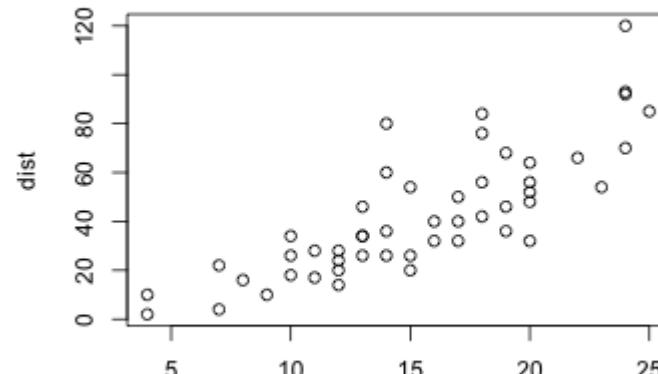
Base R and packages have a bunch of built in data frames with special names you can call on

For example we have `cars`:

```
head(cars)
```

```
##      speed dist
## 1        4    2
## 2        4   10
## 3        7    4
## 4        7   22
## 5        8   16
## 6        9   10
```

```
plot(cars)
```



# Back to objects

Each object class has its own set of rules for determining valid operations

```
# Create a small data frame called "d"  
d = data.frame(x = 1:2, y = 3:4)  
d
```

```
##   x y  
## 1 1 3  
## 2 2 4
```

# Back to objects

At the same time, you can (usually) convert an object from one type to another

```
# Convert it to (i.e., create) a matrix call "mat"
mat = as.matrix(d)
mat
```

```
##      x y
## [1,] 1 3
## [2,] 2 4
```

# Working with multiple objects

In R we can have multiple data frames in memory at once

Even though we just made `mat`, `d` still exists:

```
d
```

```
##   x y
## 1 1 3
## 2 2 4
```

# Ways to learn about objects

Printing an object directly in the console is often handy

- e.g., Type `d` and hit Enter

# Ways to learn about objects

Printing an object directly in the console is often handy

- e.g., Type `d` and hit Enter

`View()` is very helpful, and has the same effect as clicking on the object in your RStudio *Environment* pane

- e.g., Type `View(d)` and hit Enter

# Ways to learn about objects

Use the `str` command to learn about an object's **structure**

```
# d = data.frame(x = 1:2, y = 3:4) # Create a small data frame called "d"  
str(d) # Evaluate its structure
```

```
## 'data.frame':    2 obs. of  2 variables:  
##   $ x: int  1 2  
##   $ y: int  3 4
```

# Ways to learn about objects

Use the `str` command to learn about an object's **structure**

```
# d = data.frame(x = 1:2, y = 3:4) # Create a small data frame called "d"  
str(d) # Evaluate its structure
```

```
## 'data.frame': 2 obs. of 2 variables:  
## $ x: int 1 2  
## $ y: int 3 4
```

You can also use `class` to get an object's class without all the other details

# Global environment

Let's go back to the simple data frame that we created a few slides earlier.

```
d
```

```
##   x y
## 1 1 3
## 2 2 4
```

# Global environment

Let's go back to the simple data frame that we created a few slides earlier.

```
d
```

```
##   x y  
## 1 1 3  
## 2 2 4
```

Now, let's try to do a logical comparison of these "x" and "y" variables:

```
x < y
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

# Global environment

Let's go back to the simple data frame that we created a few slides earlier.

```
d
```

```
##   x  y  
## 1 1  3  
## 2 2  4
```

Now, let's try to do a logical comparison of these "x" and "y" variables:

```
x < y
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

Uh-oh. What went wrong here?

# Global environment

The error message provides the answer to our question:

```
## Error in eval(predvars, data, env): object 'x' not found
```

# Global environment

The error message provides the answer to our question:

```
## Error in eval(predvars, data, env): object 'x' not found
```

R looked in our *Global Environment* and couldn't find x

The screenshot shows the RStudio interface with the 'Global Environment' tab selected. The window displays the following variables:

Data	Values
d	2 obs. of 2 variables
mat	int [1:2, 1:2] 1 2 3 4
Values	
a	15
b	20
vibe	"immaculate"

# Global environment

The error message provides the answer to our question:

```
## Error in eval(predvars, data, env): object 'x' not found
```

R looked in our *Global Environment* and couldn't find x

The screenshot shows the RStudio interface with the 'Environment' tab selected. The global environment contains the following objects:

Data	Values
d	2 obs. of 2 variables
mat	int [1:2, 1:2] 1 2 3 4
a	15
b	20
vibe	"immaculate"

We have to tell R that x and y belong to the object d

"Everything has a name"

---

# Reserved words

R has a bunch of key/reserved words that serve specific functions

- You can't (re)assign these, even if you wanted to

See [here](#) for a full list, including (but not limited to):

```
if  
else  
while # looping  
function  
for # looping  
TRUE  
FALSE  
NULL # null/undefined  
Inf #infinity  
NaN # not a number  
NA # not available / missing
```

# Semi-reserved words

There are other words that are sort of reserved, in that they have a particular meaning

- These are named functions or constants (e.g., `pi`) that you can re-assign if you really want to... but that already come with important meanings from base R

The most important example is `c()`, which binds and concatenates objects together

```
my_vector = c(1, 2, 5)  
my_vector
```

# Semi-reserved words

What do you think will happen if you type the following?

```
c = 4  
c(1, 2 ,5)
```

# Semi-reserved words

What do you think will happen if you type the following?

```
c = 4  
c(1, 2 ,5)
```

```
## [1] 1 2 5
```

# Semi-reserved words

What do you think will happen if you type the following?

```
c = 4  
c(1, 2, 5)
```

```
## [1] 1 2 5
```

In this case, R is "smart" enough to distinguish between the variable `c` and the built-in function `c()`

# Semi-reserved words (cont.)

But R won't always be able to distinguish between conflicting definitions! For example:

```
pi
```

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

```
pi = 2  
pi
```

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

# Semi-reserved words (cont.)

But R won't always be able to distinguish between conflicting definitions! For example:

```
pi
```

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

```
pi = 2  
pi
```

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

**Bottom line:** Don't use (semi-)reserved words!

# Indexing

---

# Indexing

How do we index in R?

# Indexing

How do we index in R?

We've already seen an example of indexing in the form of R console output:

```
1+2
```

```
## [1] 3
```

The `[1]` above denotes the first (and, in this case, only) element of our output

# Indexing

How do we index in R?

We've already seen an example of indexing in the form of R console output:

```
1+2
```

```
## [1] 3
```

The `[1]` above denotes the first (and, in this case, only) element of our output

In this case, a vector of length one equal to the value "3"

# Indexing

Try the following in your console to see a more explicit example of indexed output:

```
rnorm(n = 50, mean = 0, sd = 1) # take 50 draws from the standard normal distribution
```

# Indexing

Try the following in your console to see a more explicit example of indexed output:

```
rnorm(n = 50, mean = 0, sd = 1) # take 50 draws from the standard normal distribution
```

```
## [1] -1.41483518 -1.39591993 -1.02013413 -0.13671991  1.18660542  0.80462978  2.33453926 -1.10420607  
## [9]  1.21568002 -1.13151261 -0.12103046  0.34546620  1.09379966  0.70560678 -0.50982841  0.82786513  
## [17] -0.12755617 -0.82284974  0.73899620 -1.93849098 -0.03876197 -0.07471929  0.55791366 -1.83303264  
## [25] -0.84005486 -0.33578374 -0.44677895  1.07054240  1.75592590 -0.66583994  1.26048726  0.27863227  
## [33]  1.08555987 -0.19758615 -0.82784541  0.03684417  0.49127734  0.85535554 -0.83424336 -1.06534754  
## [41] -0.41584392 -0.58796263 -1.19953677  0.11288369  1.35528075 -0.09053809  0.36907499  0.64991460  
## [49] -0.91154457  0.96397401
```

# Option 1: []

We can use `[]` to index objects that we create in R

```
a = 1:10  
a[4] ## Get the 4th element of object "a"
```

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

```
a[c(4, 6)] ## Get the 4th and 6th elements
```

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

# Option 1: []

This also works on larger arrays (vectors, matrices, data frames, and lists)

```
starwars[1, 1] ## Show the cell corresponding to the 1st row & 1st column of the data frame.
```

```
## # A tibble: 1 × 1
##   name
##   <chr>
## 1 Luke Skywalker
```

# Option 1: []

This also works on larger arrays (vectors, matrices, data frames, and lists)

```
starwars[1, 1] ## Show the cell corresponding to the 1st row & 1st column of the data frame.
```

```
## # A tibble: 1 × 1
##   name
##   <chr>
## 1 Luke Skywalker
```

What does `starwars[1:3, 1]` give you?

# Option 1: []

This also works on larger arrays (vectors, matrices, data frames, and lists)

```
starwars[1, 1] ## Show the cell corresponding to the 1st row & 1st column of the data frame.
```

```
## # A tibble: 1 × 1
##   name
##   <chr>
## 1 Luke Skywalker
```

What does `starwars[1:3, 1]` give you?

```
## # A tibble: 3 × 1
##   name
##   <chr>
## 1 Luke Skywalker
## 2 C-3PO
## 3 R2-D2
```

## Option 1: []

We haven't discusssed them yet, but **lists** are a more complex type of array object in R

## Option 1: []

We haven't discussed them yet, but **lists** are a more complex type of array object in R

They can contain a collection of objects that don't share the same structure

## Option 1: []

We haven't discussed them yet, but **lists** are a more complex type of array object in R

They can contain a collection of objects that don't share the same structure

For example, you can have lists containing:

- a scalar, a string, and a data frame
- a list of data frames
- a list of lists

# Option 1: []

The relevance to indexing is that lists require two square brackets `[][]` to index the parent list item and then the standard `[]` within that parent item. An example might help to illustrate:

```
my_list = list(  
  a = "hello",  
  b = c(1,2,3),  
  c = data.frame(x = 1:5, y = 6:10))  
my_list[[1]] # Return the 1st list object
```

```
## [1] "hello"
```

```
my_list[[2]][3] # Return the 3rd element of the 2nd list object
```

```
## [1] 3
```

# Option 2: \$

Lists provide a nice segue to our other indexing operator: \$

- Let's continue with the my\_list example from the previous slide.

```
my_list
```

```
## $a
## [1] "hello"
##
## $b
## [1] 1 2 3
##
## $c
##   x   y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 12
```

# Option 2: \$

Lists provide a nice segue to our other indexing operator: \$.

- Let's continue with the my\_list example from the previous slide

```
my_list
```

```
## $a
## [1] "hello"
##
## $b
## [1] 1 2 3
##
## $c
##   x  y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 12
```

# Option 2: \$

We can call these objects directly by name using the dollar sign, e.g.

```
my_list$a ## Return list object "a"
```

```
## [1] "hello"
```

```
my_list$b[3] ## Return the 3rd element of list object "b"
```

```
## [1] 3
```

```
my_list$c$x ## Return column "x" of list object "c"
```

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

## Option 2: \$

The \$ form of indexing also works for other object types

In some cases, you can also combine the two index options:

```
starwars$name[1]  
## [1] "Luke Skywalker"
```

## Option 2: \$

Finally, `$` provides another way to avoid the "object not found" problem that we ran into earlier

```
x < y # Doesn't work
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

```
d$x < d$y # Works!
```

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

# Cleaning up

---

# Removing objects

Use `rm()` to remove an object or objects from your working environment

```
a = "hello"  
b = "world"  
rm(a, b)
```

You can use `rm(list = ls())` to remove all objects in your working environment, though this is **frowned upon**

- Better just to start a new R session

# The tidyverse

---

# What is "tidy" data?

What we are going to learn is how to use a set of packages called the **tidyverse**

# What is "tidy" data?

What we are going to learn is how to use a set of packages called the **tidyverse**

These sets of packages make working with data **extremely easy** and **intuitive**

# What is "tidy" data?

Resources:

- [Vignette](#) (from the **tidyr** package)
- [Original paper](#) (Hadley Wickham, 2014 JSS)

# What is "tidy" data?

Resources:

- [Vignette](#) (from the `tidyr` package)
- [Original paper](#) (Hadley Wickham, 2014 JSS)

Key points:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

# What is "tidy" data?

Resources:

- [Vignette](#) (from the `tidyr` package)
- [Original paper](#) (Hadley Wickham, 2014 JSS)

Key points:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Basically, tidy data is more likely to be [long \(i.e. narrow\)](#) than wide

# Checklist

Install tidyverse: `install.packages('tidyverse')` (already done on cloud)

Install nycflights13: `install.packages('nycflights13', repos = 'https://cran.rstudio.com')`

# Tidyverse vs. base R

Lots of debate over tidyverse vs base R

# Tidyverse vs. base R

Lots of debate over tidyverse vs base R

The answer is **obvious**: We should teach the tidyverse first

- Good documentation and support
- Consistent philosophy and syntax
- Nice front-end for big data tools
- For data cleaning, plotting, the tidyverse is elite

# Tidyverse vs. base R

Base R is still great

- Base R is extremely flexible and powerful
- The tidyverse can't do everything
- Using base R and the tidyverse together is often a good idea

# Tidyverse vs. base R

One point of convenience is that there is often a direct correspondence between a tidyverse command and its base R equivalent:

tidyverse	base
?readr::read_csv	?utils::read.csv
?dplyr::if_else	?base::ifelse
?tibble::tibble	?base::data.frame

Tidyverse functions typically have extra features on top of base R

# Tidyverse vs. base R

One point of convenience is that there is often a direct correspondence between a tidyverse command and its base R equivalent:

tidyverse	base
?readr::read_csv	?utils::read.csv
?dplyr::if_else	?base::ifelse
?tibble::tibble	?base::data.frame

Tidyverse functions typically have extra features on top of base R

There are always many ways to achieve a single goal in R

# Tidyverse packages

Let's load the tidyverse meta-package and check the output.

```
library(tidyverse)
```

# Tidyverse packages

Let's load the tidyverse meta-package and check the output.

```
library(tidyverse)
```

We have actually loaded a number of packages: **ggplot2**, **tibble**, **dplyr**, etc

# Tidyverse packages

Let's load the tidyverse meta-package and check the output.

```
library(tidyverse)
```

We have actually loaded a number of packages: **ggplot2**, **tibble**, **dplyr**, etc

We can also see information about the package versions and some **namespace conflicts**

# Tidyverse packages

The tidyverse actually comes with a lot more packages than those that are just loaded automatically

```
tidyverse_packages()
```

```
## [1] "broom"          "cli"            "crayon"         "dbplyr"        "dplyr"          "dtplyr"  
## [7] "forcats"        "googledrive"    "googlesheets4" "ggplot2"       "haven"         "hms"  
## [13] "httr"           "jsonlite"       "lubridate"      "magrittr"      "modelr"        "pillar"  
## [19] "purrr"          "readr"          "readxl"         "reprex"        "rlang"         "rstudioapi"  
## [25] "rvest"          "stringr"        "tibble"         "tidyrm"        "xml2"          "tidyverse"
```

e.g. the **lubridate** package is for working with dates and the **rvest** package is for webscraping

# Tidyverse packages

The tidyverse actually comes with a lot more packages than those that are just loaded automatically

```
tidyverse_packages()
```

```
## [1] "broom"          "cli"            "crayon"         "dbplyr"        "dplyr"          "dtplyr"  
## [7] "forcats"        "googledrive"    "googlesheets4" "ggplot2"       "haven"         "hms"  
## [13] "httr"           "jsonlite"       "lubridate"      "magrittr"      "modelr"        "pillar"  
## [19] "purrr"          "readr"          "readxl"         "reprex"        "rlang"         "rstudioapi"  
## [25] "rvest"          "stringr"        "tibble"         "tidyrr"        "xml2"          "tidyverse"
```

e.g. the **lubridate** package is for working with dates and the **rvest** package is for webscraping

These packages have to be loaded separately

# Tidyverse packages

We're going to focus on two workhorse packages:

1. **dplyr**
2. **tidyr**

These are the packages for cleaning and wrangling data

# Tidyverse packages

We're going to focus on two workhorse packages:

1. **dplyr**
2. **tidyr**

These are the packages for cleaning and wrangling data

They are thus the ones that you will likely make the most use of

# Tidyverse packages

We're going to focus on two workhorse packages:

1. **dplyr**
2. **tidyr**

These are the packages for cleaning and wrangling data

They are thus the ones that you will likely make the most use of

Data cleaning and wrangling is important and knowing how to do it well is a good skill for any data-oriented job

# Pipes: |>

The pipe operator `|>` lets us perform a sequence of operations in a very nice and tidy way

# Pipes: |>

The pipe operator `| >` lets us perform a sequence of operations in a very nice and tidy way

Let's consider a fake example to get the idea for why its really beneficial

# Pipes: |>

Let's say we wanted to apply a sequence of operations that tells the computer what you did throughout your day:

1. Wake up
2. Get out of bed
3. Comb hair
4. Go downstairs
5. Drink a cup
6. Grab hat
7. Catch bus

# Pipes: |>

If you were to code this up in a traditional way it might look one of two ways:

A bunch of lines doing all the different steps

```
me = wake_up(me)
me = get_out_of_bed(me)
me = comb_hair(me)
me = go(me, "downstairs")
me = drink(me, "cup")
me = grab(me, "hat")
me = catch(me, "bus")
```

# Pipes: |>

If you were to code this up in a traditional way it might look one of two ways:

Or if you're a little crazy then do it all in one line

```
me = catch(grab(drink(go(comb_hair(get_out_of_bed(wake_up(me)))), where = "downstairs"), what = "cups")))
```

These are kind of tedious or messy and out of the order you'd think

# Pipes: |>

With pipes we can do everything at once, but have it be **in order**:

```
me = me |>  
  wake_up() |>  
  get_out_of_bed() |>  
  comb_hair() |>  
  go("downstairs") |>  
  drink("cup") |>  
  grab("hat") |>  
  catch("bus")
```

This makes everything **very intuitive** to read and code!

# Pipes: |>

Here's a real example: suppose we wanted to figure out the average highway miles per gallon of Audi's in the `mpg` dataset:

```
mpg
```

```
## # A tibble: 234 × 11
##   manufacturer model      displ  year   cyl trans   drv   cty   hwy fl class
##   <chr>        <chr>     <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi         a4          1.8  1999     4 auto(l5)   f     18    29  p   compact
## 2 audi         a4          1.8  1999     4 manual(m5) f     21    29  p   compact
## 3 audi         a4          2    2008     4 manual(m6) f     20    31  p   compact
## 4 audi         a4          2    2008     4 auto(av)    f     21    30  p   compact
## 5 audi         a4          2.8  1999     6 auto(l5)    f     16    26  p   compact
## 6 audi         a4          2.8  1999     6 manual(m5) f     18    26  p   compact
## 7 audi         a4          3.1  2008     6 auto(av)    f     18    27  p   compact
## 8 audi         a4 quattro  1.8  1999     4 manual(m5) 4     18    26  p   compact
## 9 audi         a4 quattro  1.8  1999     4 auto(l5)    4     16    25  p   compact
## 10 audi        a4 quattro 2    2008     4 manual(m6) 4     20    28  p   compact
## # ... with 224 more rows
```

# Pipes: |>

There's two ways you might do this without taking advantage of pipes:

# Pipes: |>

There's two ways you might do this without taking advantage of pipes:

The first is to do it step-by-step, line-by-line which requires a lot of variable assignment

```
audis_mpg = filter(mpg, manufacturer=="audi")
audis_mpg_grouped = group_by(filter(mpg, manufacturer=="audi"), model)
summarise(audis_mpg_grouped, hwy_mean = mean(hwy))
```

```
## # A tibble: 3 × 2
##   model      hwy_mean
##   <chr>        <dbl>
## 1 a4            28.3
## 2 a4 quattro    25.8
## 3 a6 quattro    24
```

# Pipes: |>

Next you could do it all in one line which is hard to read

```
summarise(group_by(filter(mpg, manufacturer=="audi")), model), hwy_mean = mean(hwy))
```

```
## # A tibble: 3 × 2
##   model      hwy_mean
##   <chr>       <dbl>
## 1 a4          28.3
## 2 a4 quattro  25.8
## 3 a6 quattro  24
```

# Pipes: |>

Or, you could use **pipes** |>:

```
mpg |> filter(manufacturer=="audi") |> group_by(model) |> summarise(hwy_mean = mean(hwy))
```

```
## # A tibble: 3 × 2
##   model      hwy_mean
##   <chr>       <dbl>
## 1 a4          28.3
## 2 a4 quattro  25.8
## 3 a6 quattro  24
```

# Pipes: |>

Or, you could use **pipes** |>:

```
mpg |> filter(manufacturer=="audi") |> group_by(model) |> summarise(hwy_mean = mean(hwy))
```

```
## # A tibble: 3 × 2
##   model      hwy_mean
##   <chr>       <dbl>
## 1 a4          28.3
## 2 a4 quattro  25.8
## 3 a6 quattro  24
```

It performs the operations from left to right, exactly like you'd think of them: take this object (mpg), do this (filter), then do this (group by car model), then do this (take the mean of highway miles)

# Use vertical space

Pipes are even more readable if we write it over several lines:

```
mpg |>
  filter(manufacturer=="audi") |>
  group_by(model) |>
  summarise(hwy_mean = mean(hwy))
```

```
## # A tibble: 3 × 2
##   model      hwy_mean
##   <chr>        <dbl>
## 1 a4            28.3
## 2 a4 quattro    25.8
## 3 a6 quattro    24
```

Using vertical space costs nothing and makes for much more readable code

# dplyr

---

# Aside: dplyr 1.0.0 release

Please make sure that you are running at least **dplyr** 1.0.0 before continuing.

```
packageVersion("dplyr")
```

```
## [1] '1.0.8'
```

```
# install.packages('dplyr') ## install updated version if < 1.0.0
```

# The five key dplyr verbs

1. `filter`: Subset/filter rows based on their values
2. `arrange`: Reorder/arrange rows based on their values
3. `select`: Select columns/variables
4. `mutate`: Create new columns/variables
5. `summarise`: Collapse multiple rows into a single summary value, potentially by a grouping variable

# The five key dplyr verbs

1. `filter`: Subset/filter rows based on their values
2. `arrange`: Reorder/arrange rows based on their values
3. `select`: Select columns/variables
4. `mutate`: Create new columns/variables
5. `summarise`: Collapse multiple rows into a single summary value, potentially by a grouping variable

Let's practice these commands together using the `starwars` data frame that comes pre-packaged with `dplyr`

# Starwars

Here's the `starwars` dataset, it has 87 observations of 14 variables

```
starwars
```

```
## # A tibble: 87 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex gender homeworld species
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>        <dbl> <chr> <chr> <chr>    <chr>
## 1 Luke Skywalker 172     77 blond      fair        blue          19 male   masculin… Tatooine Human
## 2 C-3PO           167     75 <NA>       gold        yellow        112 none   masculin… Tatooine Droid
## 3 R2-D2            96     32 <NA>       white, bl… red          33 none   masculin… Naboo   Droid
## 4 Darth Vader    202    136 none       white        yellow        41.9 male   masculin… Tatooine Human
## 5 Leia Organa    150     49 brown      light       brown          19 female feminin… Alderaan Human
## 6 Owen Lars      178    120 brown, gr… light       blue          52 male   masculin… Tatooine Human
## 7 Beru Whitesun 165     75 brown      light       blue          47 female feminin… Tatooine Human
## 8 R5-D4            97     32 <NA>       white, red red          NA none   masculin… Tatooine Droid
## 9 Biggs Darkli… 183     84 black      light       brown          24 male   masculin… Tatooine Human
## 10 Obi-Wan Keno… 182     77 auburn, w… fair        blue-gray      57 male   masculin… Stewjon Human
## # ... with 77 more rows, and 2 more variables: vehicles <list>, starships <list>
```

# 1) dplyr::filter

Why filter?

# 1) dplyr::filter

Why filter?

Filtering subsets your data

# 1) dplyr::filter

Why filter?

Filtering subsets your data

This means that you can take out data that meet certain characteristics

# 1) dplyr::filter

Why filter?

Filtering subsets your data

This means that you can take out data that meet certain characteristics

e.g. if you want to run your analysis only on low income areas, or if you want to focus on years after 2005

# 1) dplyr::filter

Here we are subsetting the observations of humans that are at least 190cm

```
starwars |>  
  filter(  
    species == "Human",  
    height >= 190  
  )
```

```
## # A tibble: 4 × 14  
##   name  height  mass hair_color skin_color eye_color birth_year sex  gender homeworld species films  
##   <chr>   <int> <dbl> <chr>       <chr>       <chr>          <dbl> <chr> <chr> <chr>       <chr> <list>  
## 1 Dart...     202    136 none      white       yellow        41.9 male  masculin... Tatooine Human <chr>  
## 2 Qui-...     193     89 brown     fair        blue         92   male  masculin... <NA>   Human <chr>  
## 3 Dooku      193     80 white     fair        brown        102  male  masculin... Serenno Human <chr>  
## 4 Bail...     191     NA black     tan         brown        67   male  masculin... Alderaan Human <chr>  
## # ... with 1 more variable: starships <list>
```

# 1) dplyr::filter

You can filter using regular expressions with grep-type commands or the `stringr` package

```
starwars |>  
  filter(stringr::str_detect(name, "Skywalker"))
```

```
## # A tibble: 3 × 14  
##   name  height  mass hair_color skin_color eye_color birth_year sex  gender homeworld species films  
##   <chr>  <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>  <chr>      <chr>  <list>  
## 1 Luke...    172     77 blond     fair       blue           19   male  masculin... Tatooine Human  <chr>  
## 2 Anak...    188     84 blond     fair       blue          41.9  male  masculin... Tatooine Human  <chr>  
## 3 Shmi...    163     NA black     fair       brown          72   female feminin... Tatooine Human  <chr>  
## # ... with 1 more variable: starships <list>
```

This subsets the observations for individuals whose names contain "Skywalker"

# 1) dplyr::filter

A very common `filter` use case is identifying/removing missing data cases:

```
starwars |>  
  filter(is.na(height))
```

```
## # A tibble: 6 × 14  
##   name  height  mass hair_color skin_color eye_color birth_year sex  gender homeworld species films  
##   <chr>   <int> <dbl> <chr>       <chr>       <chr>           <dbl> <chr> <chr>       <chr>       <chr> <list>  
## 1 Arve...     NA     NA brown      fair        brown            NA male  masculin... <NA>       Human    <chr>  
## 2 Finn       NA     NA black      dark        dark             NA male  masculin... <NA>       Human    <chr>  
## 3 Rey        NA     NA brown      light       hazel            NA femal... feminin... <NA>       Human    <chr>  
## 4 Poe ...    NA     NA brown      light       brown            NA male  masculin... <NA>       Human    <chr>  
## 5 BB8        NA     NA none       none       black            NA none  masculin... <NA>       Droid    <chr>  
## 6 Capt...     NA     NA unknown   unknown    unknown           NA <NA> <NA>       <NA>       <NA> <chr>  
## # ... with 1 more variable: starships <list>
```

# 1) dplyr::filter

To remove missing observations, use negation:

```
starwars |>  
  filter(!is.na(height))
```

```
## # A tibble: 81 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex gender homeworld species  
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr> <chr>       <chr>  
## 1 Luke Skywalker 172    77  blond      fair        blue         19  male  masculin… Tatooine Human  
## 2 C-3PO          167    75 <NA>       gold        yellow       112  none  masculin… Tatooine Droid  
## 3 R2-D2          96     32 <NA>       white, bl… red         33  none  masculin… Naboo  Droid  
## 4 Darth Vader   202   136  none       white        yellow       41.9 male  masculin… Tatooine Human  
## 5 Leia Organa   150    49  brown      light       brown        19  femal… feminin… Alderaan Human  
## 6 Owen Lars     178   120  brown, gr… light       blue         52  male  masculin… Tatooine Human  
## 7 Beru Whitesun 165    75  brown      light       blue         47  femal… feminin… Tatooine Human  
## 8 R5-D4          97     32 <NA>       white, red red        NA  none  masculin… Tatooine Droid  
## 9 Biggs Darkli… 183    84  black      light       brown        24  male  masculin… Tatooine Human  
## 10 Obi-Wan Keno… 182    77  auburn, w… fair        blue-gray     57  male  masculin… Stewjon Human  
## # ... with 71 more rows, and 2 more variables: vehicles <list>, starships <list>
```

## 2) dplyr::arrange

`arrange` sorts the data frame based on the variables you supply:

```
starwars |>  
  arrange(birth_year)
```

```
## # A tibble: 87 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex gender homeworld species  
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr> <chr> <chr>  
## 1 Wicket Syst...     88    20  brown      brown      brown          8 male  masculin... Endor  Ewok  
## 2 IG-88            200   140 none       metal      red             15 none  masculin... <NA>  Droid  
## 3 Luke Skywalker... 172    77  blond      fair       blue            19 male  masculin... Tatooine Human  
## 4 Leia Organa       150    49  brown      light      brown            19 female feminin... Alderaan Human  
## 5 Wedge Antill...   170    77  brown      fair       hazel           21 male  masculin... Corellia Human  
## 6 Plo Koon          188    80  none       orange     black            22 male  masculin... Dorin  Kel  
## 7 Biggs Darkli...   183    84  black      light      brown            24 male  masculin... Tatooine Human  
## 8 Han Solo          180    80  brown      fair       brown            29 male  masculin... Corellia Human  
## 9 Lando Calris...   177    79  black      dark       brown           31 male  masculin... Socorro Human  
## 10 Boba Fett        183   78.2 black     fair       brown          31.5 male  masculin... Kamino Human  
## # ... with 77 more rows, and 2 more variables: vehicles <list>, starships <list>
```

# 1) dplyr::arrange

Why arrange?

# 1) dplyr::arrange

Why arrange?

Arrange sorts your data

# 1) dplyr::arrange

Why arrange?

Arrange sorts your data

This makes it easy to check and see patterns in the data

## 2) dplyr::arrange

We can also arrange items in descending order using `arrange(desc())`

```
starwars |>  
  arrange(desc(birth_year))
```

```
## # A tibble: 87 × 14  
##   name      height  mass hair_color skin_color eye_color birth_year sex gender homeworld species  
##   <chr>     <int>  <dbl> <chr>       <chr>       <chr>       <dbl> <chr> <chr> <chr>       <chr>  
## 1 Yoda        66    17  white       green       brown        896 male  masculin... <NA>       Yoda  
## 2 Jabba Desili... 175  1358 <NA>       green-tan... orange       600 herm... masculin... Nal Hutta  Hutt  
## 3 Chewbacca    228   112 brown      unknown      blue         200 male  masculin... Kashyyyk Wookiee  
## 4 C-3PO        167    75 <NA>       gold        yellow       112 none  masculin... Tatooine Droid  
## 5 Dooku        193    80 white      fair        brown        102 male  masculin... Serenno Human  
## 6 Qui-Gon Jinn 193    89 brown      fair        blue         92 male  masculin... <NA>       Human  
## 7 Ki-Adi-Mundi 198    82 white      pale        yellow       92 male  masculin... Cerean    Cerean  
## 8 Finis Valorum 170    NA blond     fair        blue         91 male  masculin... Coruscant Human  
## 9 Palpatine    170    75 grey       pale        yellow       82 male  masculin... Naboo     Human  
## 10 Cliegg Lars 183    NA brown     fair        blue         82 male  masculin... Tatooine Human  
## # ... with 77 more rows, and 2 more variables: vehicles <list>, starships <list>
```

### 3) dplyr::select

Why select?

### 3) dplyr::select

Why select?

Select lets you choose columns to keep or drop

### 3) dplyr::select

Why select?

Select lets you choose columns to keep or drop

This means you are essentially getting rid of variables, likely ones you do not need

### 3) dplyr::select

Why select?

Select lets you choose columns to keep or drop

This means you are essentially getting rid of variables, likely ones you do not need

e.g. if you used an emissions rate and marginal damage per unit of pollution variable to construct the marginal damage of output, you may not need the first two variables any more

### 3) dplyr::select

Use commas to select multiple columns out of a data frame, deselect a column with "-", select across multiple columns with "first:last":

```
starwars |>  
  select(name:skin_color, species, -height)
```

```
## # A tibble: 87 × 5  
##   name           mass hair_color   skin_color species  
##   <chr>        <dbl> <chr>       <chr>      <chr>  
## 1 Luke Skywalker     77  blond       fair       Human  
## 2 C-3PO              75  <NA>        gold       Droid  
## 3 R2-D2              32  <NA>        white, blue Droid  
## 4 Darth Vader        136 none         white      Human  
## 5 Leia Organa         49  brown        light      Human  
## 6 Owen Lars          120 brown, grey  light      Human  
## 7 Beru Whitesun lars  75  brown        light      Human  
## 8 R5-D4              32  <NA>        white, red Droid  
## 9 Biggs Darklighter   84  black        light      Human  
## 10 Obi-Wan Kenobi    77  auburn, white fair      Human
```

### 3) dplyr::select

You can also rename your selected variables in place

```
starwars |>  
  select(alias = name, crib = homeworld)
```

```
## # A tibble: 87 × 2  
##   alias      crib  
##   <chr>     <chr>  
## 1 Luke Skywalker Tatooine  
## 2 C-3PO        Tatooine  
## 3 R2-D2        Naboo  
## 4 Darth Vader Tatooine  
## 5 Leia Organa Alderaan  
## 6 Owen Lars    Tatooine  
## 7 Beru Whitesun lars Tatooine  
## 8 R5-D4        Tatooine  
## 9 Biggs Darklighter Tatooine  
## 10 Obi-Wan Kenobi Stewjon  
## # ... with 77 more rows
```

### 3) dplyr::select

If you just want to rename columns without subsetting them, you can use

rename:

```
starwars |>  
  rename(alias = name, crib = homeworld)
```

```
## # A tibble: 87 × 14  
##   alias     height   mass hair_color skin_color eye_color birth_year sex gender crib species films  
##   <chr>      <int>  <dbl> <chr>       <chr>       <chr>        <dbl> <chr> <chr> <chr> <chr> <lis>  
## 1 Luke Skywalker 172     77  blond      fair       blue          19  male  masculin Tato... Human <chr>  
## 2 C-3PO            167     75 <NA>       gold       yellow        112  none  masculin Tato... Droid <chr>  
## 3 R2-D2             96      32 <NA>       white, bl... red           33  none  masculin Naboo Droid <chr>  
## 4 Darth Vader    202     136  none       white       yellow        41.9 male  masculin Tato... Human <chr>  
## 5 Leia Organa    150      49  brown      light       brown         19  fema... feminin Alde... Human <chr>  
## 6 Owen Lars     178     120  brown, gr... light       blue          52  male  masculin Tato... Human <chr>  
## 7 Beru Whitesun 165      75  brown      light       blue          47  fema... feminin Tato... Human <chr>  
## 8 R5-D4            97      32 <NA>       white, red red           NA  none  masculin Tato... Droid <chr>  
## 9 Biggs Darko    183     84  black      light       brown         24  male  masculin Tato... Human <chr>  
## 10 Obi-Wan Kenobi 182     77  auburn    w... fair       blue-gray       57  male  masculin Stew... Human <chr>
```

### 3) dplyr::select cont.

The `select(contains(PATTERN))` option provides a nice shortcut in relevant cases.

