Introduction to R

CRDDS Summer Research Data Bootcamp

May 17, 2023

Scott Nordstrom

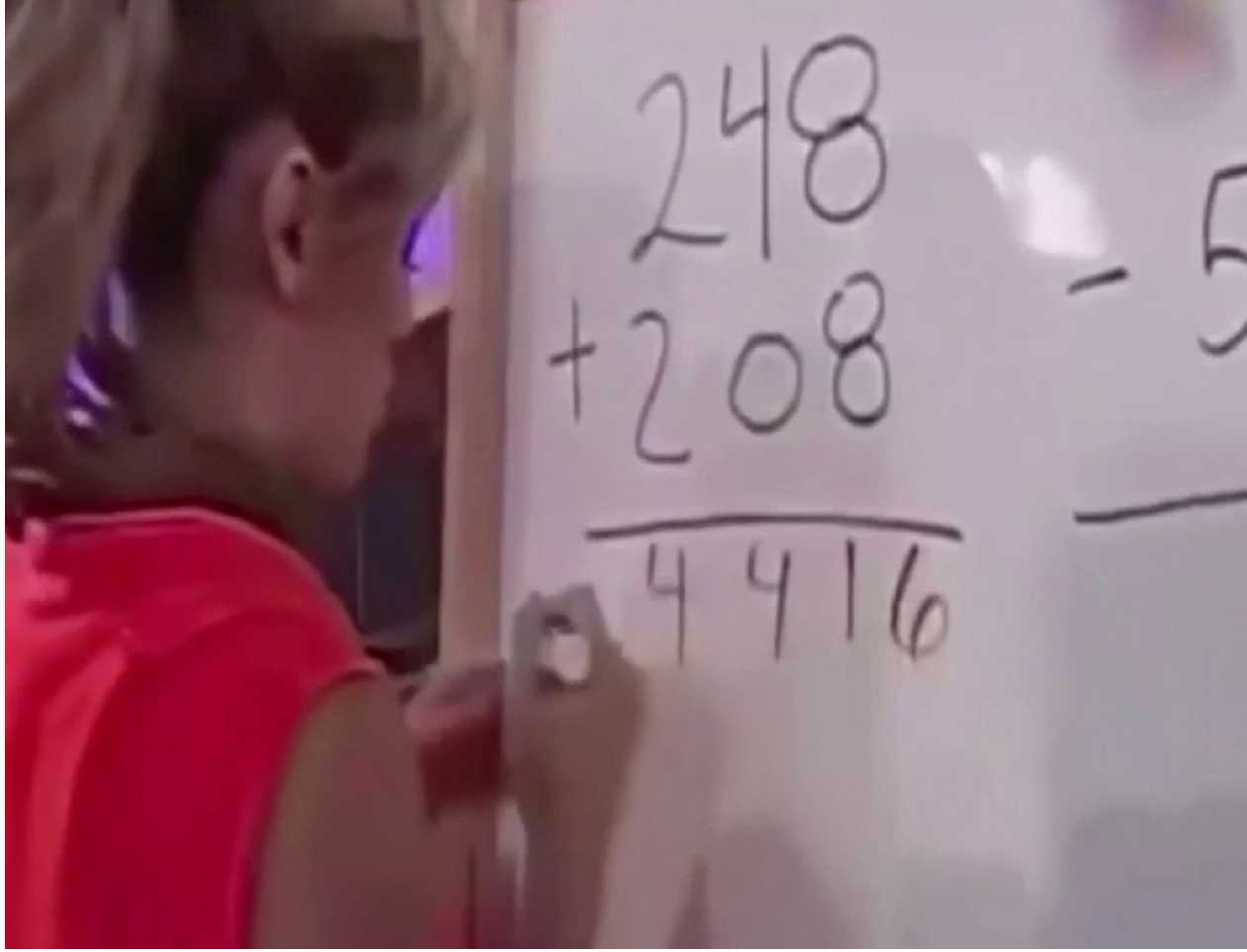
(Scott.Nordstrom@Colorado.edu,

scottwatsonnordstrom@gmail.com)

https://posit.cloud/

Why use R?

- Less error-prone
- More efficient and faster
- Handle large, disparate datasets
- Reproducible

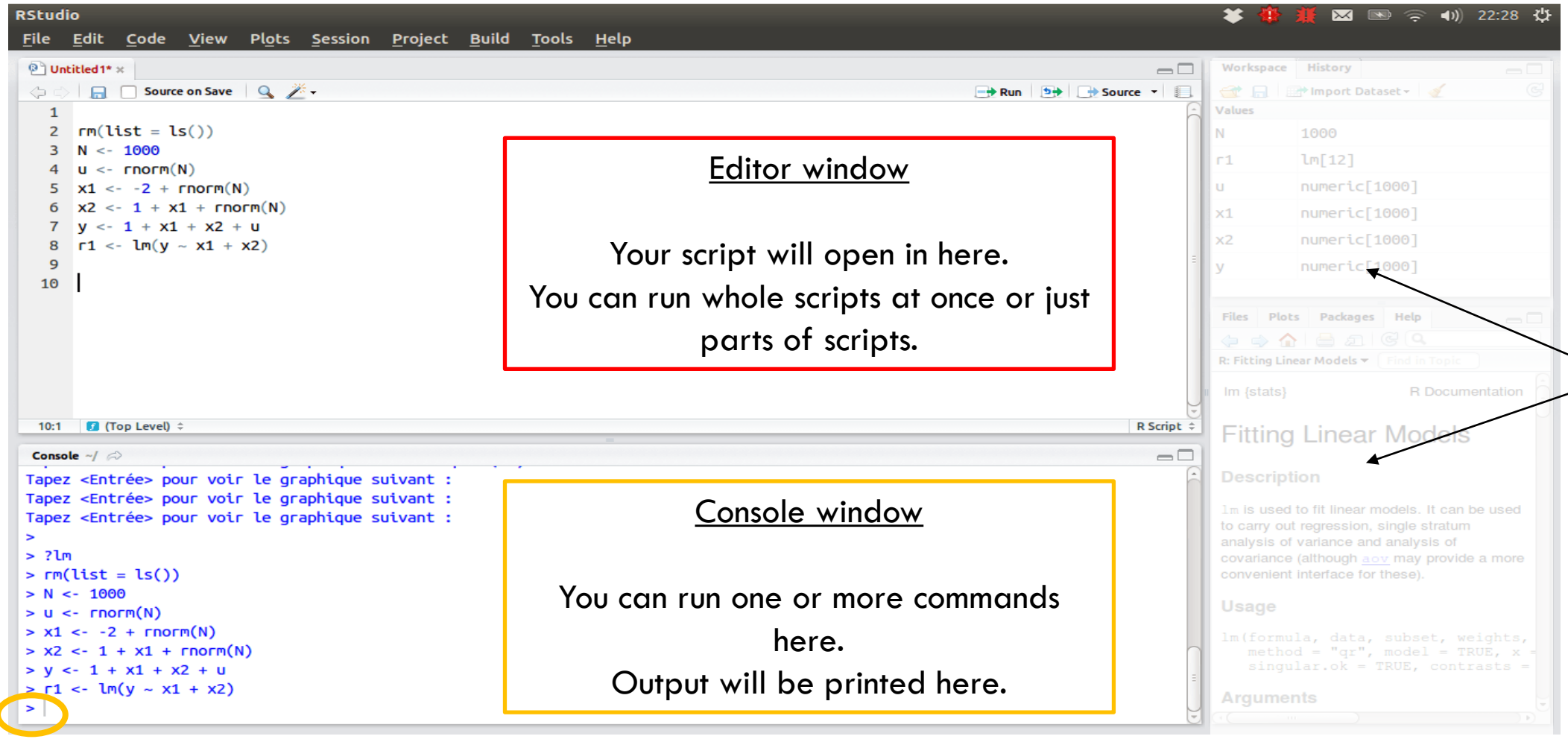


- Others can re-create your work
- *You* can re-create your work
- Reliable – should give same answer each time

R vs. RStudio

- R is the language, RStudio is software
- Analogy: if R was English, then RStudio would be Microsoft Word
- Further torturing the analogy: R is English, Rstudio is Microsoft Word, a .R file (“script”) is a poem

RStudio: what's going on here?



Editor window

Your script will open in here.
You can run whole scripts at once or just parts of scripts.

Console window

You can run one or more commands here.
Output will be printed here.

Let's ignore these for now.

Running R commands in the console

RStudio

File Edit Code View Plots Session Project Build Tools Help

Untitled1* | Source on Save | Run | Source

```
1
2  rm(list = ls())
3  N <- 1000
4  u <- rnorm(N)
5  x1 <- -2 + rnorm(N)
6  x2 <- 1 + x1 + rnorm(N)
7  y <- 1 + x1 + x2 + u
8  r1 <- lm(y ~ x1 + x2)
9
10
```

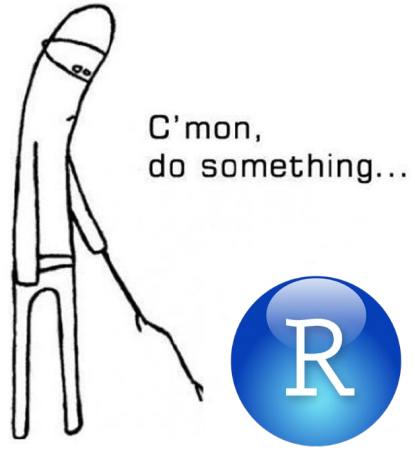
10:1 (Top Level) R Script

Console ~/

```
Tapez <Entrée> pour voir le graphique suivant :
Tapez <Entrée> pour voir le graphique suivant :
Tapez <Entrée> pour voir le graphique suivant :
>
> ?lm
> rm(list = ls())
> N <- 1000
> u <- rnorm(N)
> x1 <- -2 + rnorm(N)
> x2 <- 1 + x1 + rnorm(N)
> y <- 1 + x1 + x2 + u
> r1 <- lm(y ~ x1 + x2)
>
```

You can type things directly into the console and hit “Enter”

Note: you can scroll up to see old commands you’ve run, but otherwise, *this won’t necessarily save your work.*



Arithmetic and logic in R

- R (and any programming language) can do simple arithmetic

- Try running the following commands in the console:

I'll use this font throughout the workshop denote R code

The font is "Monaco" if you ever want to use it – it's the same as the default in RStudio!

```
2 + 2
[1] 4
```

This is "*returned*" by the command – running commands usually prints the output in the console.