```
starwars |>  
  select(name, contains("color"))
```

```
## # A tibble: 87 × 4  
##   name          hair_color    skin_color eye_color  
##   <chr>         <chr>        <chr>      <chr>  
## 1 Luke Skywalker  blond       fair        blue  
## 2 C-3PO           <NA>        gold        yellow  
## 3 R2-D2           <NA>        white, blue red  
## 4 Darth Vader    none        white        yellow  
## 5 Leia Organa    brown       light        brown  
## 6 Owen Lars      brown, grey light        blue  
## 7 Beru Whitesun lars brown       light        blue  
## 8 R5-D4           <NA>        white, red red  
## 9 Biggs Darklighter black       light        brown  
## 10 Obi-Wan Kenobi auburn     white, fair blue-gray
```

### 3) dplyr::select

The `select(..., everything())` option is another useful shortcut if you only want to bring some variable(s) to the "front" of a data frame

```
starwars |>  
  select(species, homeworld, everything()) |>  
  head(5)
```

```
## # A tibble: 5 × 14  
##   species homeworld name   height  mass hair_color skin_color eye_color birth_year sex   gender films  
##   <chr>     <chr>    <chr>   <int>  <dbl>   <chr>       <chr>       <chr>           <dbl> <chr> <chr>   <list>  
## 1 Human     Tatooine  Luke...     172     77  blond      fair        blue            19   male  masculin... <chr>  
## 2 Droid      Tatooine  C-3P0     167     75 <NA>       gold        yellow          112  none  masculin... <chr>  
## 3 Droid      Naboo     R2-D2      96      32 <NA>      white, bl... red             33   none  masculin... <chr>  
## 4 Human     Tatooine  Dart...     202    136  none       white        yellow          41.9 male  masculin... <chr>  
## 5 Human     Alderaan  Leia...     150     49  brown      light        brown            19   fema... feminin... <chr>  
## # ... with 1 more variable: starships <list>
```

### 3) dplyr::select

You can also use `relocate` to do the same thing

```
starwars |>  
  relocate(species, homeworld) |>  
  head(5)
```

```
## # A tibble: 5 × 14  
##   species homeworld name   height  mass hair_color skin_color eye_color birth_year sex   gender films  
##   <chr>     <chr>    <chr>   <int> <dbl>   <chr>       <chr>       <chr>           <dbl> <chr> <chr>   <list>  
## 1 Human     Tatooine  Luke...     172     77  blond      fair        blue            19   male  masculin... <chr>  
## 2 Droid      Tatooine  C-3PO      167     75 <NA>       gold        yellow          112  none  masculin... <chr>  
## 3 Droid      Naboo     R2-D2      96      32 <NA>       white, bl... red             33   none  masculin... <chr>  
## 4 Human     Tatooine  Dart...     202    136  none       white        yellow          41.9 male  masculin... <chr>  
## 5 Human     Alderaan  Leia...     150     49  brown      light        brown            19   fema... feminin... <chr>  
## # ... with 1 more variable: starships <list>
```

## 4) dplyr::mutate

Why mutate?

## 4) dplyr::mutate

Why mutate?

You may need to create new variables

## 4) dplyr::mutate

Why mutate?

You may need to create new variables

e.g. if I give you nominal house prices and the rate of house price inflation, you need to combine these two things to make a new **real house price** variable

## 4) dplyr::mutate

You can create new columns from scratch as transformations of existing columns:

```
starwars |>  
  select(name, birth_year) |>  
  mutate(dog_years = birth_year * 7) |>  
  mutate(comment = paste0(name, " is ", dog_years, " in dog years.))
```

```
## # A tibble: 87 × 4  
##   name           birth_year  dog_years comment  
##   <chr>          <dbl>      <dbl> <chr>  
## 1 Luke Skywalker     19        133  Luke Skywalker is 133 in dog years.  
## 2 C-3PO              112       784  C-3PO is 784 in dog years.  
## 3 R2-D2              33        231  R2-D2 is 231 in dog years.  
## 4 Darth Vader        41.9      293. Darth Vader is 293.3 in dog years.  
## 5 Leia Organa         19        133  Leia Organa is 133 in dog years.  
## 6 Owen Lars            52       364  Owen Lars is 364 in dog years.  
## 7 Beru Whitesun lars    47       329  Beru Whitesun lars is 329 in dog years.  
## 8 R5-D4              NA        NA   R5-D4 is NA in dog years.
```

## 4) dplyr::mutate

Note: `mutate` creates variables in order, so you can chain multiple mutates in a single call

```
starwars |>
  select(name, birth_year) |>
  mutate(
    dog_years = birth_year * 7, ## Separate with a comma
    comment = paste0(name, " is ", dog_years, " in dog years.")
  )
```

```
## # A tibble: 87 × 4
##   name           birth_year  dog_years comment
##   <chr>          <dbl>      <dbl> <chr>
## 1 Luke Skywalker     19        133  Luke Skywalker is 133 in dog years.
## 2 C-3PO              112       784  C-3PO is 784 in dog years.
## 3 R2-D2              33        231  R2-D2 is 231 in dog years.
## 4 Darth Vader        41.9      293. Darth Vader is 293.3 in dog years.
## 5 Leia Organa         19        133  Leia Organa is 133 in dog years.
## 6 Owen Lars           52       364  Owen Lars is 364 in dog years.
```

## 4) dplyr::mutate

Boolean, logical and conditional operators all work well with `mutate` too:

```
starwars |>
  select(name, height) |>
  filter(name %in% c("Luke Skywalker", "Anakin Skywalker")) |>
  mutate(tall1 = height > 180) |> # TRUE or FALSE
  mutate(tall2 = ifelse(height > 180, "Tall", "Short")) ## Same effect, but can choose labels
```

```
## # A tibble: 2 × 4
##   name           height tall1 tall2
##   <chr>          <int> <lgl> <chr>
## 1 Luke Skywalker     172 FALSE Short
## 2 Anakin Skywalker    188 TRUE  Tall
```

## 4) dplyr::mutate

Lastly, combining `mutate` with `across` allows you to easily work on a subset of variables:

```
starwars |>
  select(name:eye_color) |>
  mutate(across(where(is.character), toupper)) |> # Take all character variables, uppercase them
head(5)
```

```
## # A tibble: 5 × 6
##   name           height  mass hair_color skin_color eye_color
##   <chr>        <int> <dbl> <chr>      <chr>      <chr>
## 1 LUKE SKYWALKER     172    77 BLOND      FAIR       BLUE
## 2 C-3PO              167    75 <NA>       GOLD       YELLOW
## 3 R2-D2               96    32 <NA>      WHITE, BLUE RED
## 4 DARTH VADER        202   136 NONE       WHITE      YELLOW
## 5 LEIA ORGANA         150    49 BROWN     LIGHT      BROWN
```

## 5) dplyr::summarise

Why summarise?

## 5) dplyr::summarise

Why summarise?

Often we want to get summary statistics or *collapse* our data

## 5) dplyr::summarise

Why summarise?

Often we want to get summary statistics or *collapse* our data

e.g. if I gave you a data set of each individual's marginal damage in the US, we may want to aggregate up to county-level marginal damages

# 5) dplyr::summarise

Summarising useful in combination with the `group_by` command

```
starwars |>  
  group_by(species, gender) |> # for each species-gender combo  
  summarise(mean_height = mean(height, na.rm = TRUE)) # calculate the mean height
```

## `summarise()` has grouped output by 'species'. You can override using the ` `.groups` argument.

```
## # A tibble: 42 × 3  
## # Groups:   species [38]  
##   species   gender   mean_height  
##   <chr>     <chr>        <dbl>  
## 1 Aleena    masculine      79  
## 2 Besalisk  masculine     198  
## 3 Cerean    masculine     198  
## 4 Chagrian  masculine     196  
## 5 Clawdite  feminine      168  
## 6 Droid     feminine       96  
## 7 Droid     masculine     140
```

## 5) dplyr::summarise

Note that including "na.rm = TRUE" is usually a good idea with summarise functions, it keeps NAs from propagating to the end result

```
## Probably not what we want
starwars |>
  summarise(mean_height = mean(height))
```

```
## # A tibble: 1 × 1
##   mean_height
##       <dbl>
## 1        NA
```

# 5) dplyr::summarise

We can also use `across` within summarise:

```
starwars |>
  group_by(species) |> # for each species
  summarise(across(where(is.numeric), mean, na.rm = T)) |> # take the mean of all numeric variables
  head(5)
```

```
## # A tibble: 5 × 4
##   species    height   mass birth_year
##   <chr>      <dbl>   <dbl>     <dbl>
## 1 Aleena       79     15       NaN
## 2 Besalisk     198    102       NaN
## 3 Cerean       198     82       92
## 4 Chagrian     196     NaN       NaN
## 5 Clawdite     168     55       NaN
```

# Other dplyr goodies

`group_by` and `ungroup`: For (un)grouping

- Particularly useful with the `summarise` and `mutate` commands

# Other dplyr goodies

`group_by` and `ungroup`: For (un)grouping

- Particularly useful with the `summarise` and `mutate` commands

`slice`: Subset rows by position rather than filtering by values

- E.g. `starwars |> slice(c(1, 5))`

# Other dplyr goodies

`pull`: Extract a column from as a data frame as a vector or scalar

- E.g. `starwars |> filter(gender=="female") |> pull(height)`

# Other dplyr goodies

`pull`: Extract a column from as a data frame as a vector or scalar

- E.g. `starwars |> filter(gender=="female") |> pull(height)`

`count` and `distinct`: Number and isolate unique observations

- E.g. `starwars |> count(species)`, or `starwars |> distinct(species)`

# Other dplyr goodies

There are also a whole class of **window functions** for getting leads and lags, percentiles, cumulative sums, etc.

- See `vignette("window-functions")`.

## dplyr::xxxx\_join

The last set of commands we need are the `join` commands

## dplyr::xxxx\_join

The last set of commands we need are the `join` commands

These are the same as `merge` in stata but with a bit more functionality

## dplyr::xxxx\_join

The last set of commands we need are the `join` commands

These are the same as `merge` in stata but with a bit more functionality

Why join?

## dplyr::xxxx\_join

The last set of commands we need are the `join` commands

These are the same as `merge` in stata but with a bit more functionality

Why join?

Suppose we want to understand how pollution affects housing prices

## dplyr::xxxx\_join

The last set of commands we need are the `join` commands

These are the same as `merge` in stata but with a bit more functionality

Why join?

Suppose we want to understand how pollution affects housing prices

You may need to combine data sets: one data sets on house prices in all counties, and another data set on pollution levels in all counties

# dplyr::xxxx\_join

How does joining work?

## dplyr::xxxx\_join

How does joining work?

We need two datasets (e.g. housing prices and pollution)

## dplyr::xxxx\_join

How does joining work?

We need two datasets (e.g. housing prices and pollution)

Each dataset has the same set of **key** variables (e.g. county)

# dplyr::xxxx\_join

How does joining work?

We need two datasets (e.g. housing prices and pollution)

Each dataset has the same set of **key** variables (e.g. county)

When we join, we match up the two datasets on the key, and combine them into one

# dplyr::xxxx\_join

How does joining work?

We need two datasets (e.g. housing prices and pollution)

Each dataset has the same set of **key** variables (e.g. county)

When we join, we match up the two datasets on the key, and combine them into one

In our housing example, each row of the joined dataset would now tell us the house price, the county its in, and the county's pollution

# dplyr::xxxx\_join

We merge data with **join operations**:

- `inner_join(df1, df2)`
- `left_join(df1, df2)`
- `right_join(df1, df2)`
- `full_join(df1, df2)`

(You can visualize the operations [here](#))

# dplyr::xxxx\_join

Lets use the data that comes with the the `nycflights13` package.

```
library(nycflights13)
flights

## # A tibble: 336,776 × 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier flight
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>   <dbl>
## 1 2013     1     1      517            515       2        830           819      11  UA     1.00
## 2 2013     1     1      533            529       4        850           830      20  UA     1.00
## 3 2013     1     1      542            540       2        923           850      33  AA     1.00
## 4 2013     1     1      544            545      -1       1004          1022     -18  B6     1.00
## 5 2013     1     1      554            600      -6       812           837     -25  DL     1.00
## 6 2013     1     1      554            558      -4       740           728      12  UA     1.00
## 7 2013     1     1      555            600      -5       913           854      19  B6     1.00
## 8 2013     1     1      557            600      -3       709           723     -14  EV     1.00
## 9 2013     1     1      557            600      -3       838           846      -8  B6     1.00
## 10 2013    1     1      558            600      -2       753           745      8  AA     1.00
## # ... with 336,766 more rows, and 8 more variables: tailnum <chr>, origin <chr>, dest <chr>, arr_time <dbl>, dep_time <dbl>, sched_arr_time <dbl>, sched_dep_time <dbl>, arr_delay <dbl>
```

# dplyr::xxxx\_join

planes

```
## # A tibble: 3,322 × 9
##   tailnum year type          manufacturer    model engines seats speed engine
##   <chr>   <int> <chr>        <chr>       <chr>     <int> <int> <int> <chr>
## 1 N10156  2004 Fixed wing multi engine EMBRAER EMB-145XR      2     55   NA Turbo-fan
## 2 N102UW   1998 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 3 N103US   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 4 N104UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 5 N10575   2002 Fixed wing multi engine EMBRAER EMB-145LR      2     55   NA Turbo-fan
## 6 N105UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 7 N107US   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 8 N108UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 9 N109UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## 10 N110UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214      2    182   NA Turbo-fan
## # ... with 3,312 more rows
```

# Joining operations

Let's perform a left join on the flights and planes datasets

- Note: I'm going to subset columns after the join, but only to keep text on the slide

# Joining operations

Let's perform a left join on the flights and planes datasets

- Note: I'm going to subset columns after the join, but only to keep text on the slide

```
left_join(flights, planes) |>  
  select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, type, model)
```

```
## Joining, by = c("year", "tailnum")
```

```
## # A tibble: 336,776 × 10  
##   year month   day dep_time arr_time carrier flight tailnum type  model  
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>    <chr> <chr>  
## 1 2013     1     1      517      830  UA        1545 N14228  <NA>  <NA>  
## 2 2013     1     1      533      850  UA        1714 N24211  <NA>  <NA>  
## 3 2013     1     1      542      923  AA        1141 N619AA  <NA>  <NA>  
## 4 2013     1     1      544     1004  B6        725  N804JB  <NA>  <NA>  
## 5 2013     1     1      554      812  DL        161  NCC6DN  <NA>  <NA>
```

# Joining operations

Note that dplyr made a reasonable guess about which columns to join on (i.e. columns that share the same name), and told us what it chose

```
## Joining, by = c("year", "tailnum")
```

There's an obvious problem here: the variable `year` does not have a consistent meaning across our joining datasets

# Joining operations

Note that dplyr made a reasonable guess about which columns to join on (i.e. columns that share the same name), and told us what it chose

```
## Joining, by = c("year", "tailnum")
```

There's an obvious problem here: the variable `year` does not have a consistent meaning across our joining datasets

In one it refers to the *year of flight*, in the other it refers to *year of construction*

Luckily, there's an easy way to avoid this problem: try `?dplyr::join`

# Joining operations

You just need to be more explicit in your join call by using the `by =` argument

```
left_join(  
  flights,  
  planes |> rename(year_built = year), ## Not necessary w/ below line, but helpful  
  by = "tailnum" ## Be specific about the joining column  
 ) |>  
 select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, year_built, type, model)  
 head(3) ## Just to save vertical space on the slide
```

```
## # A tibble: 3 × 11  
##   year month   day dep_time arr_time carrier flight tailnum year_built type  
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>     <int> <chr>  
## 1  2013     1     1      517      830  UA        1545 N14228    1999 Fixed wing multi engine 737-  
## 2  2013     1     1      533      850  UA        1714 N24211    1998 Fixed wing multi engine 737-  
## 3  2013     1     1      542      923  AA        1141 N619AA    1990 Fixed wing multi engine 757-
```

# Joining operations

Note what happens if we again specify the join column but don't rename the ambiguous year:

```
left_join(flights,
  planes, ## Not renaming "year" to "year_built" this time
  by = "tailnum") |>
  select(contains("year"), month, day, dep_time, arr_time, carrier, flight, tailnum, type, model)
  head(3)

## # A tibble: 3 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type
##   <int>  <int> <int> <int>    <int>    <int> <chr>   <int> <chr>   <chr>
## 1  2013    1999     1     1      517      830  UA       1545 N14228 Fixed wing multi engine 737-824
## 2  2013    1998     1     1      533      850  UA       1714 N24211 Fixed wing multi engine 737-824
## 3  2013    1990     1     1      542      923  AA       1141 N619AA Fixed wing multi engine 757-223
```

# Joining operations

Note what happens if we again specify the join column but don't rename the ambiguous `year`:

```
left_join(flights,
  planes, ## Not renaming "year" to "year_built" this time
  by = "tailnum") |>
  select(contains("year"), month, day, dep_time, arr_time, carrier, flight, tailnum, type, model)
  head(3)

## # A tibble: 3 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type
##   <int>  <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr>
## 1   2013    1999     1     1      517      830  UA        1545 N14228 Fixed wing multi engine 737-824
## 2   2013    1998     1     1      533      850  UA        1714 N24211 Fixed wing multi engine 737-824
## 3   2013    1990     1     1      542      923  AA        1141 N619AA Fixed wing multi engine 757-223
```

Make sure you know what "year.x" and "year.y" are

# tidyr

---

# Key tidy verbs

1. `pivot_longer`: Pivot wide data into long format (i.e. "melt", "reshape long")
2. `pivot_wider`: Pivot long data into wide format (i.e. "cast", "reshape wide")
3. `separate`: Split one column into multiple columns
4. `unite`: Combine multiple columns into one

# Key tidy verbs

1. `pivot_longer`: Pivot wide data into long format (i.e. "melt", "reshape long")
2. `pivot_wider`: Pivot long data into wide format (i.e. "cast", "reshape wide")
3. `separate`: Split one column into multiple columns
4. `unite`: Combine multiple columns into one

Let's practice these verbs together in class

# 1) tidy::pivot\_longer