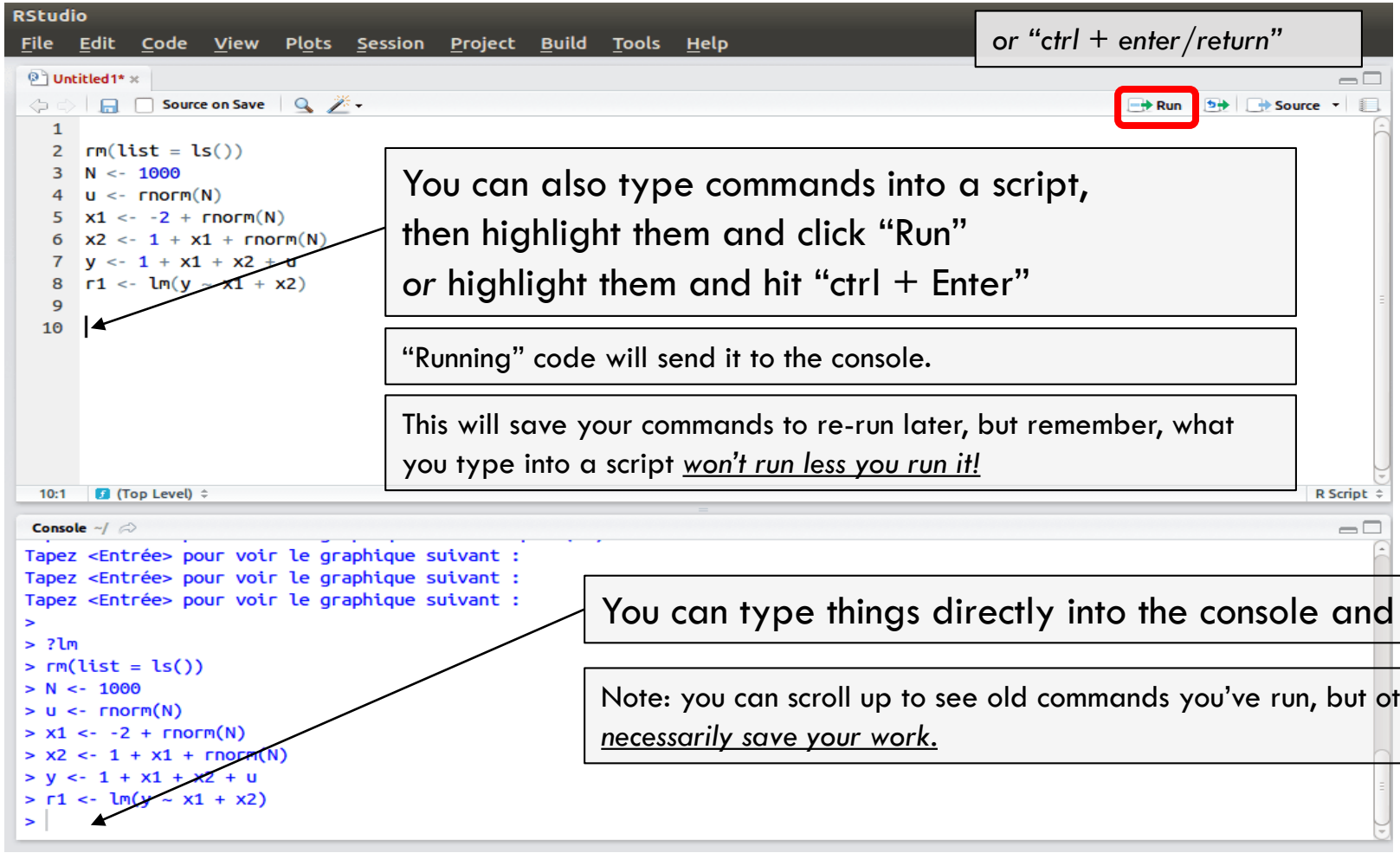
```
(2 + 2)^2
```

Keep your order of operations (PEMDAS) in mind!

```
19^2 < 4^4
```

This is what's called a "logical statement" or a "boolean statement" – like asking a question where the answer is TRUE *or* FALSE

Running R commands from the editor



or "ctrl + enter/return"

You can also type commands into a script, then highlight them and click "Run" *or* highlight them and hit "ctrl + Enter"

"Running" code will send it to the console.

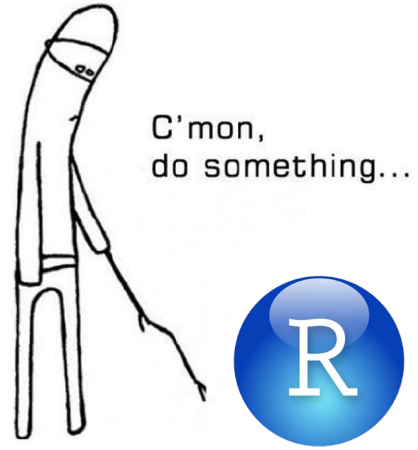
This will save your commands to re-run later, but remember, what you type into a script *won't run less you run it!*

You can type things directly into the console and hit "Enter"

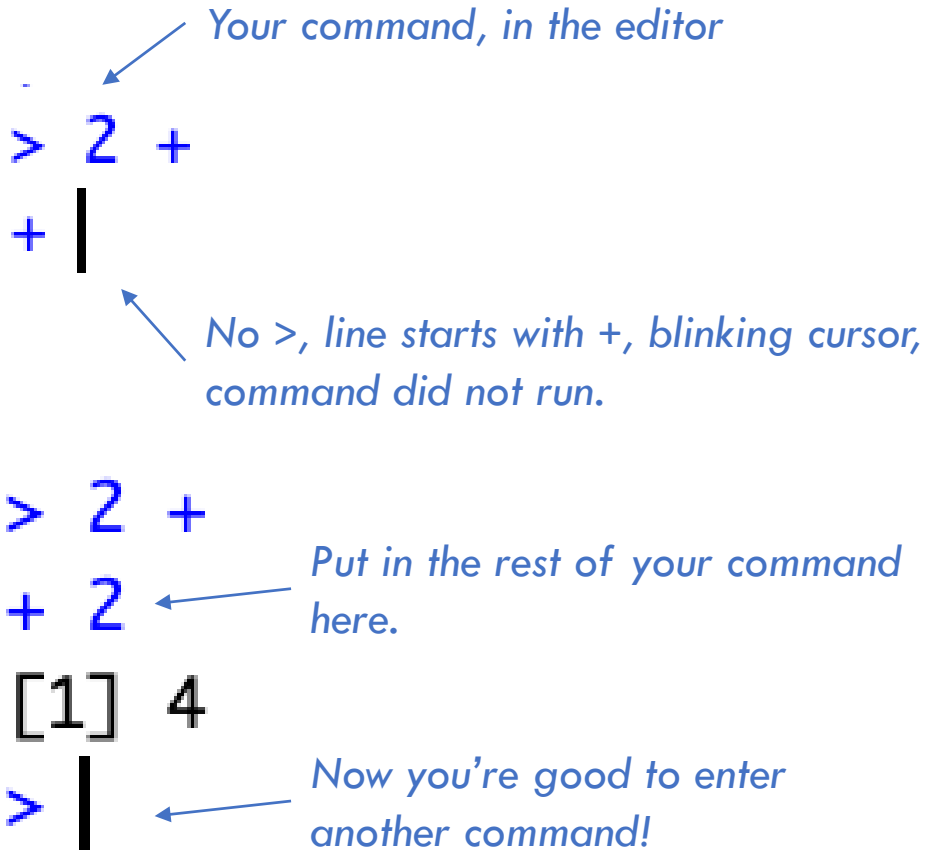
Note: you can scroll up to see old commands you've run, but otherwise, *this won't necessarily save your work.*

How I (Scott) do it:

- Use the console for trying stuff out
- Do what I want to save in the editor



“Hanging” (incomplete) commands



Common source of error is entering incomplete commands.

When the console gets an incomplete command, *it waits for you to close the command* before running. The “+” is like an invitation to finish the command.

You can fix this by finishing the command.

This happens commonly with arithmetic (e.g., +, /), parentheses and brackets, and quotes.

Variables: saving objects in memory

```
x <- 2 + 2
```

This will save the output (right of the arrow) into memory under the name x

```
x
[1] 4
```

Running just the variable name itself will print its value

```
(x <- 2 + 2)
[1] 4
```

(useful to make sure the output is what you expected it to be!)

Little known trick – wrapping this statement in parentheses will print x to the console too!

Useful rules for naming variables:

- Can't include a space
- Can't start with numbers
- Can include underscore (_) and period (.)
- Names are case sensitive

Widely followed conventions:

- Use informative names!
- Start with lowercase letters
- camelCase or underscores for names with multiple words
  - E.g., myData or my_data – easier to read than mydata

<- vs. =

- Both are “assignment” operators – they do the same thing
  - <- has extra functionality in some high-level circumstances
- Most people use <- but I prefer to use =
  - = is one keystroke while <- is three!

Image: hottake.life

```
x <- 2 + 2

y <- 3 * 18

z <- y / x
```

You can plug variables into commands!

Variables: when they stay and when they go

Declare variables and they stay in memory unless

- You overwrite them
- You remove them (rm())
- You restart RStudio (or the session)

```
x = -20
x = 4*4
x

rm(x)
x
```

See your variables with the command ls()

If you're running a script and your R session crashes, all your variables will be wiped out…

(but if you save your script, you can just run everything again!)