```
stocks = data.frame(  
  time = as.Date('2009-01-01') + 0:1,  
  stock_X = rnorm(2, 0, 1),  
  stock_Y = rnorm(2, 0, 2),  
  stock_Z = rnorm(2, 0, 4)  
)  
stocks
```

```
##          time    stock_X    stock_Y  stock_Z  
## 1 2009-01-01 1.10785122  2.3830816 0.234776  
## 2 2009-01-02 0.05259919 -0.3400277 3.121391
```

We have 4 variables, the date and the stocks

How do we get this in tidy form?

# 1) tidyverse::pivot\_longer

```
stocks |> pivot_longer(-time, names_to = "stock", values_to = "price")
```

We need to pivot the stock name variables x, y, z longer

1. Choose non-time variables: -time
2. Decide what variable holds the names: names\_to = "stock"
3. Decide what variable holds the values: values\_to = "price"

# 1) tidyverse::pivot\_longer

```
stocks |> pivot_longer(-time, names_to = "stock", values_to = "price")
```

```
## # A tibble: 6 × 3
##   time      stock    price
##   <date>    <chr>     <dbl>
## 1 2009-01-01 stock_X  1.11
## 2 2009-01-01 stock_Y  2.38
## 3 2009-01-01 stock_Z  0.235
## 4 2009-01-02 stock_X  0.0526
## 5 2009-01-02 stock_Y -0.340
## 6 2009-01-02 stock_Z  3.12
```

# 1) tidyverse::pivot\_longer

Let's quickly save the "tidy" (i.e. long) stocks data frame for use on the next slide

```
tidy_stocks = stocks |>  
  pivot_longer(-time, names_to = "stock", values_to = "price")
```

## 2) tidyverse::pivot\_wider

```
tidy_stocks |> pivot_wider(names_from = stock, values_from = price)
```

```
## # A tibble: 2 × 4
##   time      stock_X stock_Y stock_Z
##   <date>     <dbl>    <dbl>    <dbl>
## 1 2009-01-01  1.11     2.38    0.235
## 2 2009-01-02  0.0526   -0.340   3.12
```

```
tidy_stocks |> pivot_wider(names_from = time, values_from = price)
```

```
## # A tibble: 3 × 3
##   stock   `2009-01-01` `2009-01-02`
##   <chr>     <dbl>       <dbl>
## 1 stock_X     1.11      0.0526
## 2 stock_Y     2.38     -0.340
## 3 stock_Z     0.235     3.12
```

## 2) tidyverse::pivot\_wider

```
tidy_stocks |> pivot_wider(names_from = stock, values_from = price)
```

```
## # A tibble: 2 × 4
##   time      stock_X stock_Y stock_Z
##   <date>     <dbl>    <dbl>    <dbl>
## 1 2009-01-01  1.11     2.38    0.235
## 2 2009-01-02  0.0526   -0.340   3.12
```

```
tidy_stocks |> pivot_wider(names_from = time, values_from = price)
```

```
## # A tibble: 3 × 3
##   stock   `2009-01-01` `2009-01-02`
##   <chr>     <dbl>       <dbl>
## 1 stock_X     1.11      0.0526
## 2 stock_Y     2.38     -0.340
## 3 stock_Z     0.235     3.12
```

Note that the second example has effectively transposed the data

### 3) tidyverse::separate

```
conomists = data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))
conomists
```

```
##           name
## 1      Adam.Smith
## 2  Paul.Samuelson
## 3 Milton.Friedman
```

```
conomists |> separate(name, c("first_name", "last_name"))
```

```
##   first_name last_name
## 1      Adam     Smith
## 2      Paul Samuelson
## 3    Milton  Friedman
```

### 3) tidyverse::separate

```
conomists = data.frame(name = c("Adam.Smith", "Paul.Samuelson", "Milton.Friedman"))
conomists
```

```
##           name
## 1      Adam.Smith
## 2  Paul.Samuelson
## 3 Milton.Friedman
```

```
conomists |> separate(name, c("first_name", "last_name"))
```

```
##   first_name last_name
## 1      Adam     Smith
## 2      Paul Samuelson
## 3    Milton Friedman
```

This command is pretty smart. But to avoid ambiguity, you can also specify the separation character with `separate(..., sep=".")`

## 4) tidyverse

```
gdp = data.frame(  
  yr = rep(2016, times = 4),  
  mnth = rep(1, times = 4),  
  dy = 1:4,  
  gdp = rnorm(4, mean = 100, sd = 2)  
)  
gdp
```

```
##      yr mnth dy      gdp  
## 1 2016     1  1 101.33425  
## 2 2016     1  2 101.55379  
## 3 2016     1  3  95.68140  
## 4 2016     1  4  98.62554
```

## 4) tidyverse

```
## Combine "yr", "mnth", and "dy" into one "date" column
gdp |> unite(date, c("yr", "mnth", "dy"), sep = "-")
```

```
##          date      gdp
## 1 2016-1-1 101.33425
## 2 2016-1-2 101.55379
## 3 2016-1-3  95.68140
## 4 2016-1-4  98.62554
```

## 4) tidyverse::unite

Note that `unite` will automatically create a character variable:

```
gdp_u = gdp |> unite(date, c("yr", "mnth", "dy"), sep = "-") |> as_tibble()  
gdp_u
```

```
## # A tibble: 4 × 2  
##   date      gdp  
##   <chr>     <dbl>  
## 1 2016-1-1 101.  
## 2 2016-1-2 102.  
## 3 2016-1-3  95.7  
## 4 2016-1-4  98.6
```

## 4) tidyverse::unite

Note that `unite` will automatically create a character variable:

```
gdp_u = gdp |> unite(date, c("yr", "mnth", "dy"), sep = "-") |> as_tibble()
gdp_u
```

```
## # A tibble: 4 × 2
##   date      gdp
##   <chr>    <dbl>
## 1 2016-1-1 101.
## 2 2016-1-2 102.
## 3 2016-1-3  95.7
## 4 2016-1-4  98.6
```

If you want to convert it to something else (e.g. date or numeric) then you will need to modify it using `mutate`

## 4) tidyverse