Comment with # – har

If you include the # (pound sign) in a lin

R will ignore (not run) everything after i

```
2 + 2 + 2 # + 2
```

What will this return?

```
2 + 2 + # 2 + 2
```

What will happen with

```
### Define a global variable for max age to use in model:
max.age = 6 # for now set to 6 (years 2013 - 2019)

### Define variables for model
vars = c(paste0('ld.age', 0:max.age), "lfCt.age6")

length(vars)
vars

### Defining the graphical structure (predecessor)
# (I think this should just be linear?)
pred = 0:(max.age+1)

# Check to make sure this works
rbind(succ = vars, pred = c("initial", vars)[pred + 1])

### Defining families
# Every living during variable should be bernoulli
# Assume that leaf count is zero-truncated Poisson (check with group)
fam = c(rep(1, (length(vars)) - 1), 3)

### Reshaping data frame:

# It's already in long form
head(t2)
```

These are great to include to “narrate” scripts

useful to someone else looking at your code,

but also useful to future you looking at your code!

Functions

Commands that have parentheses after them are **functions**.

Most functions have an **input(s)** and **output**.

The inputs are often called **arguments**.

(you can write your own functions!)

The "argument" goes here – it's what you want to take the cosine of

cos()

cos(0)

We usually say the function "returns" some output – in this case it would return "1".

paste(..., sep = ' ')

Characters you want to paste together to here.

This function will combine a vector of characters into one object, with each object in the vector separated by whatever you give as the argument "sep"

Getting help

? Followed by the name of a function will pull up the help/documentation page for that function

? sqrt

Also provides examples!

Other data “types”

So far we have dealt only with integers.

Unsurprisingly, R can also handle non-integer numbers.

```
sqrt(.5)
[1] 0.7071068
```

Characters (“strings”):

“character” – wrapped in quotes (single or double)

“two” ← With quotes – character

```
character.two <- “two”
```

Without quotes – variable/object

```
typeof(character.two)
```

typeof() function to check the type of an object

“Logical” data
(TRUE/FALSE)

”Logical” (or “boolean”, “binary”) data is a statement that is true or false

```
5 > 4
[1] TRUE
```

```
five.four <- 5 > 4
five.four
[1] TRUE
```

Logicals get treated by R as 1/0:

TRUE: 1

FALSE: 0

Note: you can use T/F as shorthand for TRUE/FALSE

(so avoid using T *or* F *as variable names!)*

```
sum(c(T, T, F, F, T))
[1] 3
```

This can come in handy!

Moving back and forth between types

For each type, there is a function to convert into that type:

as.character(), as.numeric(), as.logical(), etc.

“5” + “4” # *will return an error*

as.numeric(“5”) + as.numeric(“4”)

[1] 9

Break!

Store multiple objects in **vectors**

Make a vector with the command c()

```
brady_sb_wins <- c(2001, 2003, 2004, 2014, 2016, 2018, 2020)
```

```
brady_sb_wins <- c('xxxvi', 'xxxviii', 'xxxix', 'xlix', 'li', 'liii', 'lv')
```

Vector elements must all be of the same type.


Michigan football legend, Tom Brady

```
ex_vec <- c(5, 6, 'red')
ex_vec
[1] "5"   "6"   "red"
```

all elements were coerced into characters

“Indexing” - picking out a slice or piece

What if I want to pick out just one (or a couple) observations?

For a vector: use brackets [ ]

letters

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z”
```

letters[5]

```
[1] "e"
```

The [5] here says “pick out just the fifth element”

Indexing a vector

Say now I want to pick out multiple elements (but not all of them)

Make a vector of the elements you want to pick out using c()
and stick that in brackets.

Pick out elements, 5, 22, 15, …

```
> letters[c(5, 22, 15, 12, 21, 20, 9, 15, 14)]
[1] "e" "v" "o" "l" "u" "t" "i" "o" "n”
```

Add to a vector:

```
usa.colors <- c('red', 'white', 'blue')

azeri.colors <- c(usa.colors, 'green')
azeri.colors
[1] "red"   "white" "blue"  "green"

lebanese.colors <- azeri.colors[-3]
lebanese.colors
[1] "red"   "white"  "green"
```

"Minus" indices to remove them (you can do this with multiple!, e.g., try – c(3, 4))

Modifying vectors:

```
squares <- c(1, 4, 9, 16, 25)

sqrt(squares[1])
sqrt(squares[2])
...

square.roots <- sqrt(squares)
square.roots
[1] 1 2 3 4 5
```

Many functions in R are written to be "vectorized", i.e., can perform operations on each element independently

Data frames

2D object meant for handling datasets (think: spreadsheets)

- Rows: typically correspond to one "data point"
- Columns: can be numbers or characters

E.g., the "cars" data frame:

```
> cars
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
```

Columns named "speed" and "dist" hold data for one observation.

Probably variables in your analysis!

Rows correspond to observations.

R has a lot of built-in features meant to work with data frames

Indexing a data frame

Data frames have rows and columns

You'll still use brackets, but now there are *two* indices

[rows, columns]

```
cars[1,1]
```

```
[1] 4
```

Picks out first row and first column

```
cars[c(5, 7, 10), 2]
```

```
[1] 16 18 17
```

Picks out rows 5, 7, 10 and second column

Indexing a data frame

But remember…

R has a lot of built-in features meant to work with data frames

Pick out one column of a data frame with the dollar sign ($)

cars$speed

cars$speed[5]

Pick out multiple columns using character vectors

cars[,c(“dist”, “speed”)]

Note that indexing is order specific!
What did this do?

More fun with indexing!

You can index with variables you have made!