```
library(lubridate)
gdp_u |> mutate(date = ymd(date))
```

```
## # A tibble: 4 × 2
##   date      gdp
##   <date>    <dbl>
## 1 2016-01-01 101.
## 2 2016-01-02 102.
## 3 2016-01-03  95.7
## 4 2016-01-04  98.6
```

# Regression and ordinary least squares

---

# Why?

## Motivation

Let's start with a few **basic, general questions**

# Why?

## Motivation

Let's start with a few **basic, general questions**

1. What is the goal of econometrics?
2. Why do economists (or other people) study or use econometrics?

# Why?

## Motivation

Let's start with a few **basic, general questions**

1. What is the goal of econometrics?
2. Why do economists (or other people) study or use econometrics?

**One simple answer:** Learn about the world using data

# Why? Example

GPA is an output from endowments (ability), and hours studied (inputs), and pollution exposure (externality)

# Why? Example

GPA is an output from endowments (ability), and hours studied (inputs), and pollution exposure (externality)

One might hypothesize a model:  $\text{GPA} = f(I, P, \text{SAT}, H)$

where  $H$  is hours studied,  $P$  is pollution exposure, SAT is SAT score and I is family income

# Why? Example

GPA is an output from endowments (ability), and hours studied (inputs), and pollution exposure (externality)

One might hypothesize a model:  $\text{GPA} = f(I, P, \text{SAT}, H)$

where  $H$  is hours studied,  $P$  is pollution exposure, SAT is SAT score and I is family income

We expect that GPA will rise with some variables, and decrease with others

# Why? Example

GPA is an output from endowments (ability), and hours studied (inputs), and pollution exposure (externality)

One might hypothesize a model:  $\text{GPA} = f(I, P, \text{SAT}, H)$

where  $H$  is hours studied,  $P$  is pollution exposure, SAT is SAT score and I is family income

We expect that GPA will rise with some variables, and decrease with others

But who needs to *expect*?

# Why? Example

GPA is an output from endowments (ability), and hours studied (inputs), and pollution exposure (externality)

One might hypothesize a model:  $\text{GPA} = f(I, P, \text{SAT}, H)$

where  $H$  is hours studied,  $P$  is pollution exposure, SAT is SAT score and I is family income

We expect that GPA will rise with some variables, and decrease with others

But who needs to *expect*?

We can test these hypotheses **using a regression model**

# How?

We can write down a linear regression model of the relationship between GPA and (H, P, SAT, PCT):

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

# How?

We can write down a linear regression model of the relationship between GPA and (H, P, SAT, PCT):

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

The left hand side of the equals sign is our **dependent variable** GPA

# How?

We can write down a linear regression model of the relationship between GPA and (H, P, SAT, PCT):

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

The left hand side of the equals sign is our **dependent variable** GPA

The right hand side of the equals sign contains all of our **independent variables** (I, P, SAT, H), and an error term  $\varepsilon_i$  (described later)

# How?

We can write down a linear regression model of the relationship between GPA and (H, P, SAT, PCT):

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

The left hand side of the equals sign is our **dependent variable** GPA

The right hand side of the equals sign contains all of our **independent variables** (I, P, SAT, H), and an error term  $\varepsilon_i$  (described later)

The subscript  $i$  means that the variable contains the value for some person  $i$  in our dataset where  $i = 1, \dots, N$

# How?

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

We are interested in how pollution  $P$  affects GPA

# How?

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

We are interested in how pollution  $P$  affects GPA

This is given by  $\beta_2$

# How?

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

We are interested in how pollution  $P$  affects GPA

This is given by  $\beta_2$

Notice that  $\beta_2 = \frac{\partial \text{GPA}_i}{\partial P_i}$

# How?

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

We are interested in how pollution  $P$  affects GPA

This is given by  $\beta_2$

Notice that  $\beta_2 = \frac{\partial \text{GPA}_i}{\partial P_i}$

$\beta_2$  tells us how GPA changes, given a 1 unit increase in pollution!

# How?

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

We are interested in how pollution P affects GPA

This is given by  $\beta_2$

Notice that  $\beta_2 = \frac{\partial \text{GPA}_i}{\partial P_i}$

$\beta_2$  tells us how GPA changes, given a 1 unit increase in pollution!

Our goal will be to estimate  $\beta_2$ , we denote estimates with hats:  $\hat{\beta}_2$

# How?

How do we estimate  $\beta_2$ ?

# How?

How do we estimate  $\beta_2$ ?

First, suppose we have a set of estimates for all of our  $\beta$ s, then we can *estimate* the GPA ( $\widehat{GPA}_i$ ) for any given person based on just (I, P, SAT, H):

$$\widehat{GPA}_i = \hat{\beta}_0 + \hat{\beta}_1 I_i + \hat{\beta}_2 P_i + \hat{\beta}_3 \text{SAT}_i + \hat{\beta}_4 H_i$$

# How?

We estimate the  $\beta$ s with **linear regression**, specifically ordinary least squares

**Ordinary least squares:** choose all the  $\beta$ s so that the sum of squared errors between the *real* GPAs and model-estimated GPAs are minimized:

$$SSE = \sum_{i=1}^N (GPA_i - \widehat{GPA}_i)^2$$

# How?

We estimate the  $\beta$ s with **linear regression**, specifically ordinary least squares

**Ordinary least squares:** choose all the  $\beta$ s so that the sum of squared errors between the *real* GPAs and model-estimated GPAs are minimized:

$$SSE = \sum_{i=1}^N (GPA_i - \widehat{GPA}_i)^2$$

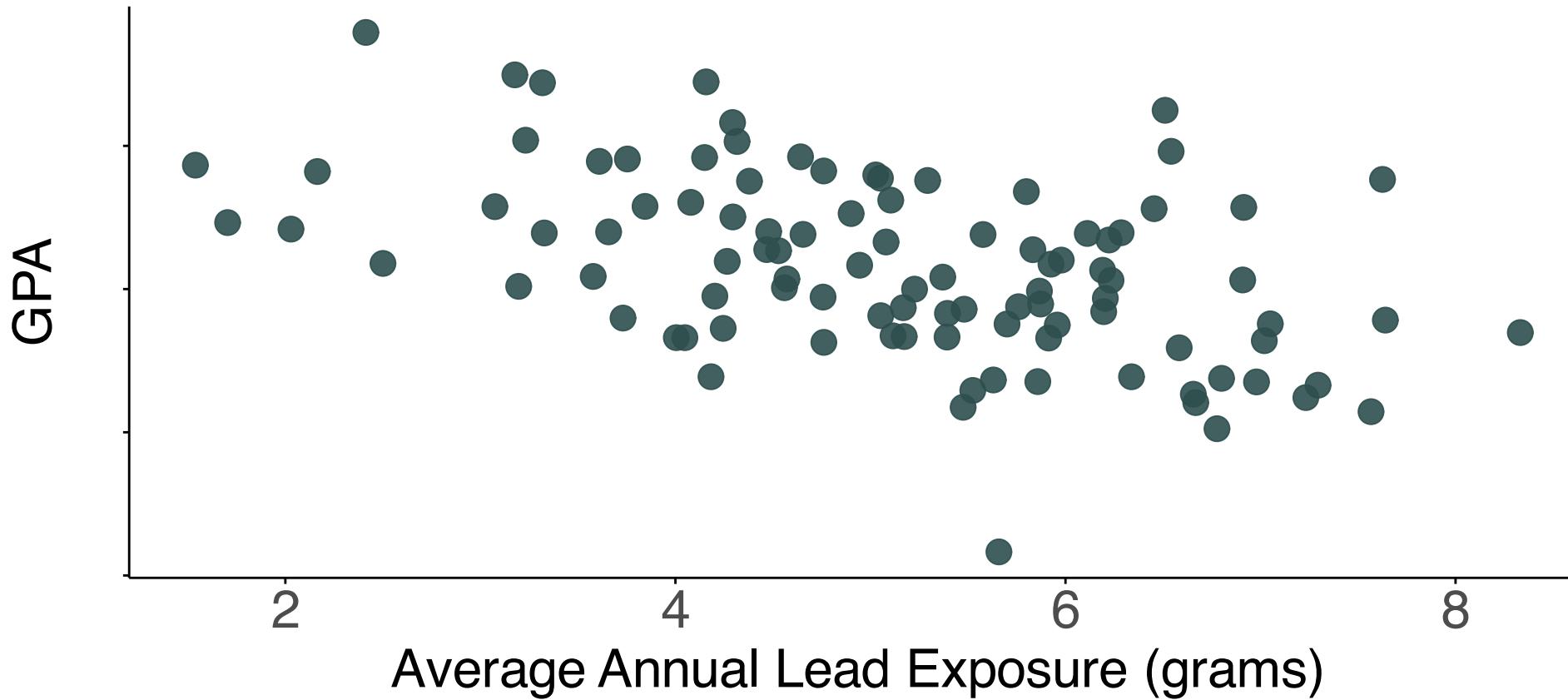
Choosing the  $\beta$ s in this fashion gives us the best-fit line through the data

# How?

# Simple example

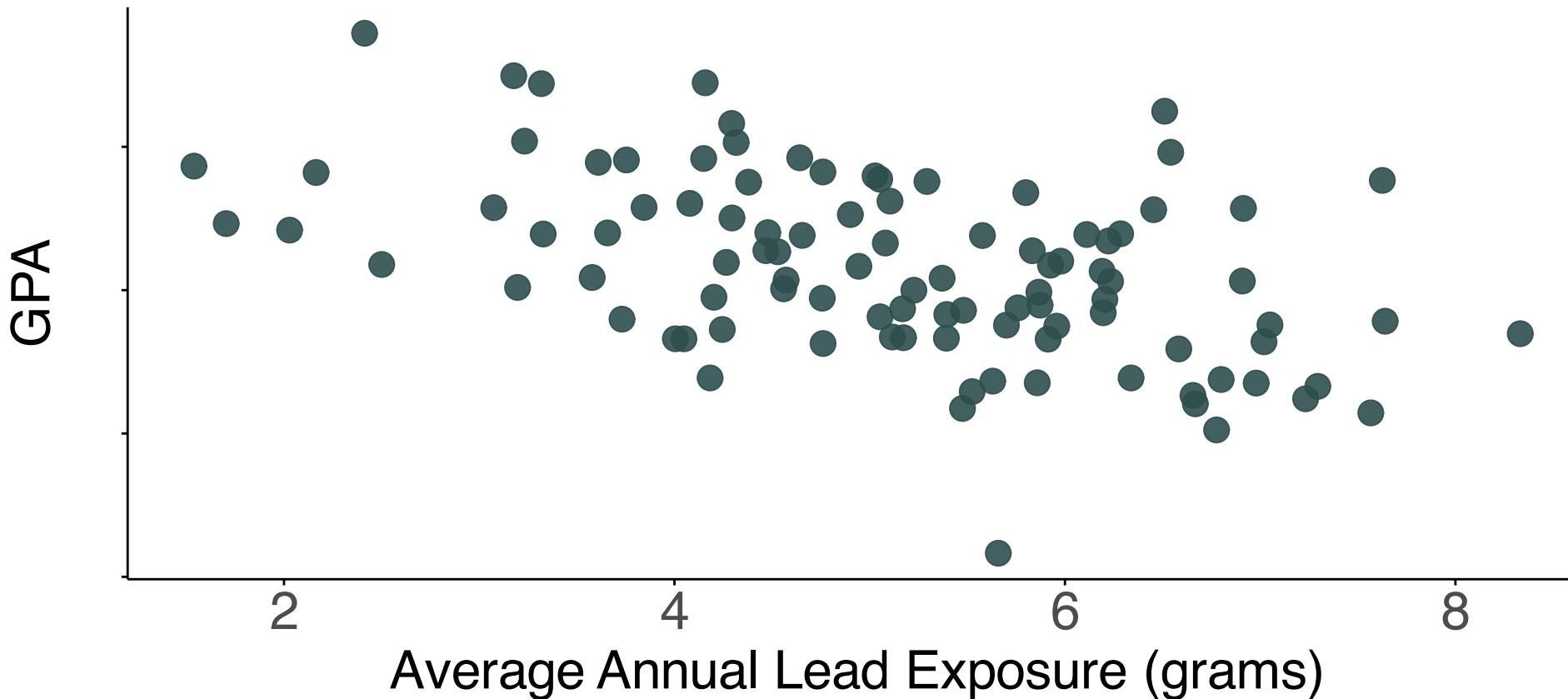
Suppose we were only looking at GPA and pollution (lead/Pb):

$$GPA_i = \beta_0 + \beta_1 P_i + \varepsilon_i$$



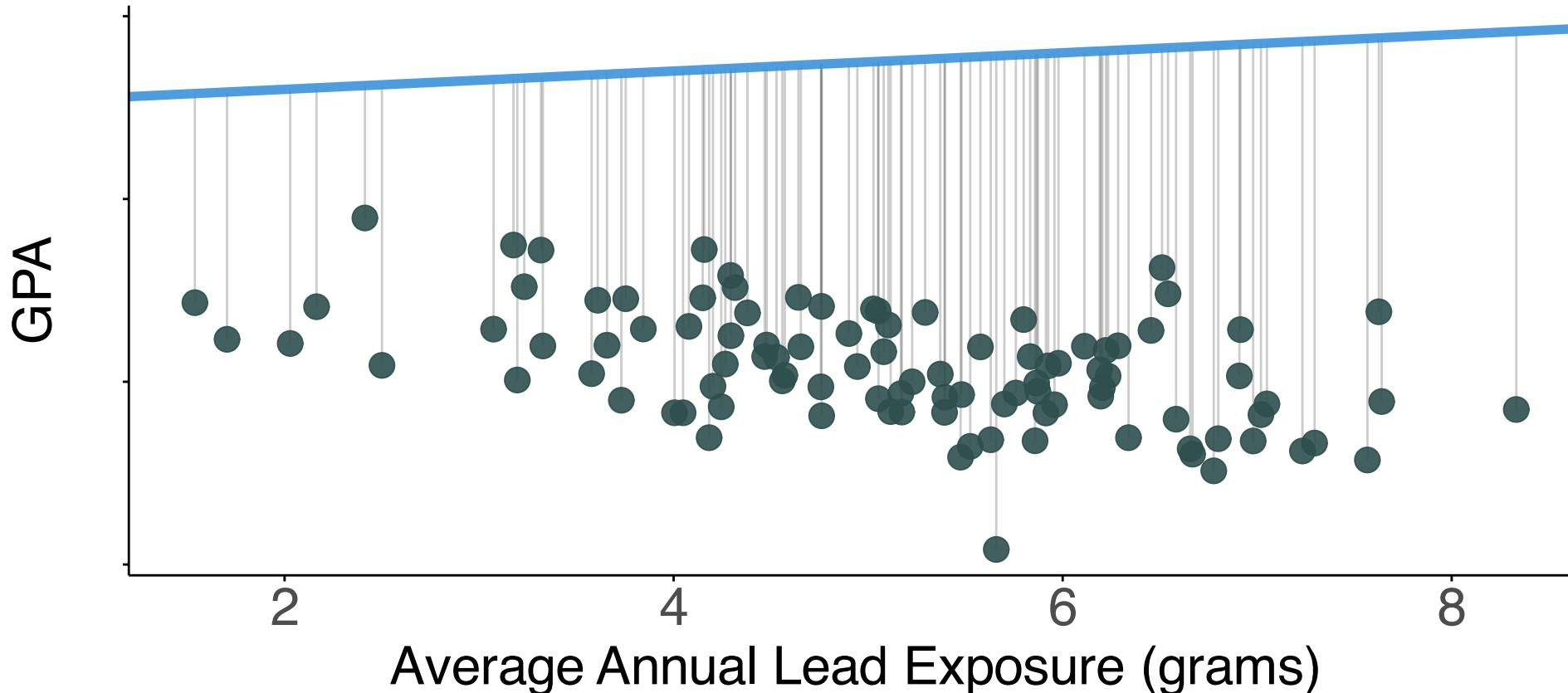
# Simple example

For any line  $(GPA_i = \hat{\beta}_0 + \hat{\beta}_1 P_i)$



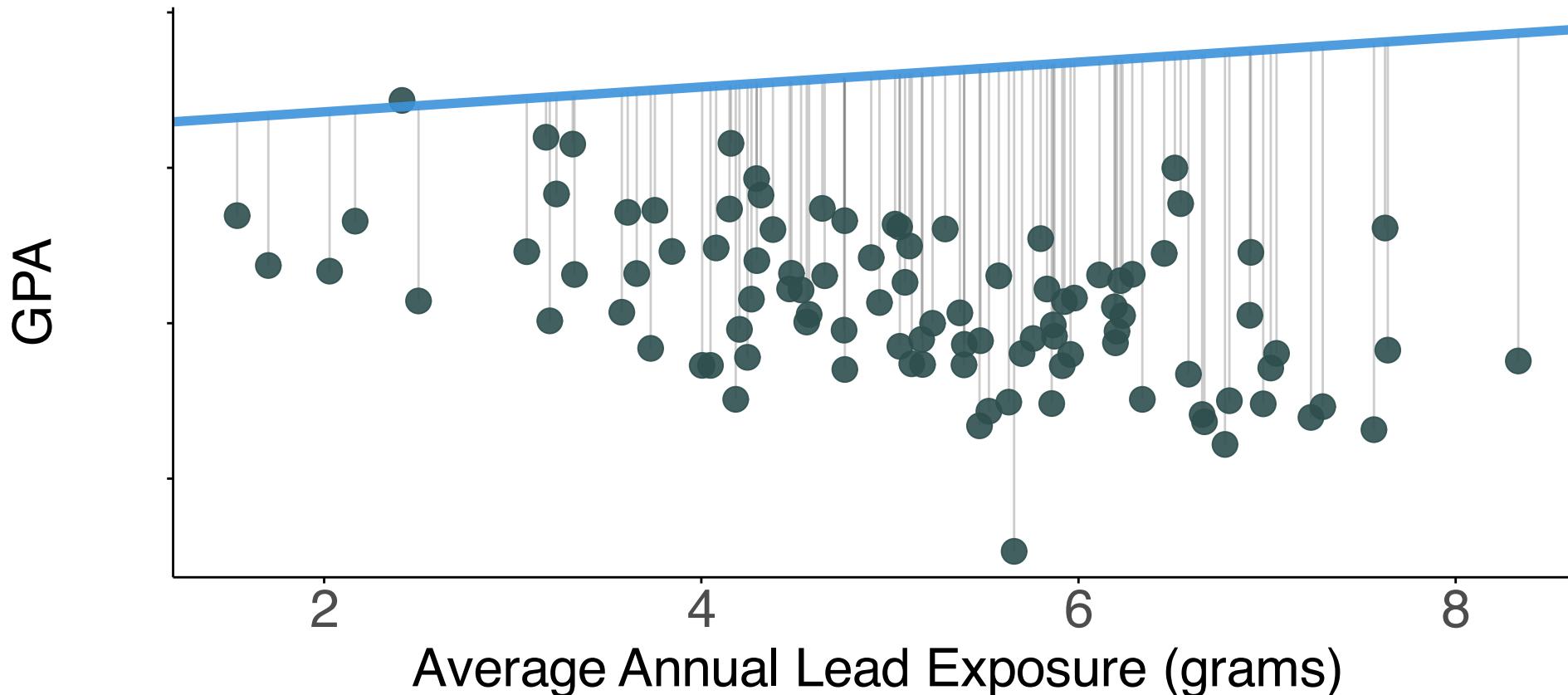
# Simple example

For any line  $(GPA_i = \hat{\beta}_0 + \hat{\beta}_1 P_i)$ , we calculate errors:  $e_i = GPA_i - \hat{GPA}_i$



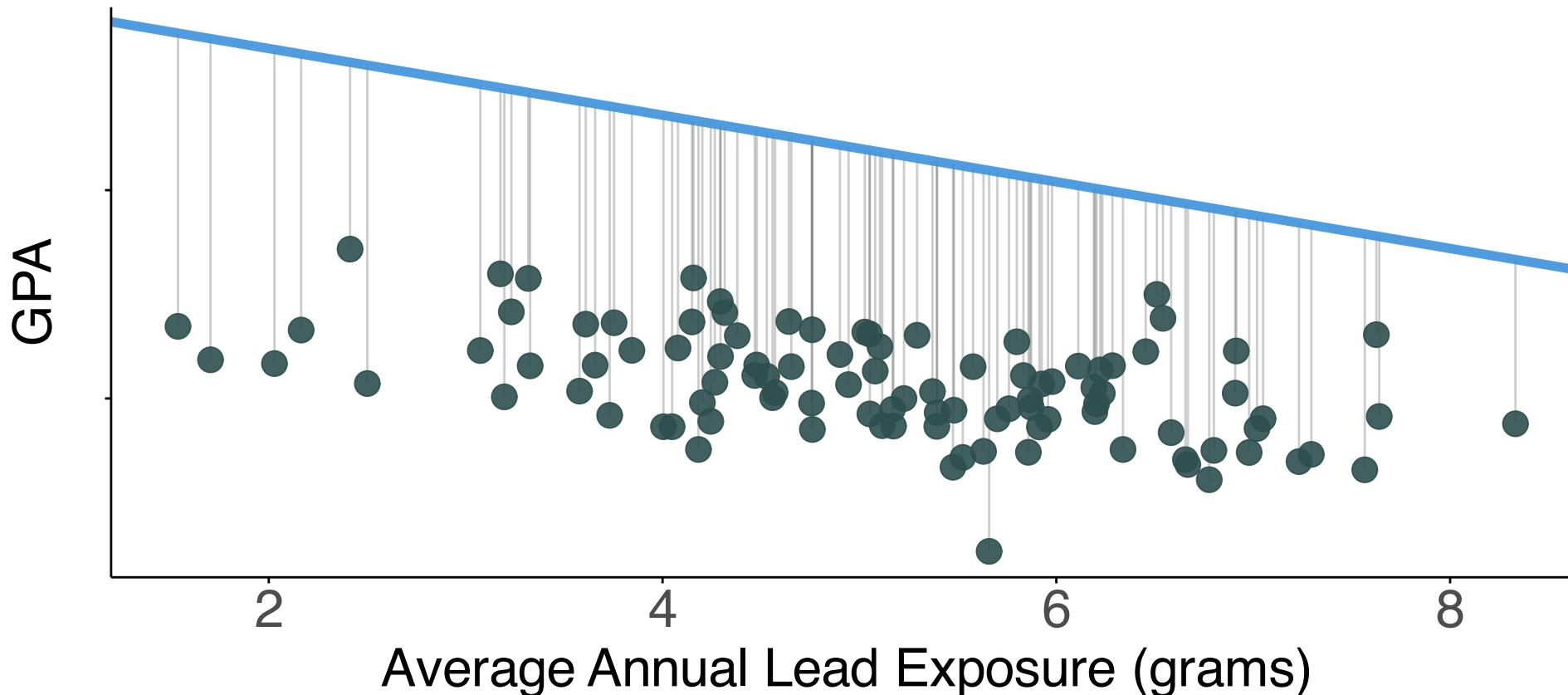
# Simple example

For any line  $(GPA_i = \hat{\beta}_0 + \hat{\beta}_1 P_i)$ , we calculate errors:  $e_i = GPA_i - \hat{GPA}_i$



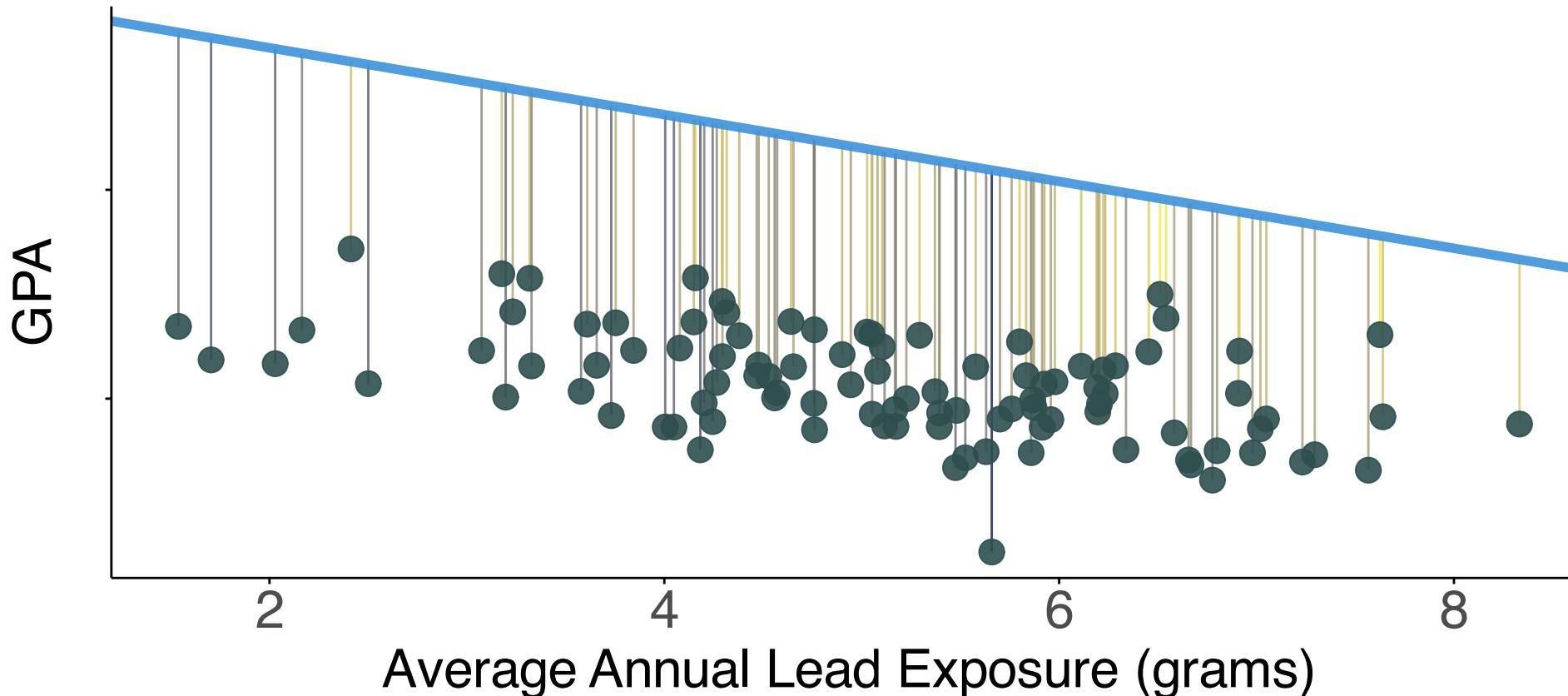
# Simple example

For any line  $(GPA_i = \hat{\beta}_0 + \hat{\beta}_1 P_i)$ , we calculate errors:  $e_i = GPA_i - \hat{GPA}_i$



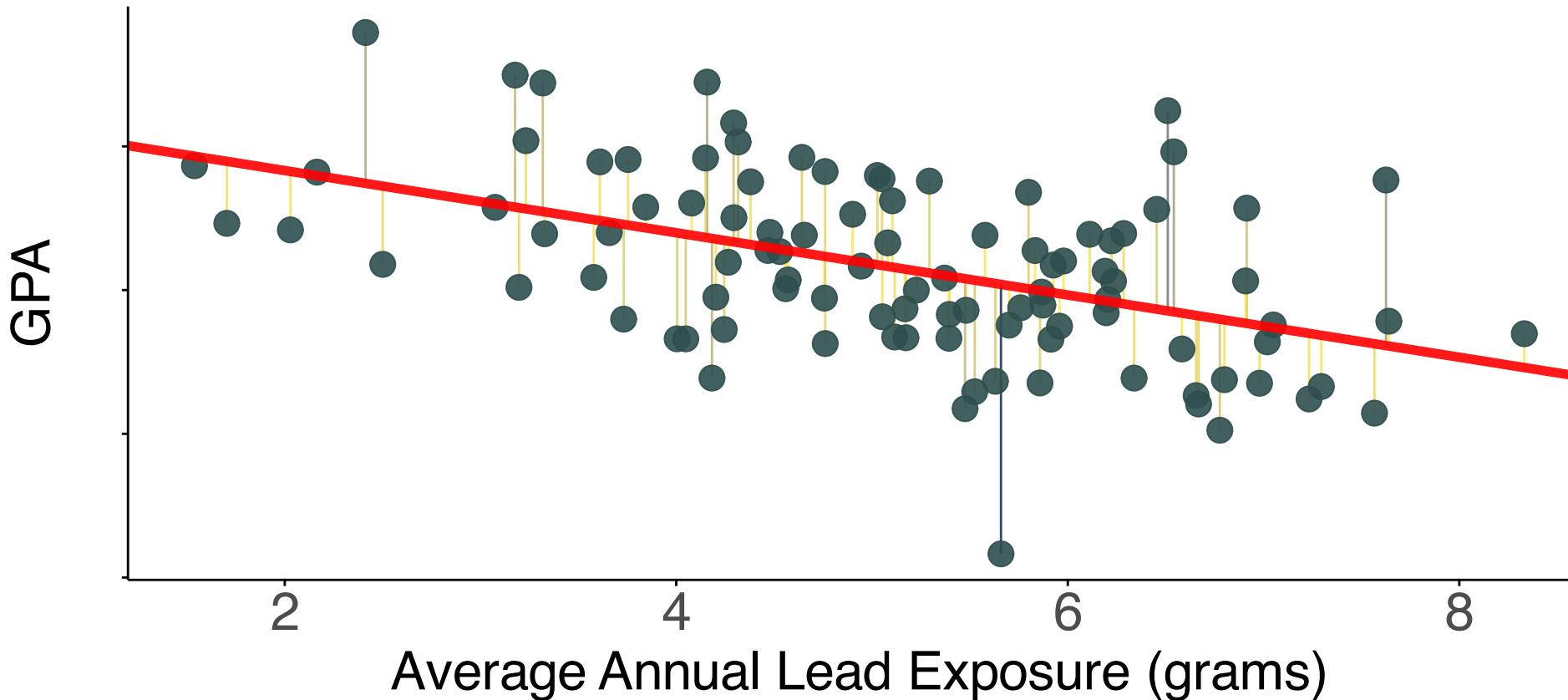
# Simple example

SSE squares the errors ( $\sum e_i^2$ ): bigger errors get bigger penalties



# Simple example

The OLS estimate is the combination of  $\hat{\beta}_0$  and  $\hat{\beta}_1$  that minimize SSE



# OLS error term

So OLS is just the best-fit line through your data

# OLS error term

So OLS is just the best-fit line through your data

# OLS error term

So OLS is just the best-fit line through your data

Why?

# OLS error term

So OLS is just the best-fit line through your data

Why?

Our model isn't perfect, the people in our dataset (i.e. our sample) may not perfectly match up to the entire population of people

# OLS error term

There's **a lot** of other stuff that determines GPAs!

# OLS error term

There's **a lot** of other stuff that determines GPAs!

We jam all that stuff into error term  $\varepsilon_i$ :

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

# OLS error term

There's **a lot** of other stuff that determines GPAs!

We jam all that stuff into error term  $\varepsilon_i$ :

$$\text{GPA}_i = \beta_0 + \beta_1 I_i + \beta_2 P_i + \beta_3 \text{SAT}_i + \beta_4 H_i + \varepsilon_i$$

So  $\varepsilon_i$  contains all the determinants of GPA that we aren't explicitly addressing in our model

# OLS properties

OLS has one **very** nice property relevant for this class:

# OLS properties

OLS has one **very** nice property relevant for this class:

**Unbiasedness:**  $E[\hat{\beta}] = \beta$

# OLS properties

**Unbiasedness:**  $E[\hat{\beta}] = \beta$

On average, our estimate  $\hat{\beta}$  exactly equals the **true**  $\beta$

# OLS properties

**Unbiasedness:**  $E[\hat{\beta}] = \beta$

On average, our estimate  $\hat{\beta}$  exactly equals the **true**  $\beta$

The key is **on average**: we are estimating our model using only some sample of the data

# OLS properties

**Unbiasedness:**  $E[\hat{\beta}] = \beta$

On average, our estimate  $\hat{\beta}$  exactly equals the **true**  $\beta$

The key is **on average**: we are estimating our model using only some sample of the data

The estimated  $\beta$  won't exactly be right for the entire population, but on average, we expect it to match

# OLS properties

**Unbiasedness:**  $E[\hat{\beta}] = \beta$

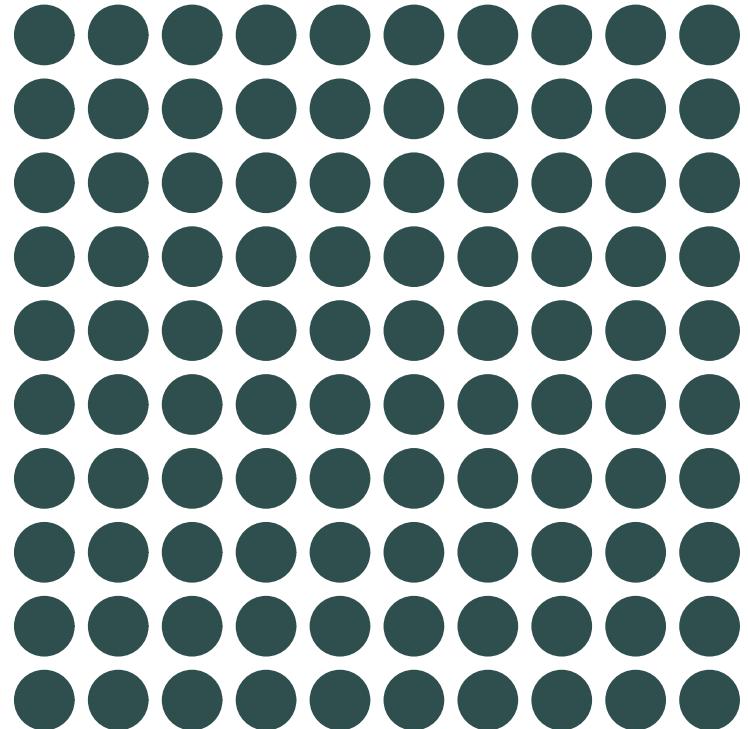
On average, our estimate  $\hat{\beta}$  exactly equals the **true**  $\beta$

The key is **on average**: we are estimating our model using only some sample of the data

The estimated  $\beta$  won't exactly be right for the entire population, but on average, we expect it to match

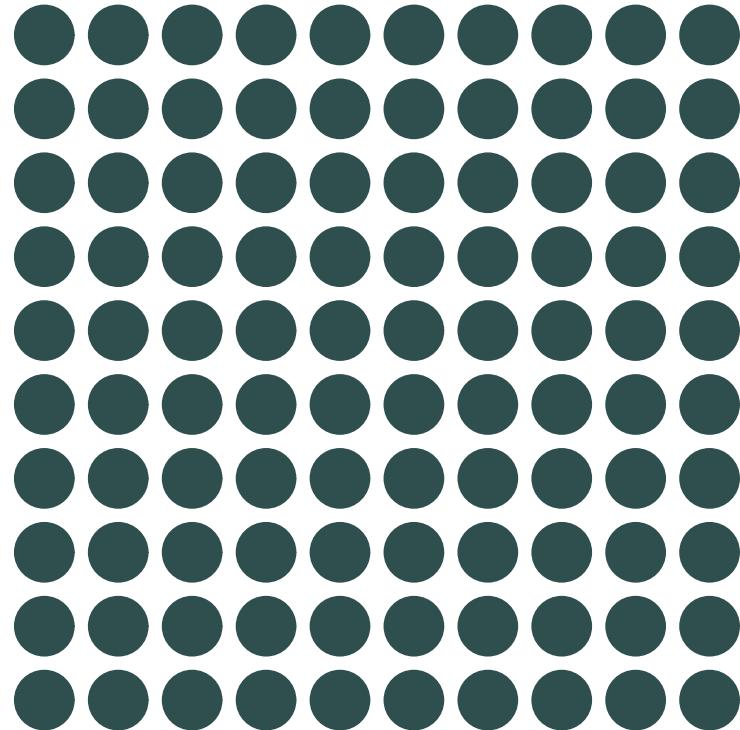
Let's see in an example where we only have a subsample of the full population of data

# OLS properties

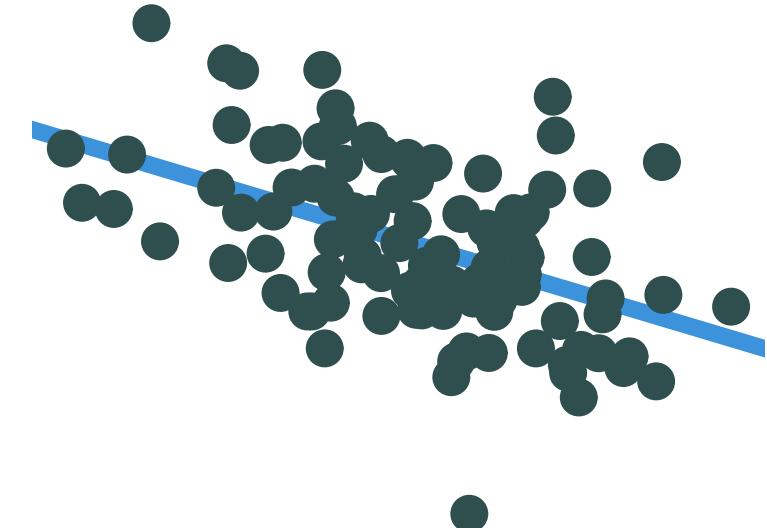


Population

# OLS properties



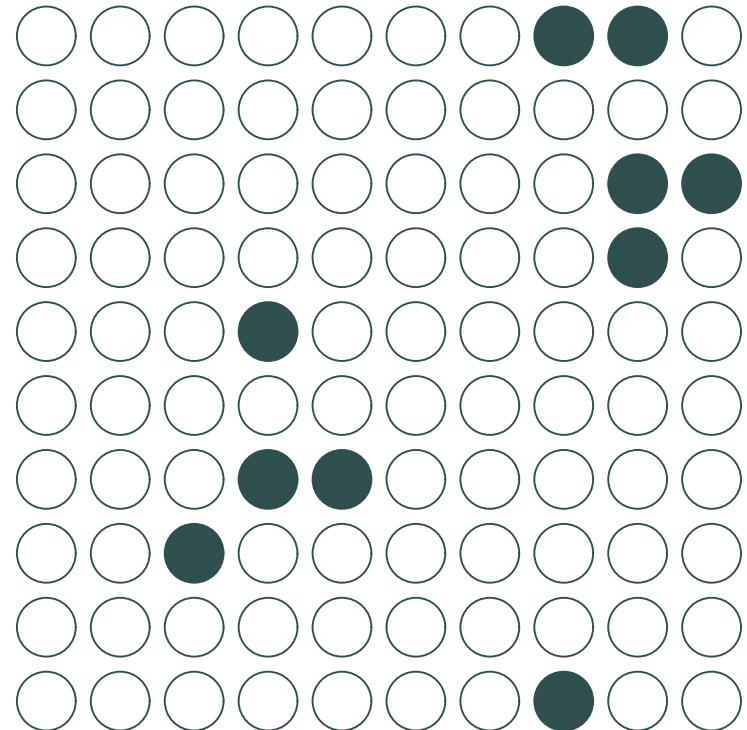
Population



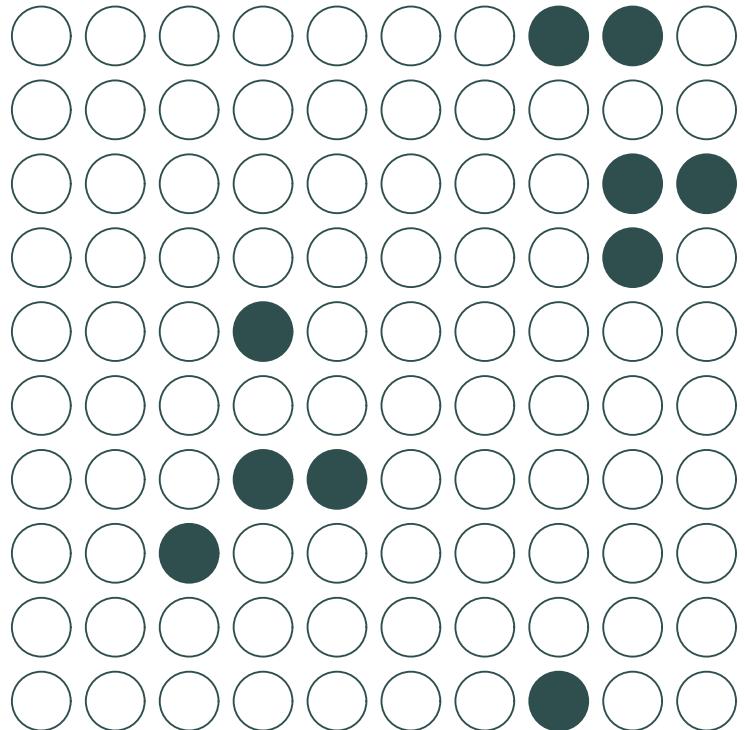
Population relationship

$$y_i = 2.53 + -0.43x_i + u_i$$

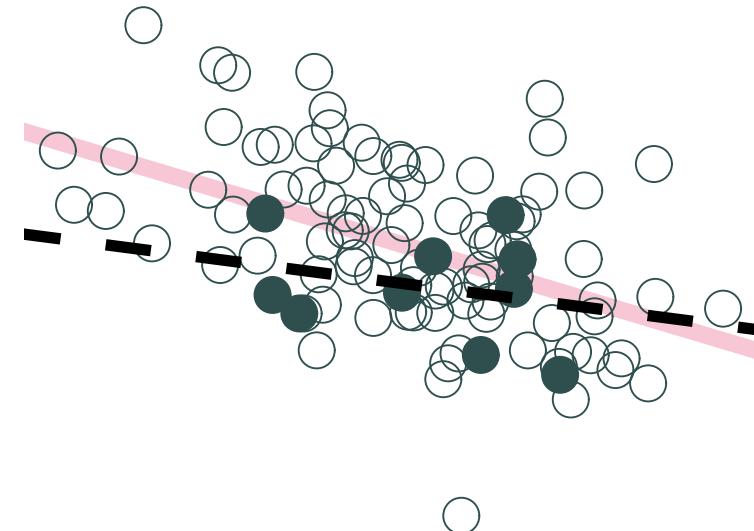
$$y_i = \beta_0 + \beta_1 x_i + u_i$$



**Sample 1:** 10 random individuals



**Sample 1:** 10 random individuals

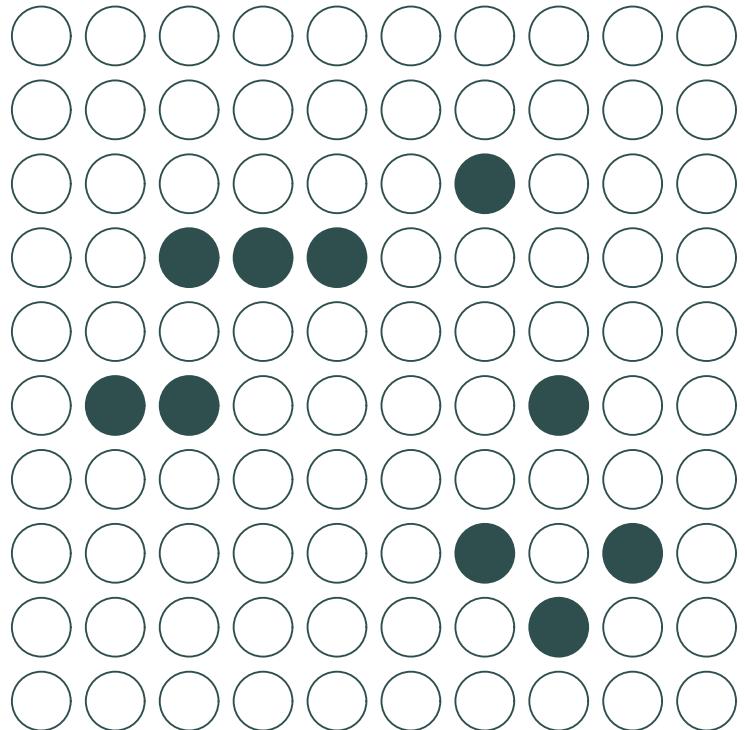


**Population relationship**

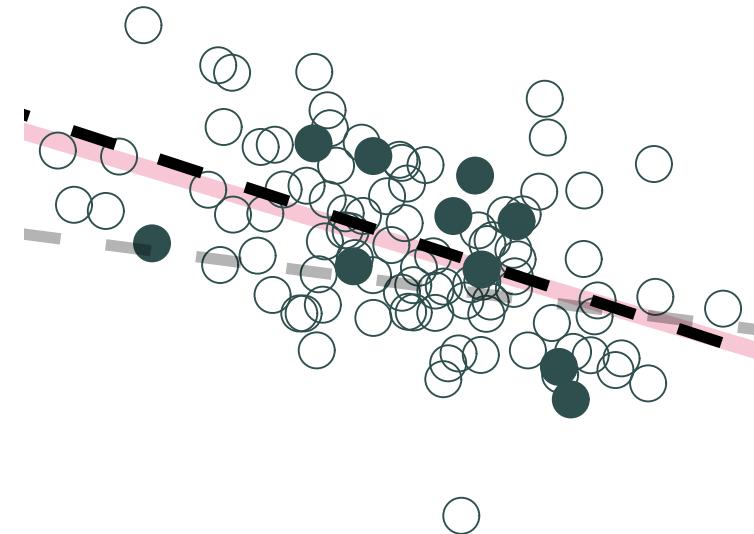
$$y_i = 2.53 + -0.43x_i + u_i$$

**Sample relationship**

$$\hat{y}_i = 0.72 + -0.19x_i$$



**Sample 2:** 10 random individuals

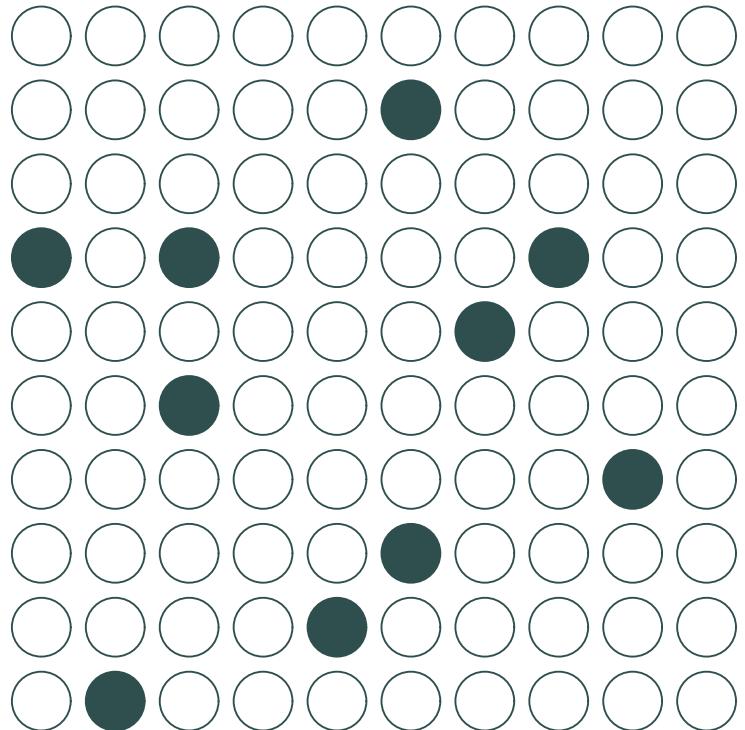


**Population relationship**

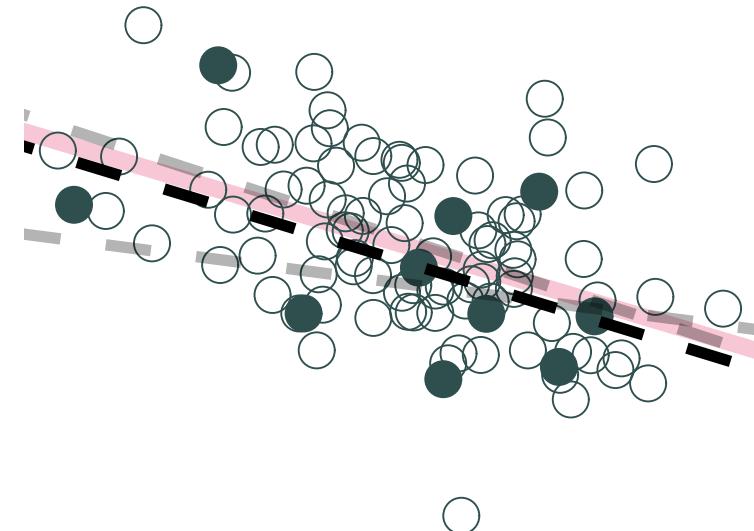
$$y_i = 2.53 + -0.43x_i + u_i$$

**Sample relationship**

$$\hat{y}_i = 2.82 + -0.47x_i$$



**Sample 3:** 10 random individuals



**Population relationship**

$$y_i = 2.53 + -0.43x_i + u_i$$

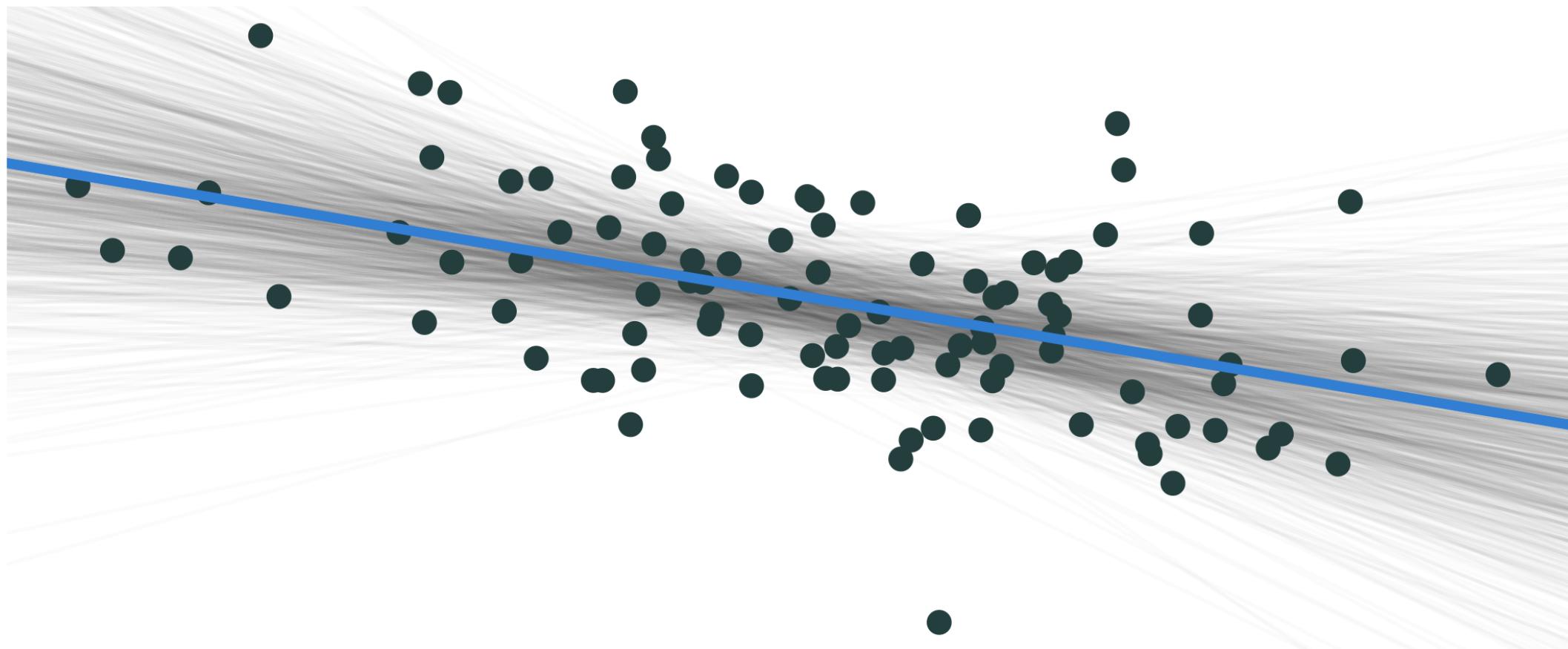
**Sample relationship**

$$\hat{y}_i = 2.32 + -0.44x_i$$

**Let's repeat this 1,000 times.**

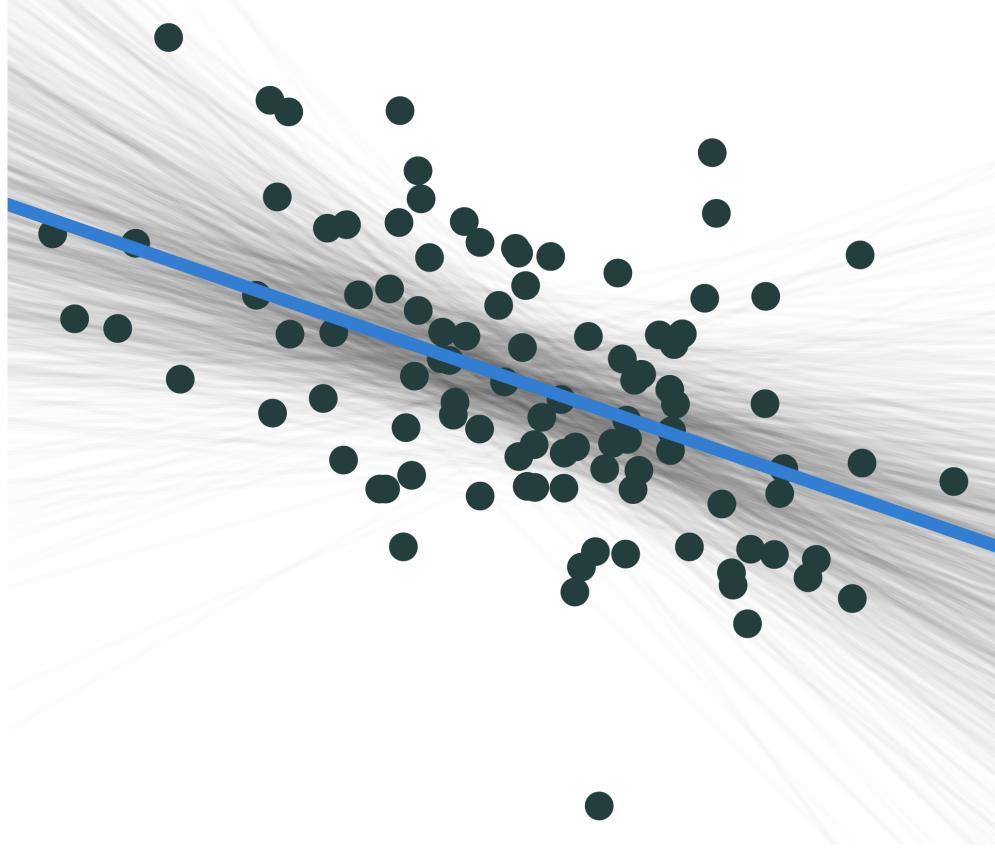
(This exercise is called a (Monte Carlo) simulation.)

# Population vs. sample



# Population vs. sample

**Question:** Why do we care about *population* vs. *sample*?



On **average**, our regression lines match the population line very nicely

However, **individual lines** (samples) can really miss the mark

Differences between individual samples and the population lead to **uncertainty** for us in the true effect

# Population vs. sample

**Answer:** Uncertainty matters!

# Population vs. sample

**Answer:** Uncertainty matters!

$\hat{\beta}$  itself is random, it will depend on the sample of data we have

# Population vs. sample

**Answer:** Uncertainty matters!

$\hat{\beta}$  itself is random, it will depend on the sample of data we have

When we take a sample and run a regression, we don't know if it's a 'good' sample ( $\hat{\beta}$  is close to  $\beta$ ) or a 'bad sample' (our sample differs greatly from the population)

# Unbiasedness

For OLS to be unbiased and give us, on average, the causal effect of some X on some Y we need a few assumptions to hold

# Unbiasedness

For OLS to be unbiased and give us, on average, the causal effect of some X on some Y we need a few assumptions to hold

Whether or not these assumptions are true is why you often hear *correlation is not causation*

# Unbiasedness

For OLS to be unbiased and give us, on average, the causal effect of some X on some Y we need a few assumptions to hold

Whether or not these assumptions are true is why you often hear *correlation is not causation*

If we want some  $\hat{\beta}_1$  on a variable  $x$  to be unbiased we need the following to be true:

$$E[x\varepsilon] = 0 \quad \leftrightarrow \quad \text{correlation}(x, \varepsilon) = 0$$

# Unbiasedness

For OLS to be unbiased and give us, on average, the causal effect of some X on some Y we need a few assumptions to hold

Whether or not these assumptions are true is why you often hear *correlation is not causation*

If we want some  $\hat{\beta}_1$  on a variable  $x$  to be unbiased we need the following to be true:

$$E[x\varepsilon] = 0 \quad \leftrightarrow \quad \text{correlation}(x, \varepsilon) = 0$$

The variable you are interested in **cannot** be correlated with the error term

# Unbiasedness

The variable you are interested in **cannot** be correlated with the error term

# Unbiasedness

The variable you are interested in **cannot** be correlated with the error term

What does this mean in words?

# Unbiasedness

The variable you are interested in **cannot** be correlated with the error term

What does this mean in words?

The error term contains all variables that determine  $y$ , but we *omitted* from our model

# Unbiasedness

The variable you are interested in **cannot** be correlated with the error term

What does this mean in words?

The error term contains all variables that determine  $y$ , but we *omitted* from our model

We are assuming that our variable of interest,  $x$ , is not correlated with any of these omitted variable

# Unbiasedness

The variable you are interested in **cannot** be correlated with the error term

What does this mean in words?

The error term contains all variables that determine  $y$ , but we *omitted* from our model

We are assuming that our variable of interest,  $x$ , is not correlated with any of these omitted variable