```
fast_cars = c(47, 48, 49)

cars[fast_cars, ]
```

```
  speed dist
47    24   92
48    24   93
49    24  120
```

You can index with logical statements!

What is this line doing?

```
cars[cars$speed < 15,]
```

== in logical statements – don't confuse with =

```
cars[cars$speed == 15,]
cars[cars$speed != 15,]
```

Exclamation mark *negates* logical statements

The dreaded error…

Error – some part of your code did not execute

- Error messages – these are informative! Read them!
- You can also google error messages!

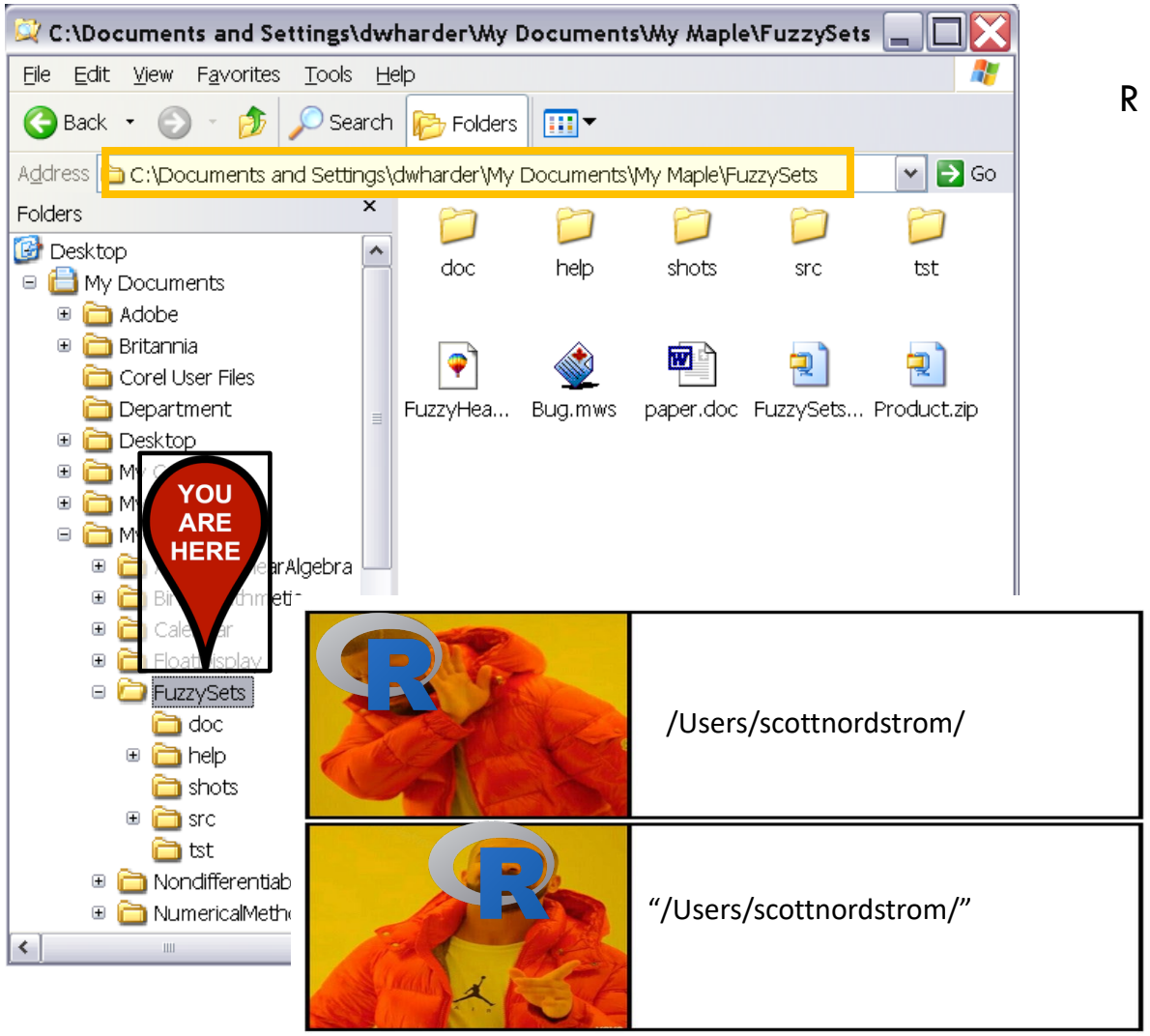
"Debugging" – fixing "bugs" in your code

Note – just because you don't get an error message doesn't mean your code runs *as you want it!*

All of that is very cool.

How do I analyze my own data though…?

Directories – where am I?



R sessions are always "parked" in some directory on your computer.

Where it is parked is called your "*working directory.*"

Check your working directory: getwd()

Set your working directory: setwd()

Directories (and file names) should be in quotes!

Reading in files

read.csv()

CSV = "comma separated value" – R can not read in .xsl or .xslx files (unless you install extra packages)

myData <- read.csv(*filename*)

You can always read in the *full* path to your file

Possibly useful function:
file.exists()

read.csv("/Users/scottnordstrom/Teaching/r_crdds_2023-05/wos/ClimateAndArt1.csv")

If your csv is in your working directory, you can just put the filename