If  $x$  is correlated with any of them, then we will have something called **omitted variable bias**

# Omitted variable bias

Here's an intuitive example

# Omitted variable bias

Here's an intuitive example

Suppose we wanted to understand the effect of lead exposure  $P$  on GPAs

# Omitted variable bias

Here's an intuitive example

Suppose we wanted to understand the effect of lead exposure  $P$  on GPAs

lead harms children's brain development, especially before age 6

# Omitted variable bias

Here's an intuitive example

Suppose we wanted to understand the effect of lead exposure  $P$  on GPAs

lead harms children's brain development, especially before age 6

We should expect early-life lead exposure to reduce future GPAs

# Omitted variable bias

Our model might look like:

$$\text{GPA}_i = \beta_0 + \beta_1 \text{P}_i + \varepsilon_i$$

# Omitted variable bias

Our model might look like:

$$\text{GPA}_i = \beta_0 + \beta_1 \text{P}_i + \varepsilon_i$$

We want to know  $\beta_1$

# Omitted variable bias

Our model might look like:

$$\text{GPA}_i = \beta_0 + \beta_1 \text{P}_i + \varepsilon_i$$

We want to know  $\beta_1$

What would happen if we took a sample of *real world data* and used OLS to estimate  $\hat{\beta}_1$ ?

# Omitted variable bias

We would have omitted variable bias

# Omitted variable bias

We would have omitted variable bias

Why? What are some examples?

# Omitted variable bias

We would have omitted variable bias

Why? What are some examples?

**Who** is more likely to be exposed to lead?

# Omitted variable bias

We would have omitted variable bias

Why? What are some examples?

**Who** is more likely to be exposed to lead?

Poorer families likely have more lead exposure, why?

# Omitted variable bias

We would have omitted variable bias

Why? What are some examples?

**Who** is more likely to be exposed to lead?

Poorer families likely have more lead exposure, why?

Richer families can move away, pay to replace lead paint, lead pipes, etc

# Omitted variable bias

We would have omitted variable bias

Why? What are some examples?

**Who** is more likely to be exposed to lead?

Poorer families likely have more lead exposure, why?

Richer families can move away, pay to replace lead paint, lead pipes, etc

This means lead exposure is correlated with lower income

# Omitted variable bias

Why does this correlation cause us problems?

# Omitted variable bias

Why does this correlation cause us problems?

Family income *also* matters for GPA, it is in  $\varepsilon_i$ , so our assumption that  $\text{correlation}(x, \varepsilon) = 0$  is violated

# Omitted variable bias

Why does this correlation cause us problems?

Family income *also* matters for GPA, it is in  $\varepsilon_i$ , so our assumption that  $\text{correlation}(x, \varepsilon) = 0$  is violated

Children from richer families tend to have higher GPAs

# Omitted variable bias

Why does this correlation cause us problems?

Family income *also* matters for GPA, it is in  $\varepsilon_i$ , so our assumption that  $\text{correlation}(x, \varepsilon) = 0$  is violated

Children from richer families tend to have higher GPAs

Why?

# Omitted variable bias

Why does this correlation cause us problems?

Family income *also* matters for GPA, it is in  $\varepsilon_i$ , so our assumption that  $\text{correlation}(x, \varepsilon) = 0$  is violated

Children from richer families tend to have higher GPAs

Why?

Access to tutoring, better schools, parental pressure, etc, etc

# Omitted variable bias

If we just look at the effect of lead exposure on GPAs without addressing its correlation with income, lead exposure will look worse than it actually is

# Omitted variable bias

If we just look at the effect of lead exposure on GPAs without addressing its correlation with income, lead exposure will look worse than it actually is

This is because our data on lead exposure is also proxying for income (since  $\text{correlation}(x, \varepsilon) = 0$ )

# Omitted variable bias

If we just look at the effect of lead exposure on GPAs without addressing its correlation with income, lead exposure will look worse than it actually is

This is because our data on lead exposure is also proxying for income (since  $\text{correlation}(x, \varepsilon) = 0$ )

So  $\hat{\beta}_1$  will pick up the effect of both!

# Omitted variable bias

If we just look at the effect of lead exposure on GPAs without addressing its correlation with income, lead exposure will look worse than it actually is

This is because our data on lead exposure is also proxying for income (since  $\text{correlation}(x, \varepsilon) = 0$ )

So  $\hat{\beta}_1$  will pick up the effect of both!

Our estimate  $\hat{\beta}_1$  is **biased** and overstates the negative effects of lead

# Omitted variable bias

How do we fix this bias?

# Omitted variable bias

How do we fix this bias?

Make income not omitted: control for it in our model

# Omitted variable bias

How do we fix this bias?

Make income not omitted: control for it in our model

If we have data on family income  $I$  we can instead write our model as:

$$\text{GPA}_i = \beta_0 + \beta_1 P_i + \beta_2 I_i + \varepsilon_i$$

$I$  is no longer omitted

# Omitted variable bias

How do we fix this bias?

Make income not omitted: control for it in our model

If we have data on family income  $I$  we can instead write our model as:

$$\text{GPA}_i = \beta_0 + \beta_1 P_i + \beta_2 I_i + \varepsilon_i$$

$I$  is no longer omitted

Independent variables in our model that we include to address bias are called  
**controls**

# Regression in R

---

# How do we actually run regressions?

Now we are going to learn how to run regressions in R

# How do we actually run regressions?

Now we are going to learn how to run regressions in R

There's a lot of regression packages, built-in is `lm`, or you can also use a new one called `fixest`

# How do we actually run regressions?

Now we are going to learn how to run regressions in R

There's a lot of regression packages, built-in is `lm`, or you can also use a new one called `fixest`

We will also be using the `broom` package to make our output look nice

# How do we actually run regressions?

Now we are going to learn how to run regressions in R

There's a lot of regression packages, built-in is `lm`, or you can also use a new one called `fixest`

We will also be using the `broom` package to make our output look nice

They work almost identically

# Using regression packages

To run a regression we need 3 things:

# Using regression packages

To run a regression we need 3 things:

1. The package that will run the regression
2. Our regression formula
3. The dataframe that contains our data

# Using regression packages

To run a regression we need 3 things:

1. The package that will run the regression
2. Our regression formula
3. The dataframe that contains our data

In general, we will always run a regression like this:

```
package_name(formula_here, data = dataframe_here)
```

# Using regression packages

To run a regression we need 3 things:

1. The package that will run the regression
2. Our regression formula
3. The dataframe that contains our data

In general, we will always run a regression like this:

```
package_name(formula_here, data = dataframe_here)
```

You can then store the output by assigning it to a variable: `results =`

```
package_name(formula_here, data = dataframe_here)
```

# Using regression packages

Let's start by using the built-in `lm` package along with the `starwars` dataset

```
starwars
```

```
## # A tibble: 87 × 14
##   name      height  mass hair_color skin_color eye_color birth_year sex gender homeworld species
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>        <dbl> <chr> <chr> <chr>    <chr>
## 1 Luke Skywalker 172     77 blond      fair        blue          19 male   masculin… Tatooine Human
## 2 C-3PO           167     75 <NA>       gold        yellow        112 none   masculin… Tatooine Droid
## 3 R2-D2            96     32 <NA>       white, bl… red          33 none   masculin… Naboo   Droid
## 4 Darth Vader    202    136 none       white        yellow        41.9 male   masculin… Tatooine Human
## 5 Leia Organa    150     49 brown      light       brown          19 female feminin… Alderaan Human
## 6 Owen Lars      178    120 brown, gr… light       blue          52 male   masculin… Tatooine Human
## 7 Beru Whitesun 165     75 brown      light       blue          47 female feminin… Tatooine Human
## 8 R5-D4            97     32 <NA>       white, red red          NA none   masculin… Tatooine Droid
## 9 Biggs Darkli… 183     84 black      light       brown          24 male   masculin… Tatooine Human
## 10 Obi-Wan Keno… 182     77 auburn, w… fair        blue-gray      57 male   masculin… Stewjon Human
## # ... with 77 more rows, and 2 more variables: vehicles <list>, starships <list>
```

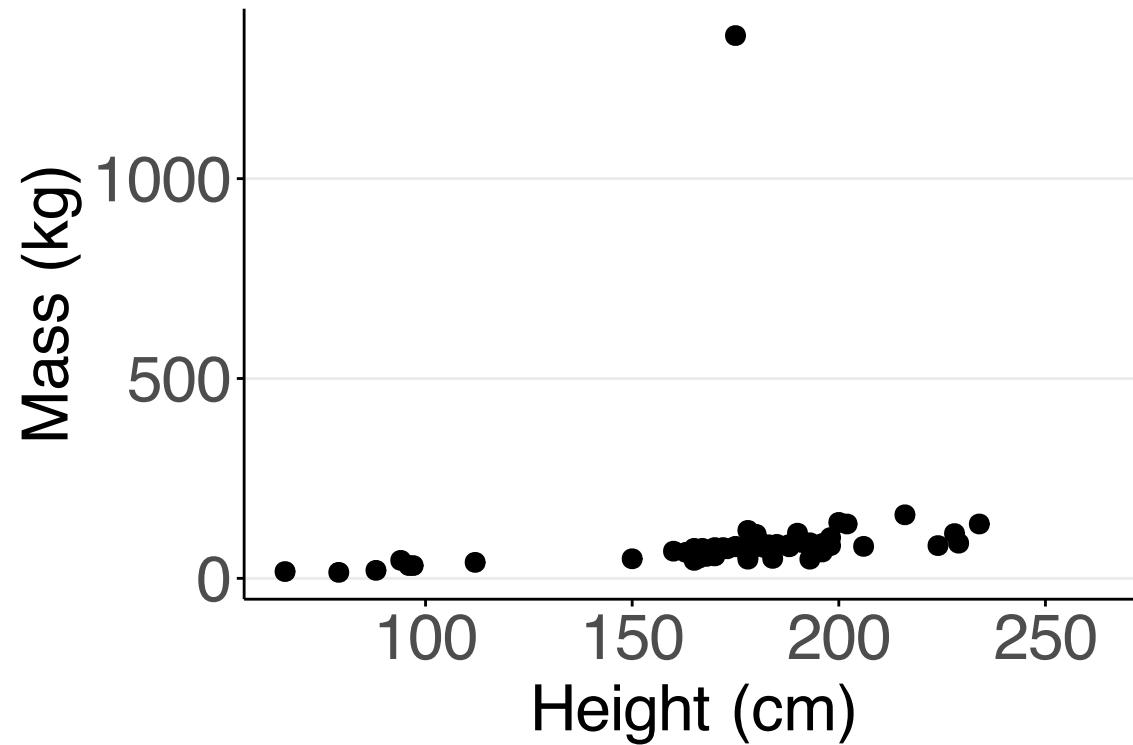
# Using regression packages

Suppose we wanted to see what was the effect of height on mass:

$$mass_i = \beta_0 + \beta_1 height_i + \varepsilon_i$$

# Using regression packages

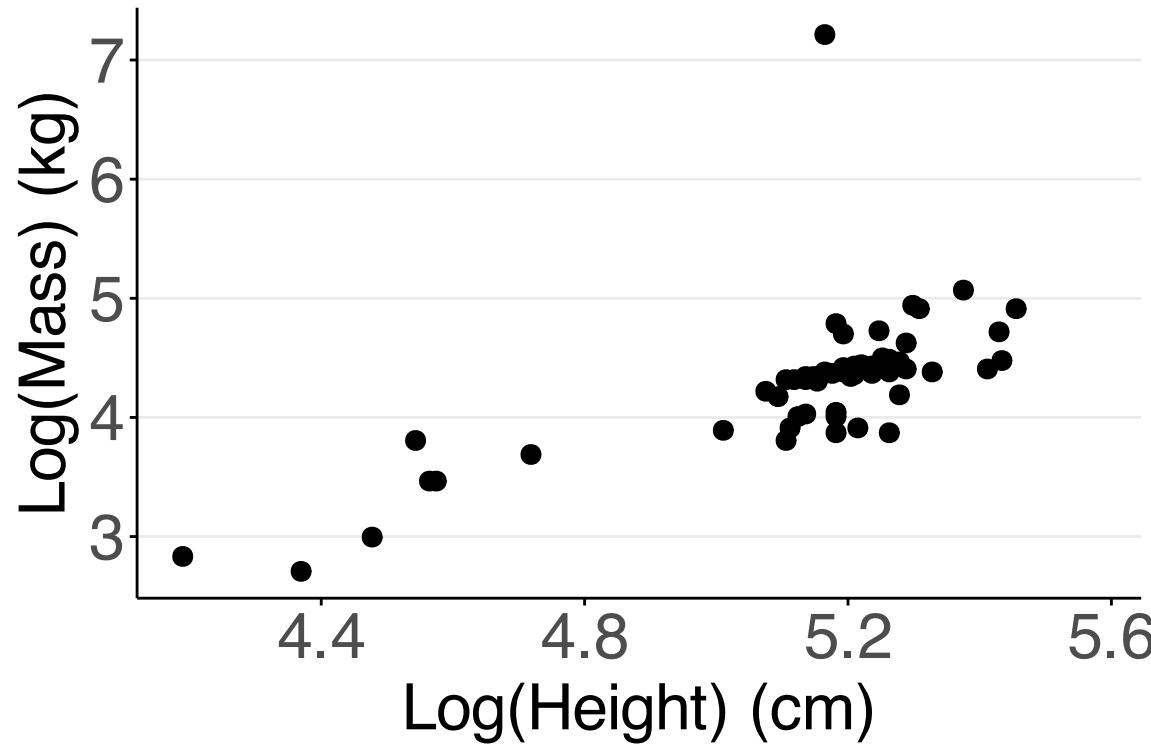
Before we begin lets look at the data and see what the relationship looks like:



Looks like there's an outlier!

# Using regression packages

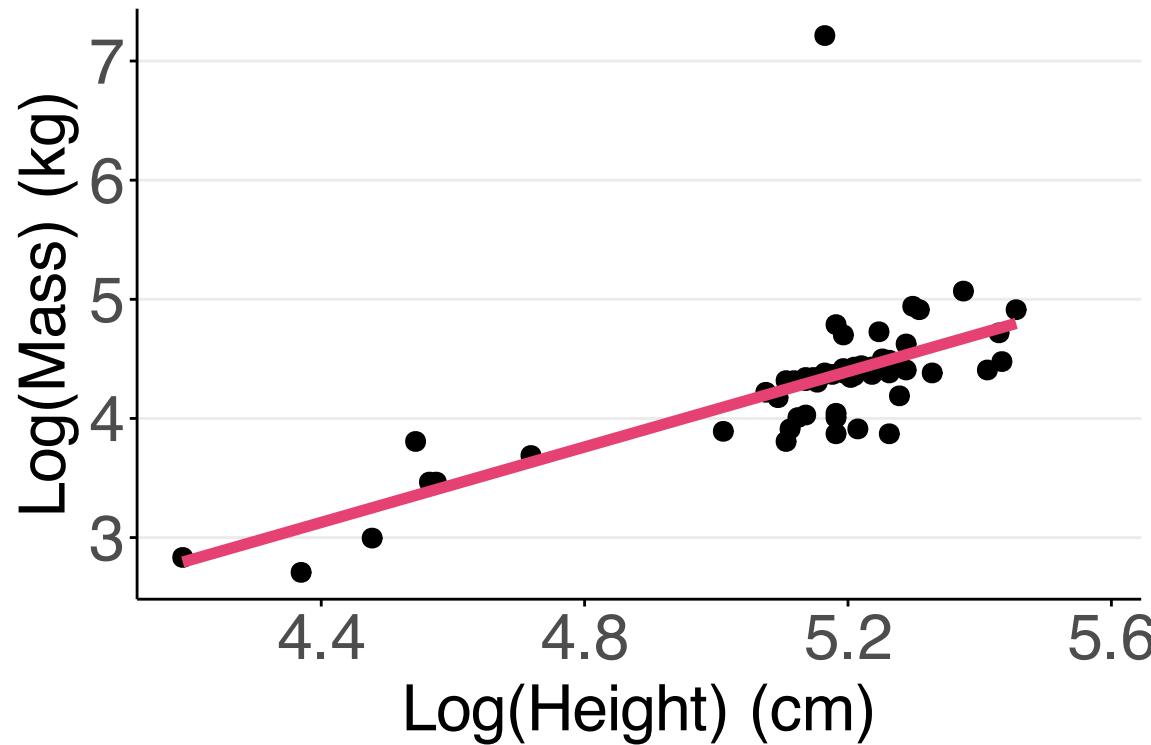
What if we plot it in log units?



That dampens the outlier, the positive association is really clear!

# Using regression packages

What if we plot it in log units?



Here's the best fit line, now let's actually estimate  $\beta_0, \beta_1$

# Using regression packages

First we need to construct our formula

# Using regression packages

First we need to construct our formula

We do this like: dependent variable ~ independent variable 1 +  
independent variable 2 + ... + independent variable N

# Using regression packages

First we need to construct our formula

We do this like: dependent variable ~ independent variable 1 +  
independent variable 2 + ... + independent variable N

For this example we would write: mass ~ height

# Using regression packages

$$mass_i = \beta_0 + \beta_1 height_i + \varepsilon_i$$

We can run the following code:

```
# package_name(formula_here, data = dataframe_here)
lm(mass ~ height, data = starwars)
```

```
##
## Call:
## lm(formula = mass ~ height, data = starwars)
##
## Coefficients:
## (Intercept)      height
## -13.8103        0.6386
```

This gives us the coefficient estimate  $\hat{\beta}_0$  (intercept), and  $\hat{\beta}_1$  (height)

# Using regression packages

We can clean up the output a bit and get other statistics by piping it to

`broom::tidy:`

```
# package_name(formula_here, data = dataframe_here)
lm(mass ~ height, data = starwars) |>
  broom::tidy()
```

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>      <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept) -13.8     111.      -0.124   0.902
## 2 height       0.639     0.626      1.02    0.312
```

The coefficient estimates are now in column `estimate`, and we have a few other things

# Using regression packages

The first column gives us our estimates:  $\hat{\beta}_0, \hat{\beta}_1$

# Using regression packages

The first column gives us our estimates:  $\hat{\beta}_0, \hat{\beta}_1$

The last column is the **p-value**: the probability that we would have gotten an estimate at least that large, if the *true* value was actually 0

# Using regression packages

The first column gives us our estimates:  $\hat{\beta}_0, \hat{\beta}_1$

The last column is the **p-value**: the probability that we would have gotten an estimate at least that large, if the *true* value was actually 0

Smaller values generally mean it is more likely that height has an effect on mass: the probability that we could have gotten our estimated value if height didn't matter is very low

# Using regression packages

The first column gives us our estimates:  $\hat{\beta}_0, \hat{\beta}_1$

The last column is the **p-value**: the probability that we would have gotten an estimate at least that large, if the *true* value was actually 0

Smaller values generally mean it is more likely that height has an effect on mass: the probability that we could have gotten our estimated value if height didn't matter is very low

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

What do these coefficient estimates mean?

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

What do these coefficient estimates mean?

The estimate for the  $\beta$  on height means: people who are 1cm taller tend to be 0.639kg heavier

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

What do these coefficient estimates mean?

The estimate for the  $\beta$  on height means: people who are 1cm taller tend to be 0.639kg heavier

This just comes from our previous example where  $\beta_1 = \frac{\partial mass_i}{\partial height_i}$

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

How do we interpret  $\beta_0$ , the estimate of the *intercept*?

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

How do we interpret  $\beta_0$ , the estimate of the *intercept*?

Well, it is just the estimated mass, given someone had zero height:

$$\hat{\beta}_0 = \hat{\beta}_0 + \hat{\beta}_1 \times 0$$

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

How do we interpret  $\beta_0$ , the estimate of the *intercept*?

Well, it is just the estimated mass, given someone had zero height:

$$\hat{\beta}_0 = \hat{\beta}_0 + \hat{\beta}_1 \times 0$$

It's kind of a nonsense interpretation here since no one has zero height

# Interpreting coefficient estimates

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -13.8     111.    -0.124    0.902
## 2 height       0.639     0.626     1.02     0.312
```

How do we interpret  $\beta_0$ , the estimate of the *intercept*?

Well, it is just the estimated mass, given someone had zero height:

$$\hat{\beta}_0 = \hat{\beta}_0 + \hat{\beta}_1 \times 0$$

It's kind of a nonsense interpretation here since no one has zero height

Generally we don't read too much into the intercept terms

# Interpreting coefficient estimates

Now, what if we changed our model a bit so it was instead:

$$\log(\text{mass}_i) = \beta_0 + \beta_1 \log(\text{height}_i) + \varepsilon_i$$

What does  $\beta_1$  mean now?

# Interpreting coefficient estimates

Now, what if we changed our model a bit so it was instead:

$$\log(\text{mass}_i) = \beta_0 + \beta_1 \log(\text{height}_i) + \varepsilon_i$$

What does  $\beta_1$  mean now?

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)}$$

# Interpreting coefficient estimates

Now, what if we changed our model a bit so it was instead:

$$\log(\text{mass}_i) = \beta_0 + \beta_1 \log(\text{height}_i) + \varepsilon_i$$

What does  $\beta_1$  mean now?

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)}$$

But we can rewrite this as:

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)} = \frac{\partial \log(\text{mass}_i)}{\partial \text{mass}_i} \frac{\partial \text{mass}_i}{\partial \text{height}_i} \frac{\partial \text{height}_i}{\partial \log(\text{height}_i)}$$

# Interpreting coefficient estimates

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)} = \frac{\partial \log(\text{mass}_i)}{\partial \text{mass}_i} \frac{\partial \text{mass}_i}{\partial \text{height}_i} \frac{\partial \text{height}_i}{\partial \log(\text{height}_i)}$$

And this is equal to:

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)} = \frac{1}{\text{mass}_i} \frac{\partial \text{mass}_i}{\partial \text{height}_i} \frac{\text{height}_i}{1}$$

# Interpreting coefficient estimates

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)} = \frac{\partial \log(\text{mass}_i)}{\partial \text{mass}_i} \frac{\partial \text{mass}_i}{\partial \text{height}_i} \frac{\partial \text{height}_i}{\partial \log(\text{height}_i)}$$

And this is equal to:

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)} = \frac{1}{\text{mass}_i} \frac{\partial \text{mass}_i}{\partial \text{height}_i} \frac{\text{height}_i}{1}$$

And finally:

$$\beta_1 = \frac{\partial \log(\text{mass}_i)}{\partial \log(\text{height}_i)} = \frac{\text{height}_i}{\text{mass}_i} \frac{\partial \text{mass}_i}{\partial \text{height}_i}$$

which is the definition of the elasticity of mass with respect to height

# Interpreting coefficient estimates

$$\log(\text{mass}_i) = \beta_0 + \beta_1 \log(\text{height}_i) + \varepsilon_i$$

In a *log-log* model,  $\beta_1$  tells us the percent change in mass, given a percent change in height

# Interpreting coefficient estimates

$$\log(\text{mass}_i) = \beta_0 + \beta_1 \log(\text{height}_i) + \varepsilon_i$$

In a *log-log* model,  $\beta_1$  tells us the percent change in mass, given a percent change in height

Let's run the regression:

```
# package_name(formula_here, data = dataframe_here)
lm(log(mass) ~ log(height), data = starwars) |>
  broom::tidy()

## # A tibble: 2 × 5
##   term      estimate std.error statistic    p.value
##   <chr>      <dbl>     <dbl>     <dbl>      <dbl>
## 1 (Intercept) -3.84      1.17     -3.27 0.00181
## 2 log(height)  1.58      0.228      6.93 0.00000000410
```

# Interpreting coefficient estimates

```
# package_name(formula_here, data = dataframe_here)
lm(log(mass) ~ log(height), data = starwars) |>
  broom::tidy()
```

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic    p.value
##   <chr>      <dbl>     <dbl>     <dbl>      <dbl>
## 1 (Intercept) -3.84      1.17     -3.27 0.00181
## 2 log(height)   1.58      0.228      6.93 0.00000000410
```

A 1% increase in height is associated with a 1.58% increase in mass!