```
setwd("/Users/scottnordstrom/Teaching/r_crdds_2023-05/wos")

read.csv("ClimateAndArt1.csv")
```

If your csv is in a *subdirectory* of your working directory, you can put the path to the file

```
setwd("/Users/scottnordstrom/Teaching/r_crdds_2023-05")

read.csv("wos/ClimateAndArt1.csv")
```

Read in the first ClimateAndArt CSV

```
# (remember to set your working directory first)
climateArt1 <- read.csv('ClimateAndArt_01.csv')
```

Inspecting the data frame:

dim(climateArt1) will tell us how many rows and columns

names(climateArt1) will show names of each column

Other useful functions:

head(climateArt1) will print out first several rows (you can guess what tail() does)

str(climateArt1) will show us the type stored in each column

Exploring the data frame

What publication types are there in the data frame?

climateArt1$Publication.type *(this will print the whole thing!)*

unique(climateArt1$Publication.type)

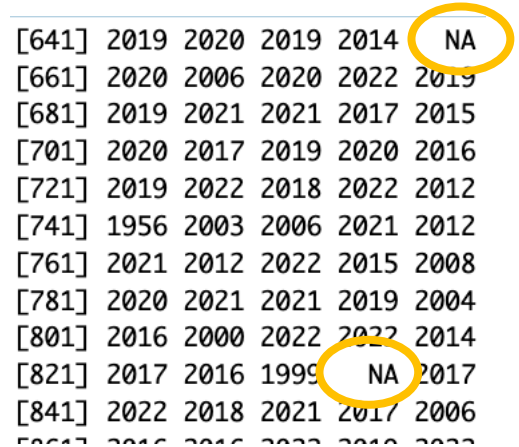
How many of each? – table()

table(climateArt1$Publication.type)

Handling missing data

What is the earliest publication in our dataset?

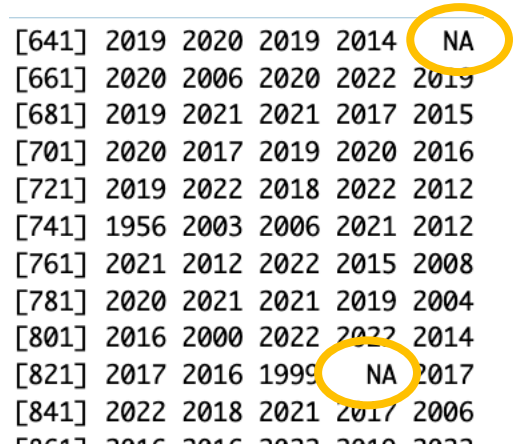
```
min(climateArt1$Publication.Year)
[1] NA
```



NAs can appear for several reasons, including empty cells in a CSV.

For character columns, may appear as empty string ('')

Handling missing data



is.na() – function that returns TRUE in positions with NA, FALSE otherwise

```
is.na(climateArt1$Publication.Year)
[1] FALSE FALSE FALSE FALSE FALSE FALSE ...
```

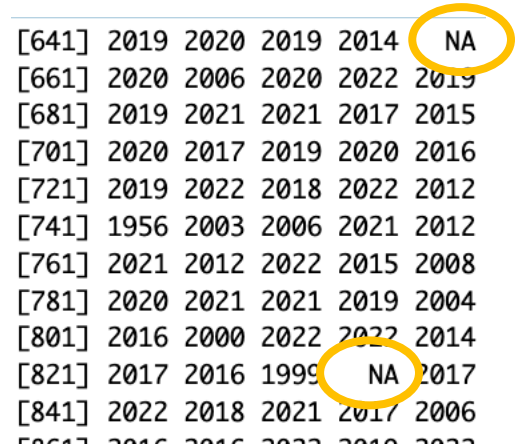
```
sum(is.na(climateArt1$Publication.Year))
[1] 13
```

Remember – logicals are really just 1s and 0s – here we're counting the 1s!

```
which(is.na(climateArt1$Publication.Year))
[1]  78  97 229 284 295 361 449 457 549 599 608 645 824
```

which() will tell you which indices in a logical vector return TRUE

Removing missing data



! Before a logical will negate it, i.e., TRUE to FALSE and FALSE to TRUE

!is.na(climateArt1$Publication.Date) will return TRUE for entries that are *not* NA

climateArt1.subs <- climateArt1[!is.na(climateArt1$Publication.Date,]

Picks out only the *rows* with non-NA dates

For missing character data:

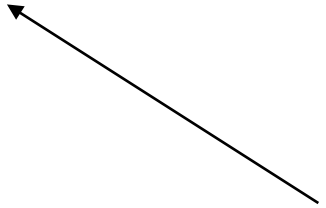
!(climateArt1$Abstract == '') or !(climateArt1$Abstract %in% '')

Combining data frames with rbind()

rbind() will combine data frames into one. ('r' for 'row')

Requires all data frames to have identical column names

```
climateArt <- rbind(
        read.csv(...),
        read.csv(...),
        ...
)
```



There are other, sleeker ways to do this – if you are curious, ask me!

Can you guess what cbind() *does?*

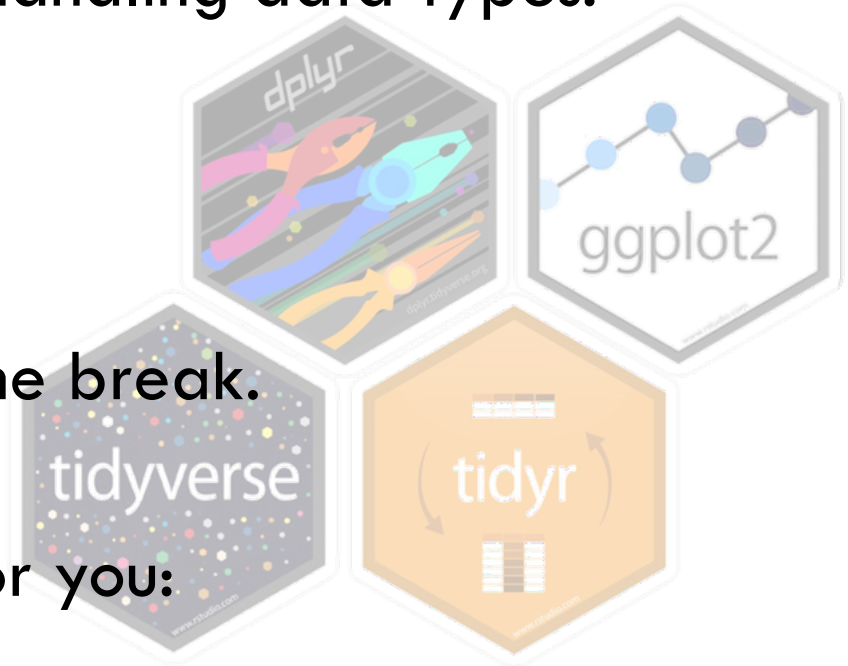
Can you guess what must be true for cbind() *to work?*

Tidy data: working with the tidyverse

Developers have made handy packages for handling data-types.

These are called the tidyverse.

We will get started with the tidyverse after the break.



For now, make sure the following lines work for you:

```
library(dplyr)
library(tidyr)
library(ggplot2)
```

Break!

On packages (“libraries”)

”Base” R is a collection of functions that run on their own.

But sometimes, people figure out ways to do things better, faster, more neatly, etc.

They bundle this code into external ”packages” that you can install and use.

Install packges with install.packages()

install.packages(‘nic’)

This is a package with “nature inspired color palettes”

You only need to install packages *once* (unless you update your version of R)

Don’t include install.packages() *in a script! It will just re-install the package, wasting time.*

Manipulating and preparing data with the tidyverse

- Tidyverse is a collection of packages for manipulating data (and other things)

- "Base" R: confusing, inconsistent hodgepodge of functions

- Tidyverse (ideally): more consistent, coherent structure/organization

Consistent structure means a little bit of knowledge goes a long way

Do I need to use the tidyverse?

No.

Much of what the tidyverse is capable of can be done in base R.

But, it might be slower, messier, more complicated, etc.

Tidyverse pros: powerful, widely used, common syntax means learning new things is easier once you've seen enough

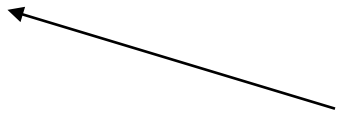
Tidyverse cons: learning curve, occasionally changing (old code might "break")

Getting started with tidyverse

To make the functions in a package accessible, use the library() command to load the package.

Today, we'll use these tidyverse packages: tidyr, dplyr, ggplot2

```
library(dplyr)
library(tidyr)
library(ggplot2)
```



It generally *is* a good idea to include calls to library() in your scripts, usually at the beginning.

You can also say library(tidyverse), which will load *all* of the tidyverse packages at once.

Fundamental tidyverse concept: pipe

Pipe: %>%

object %>% function()

Like saying “take object *and then* put it into function()”

More formally: the object on the left-hand side gets “piped” in as the first argument in function()

object %>% function(argument2, argument 3)

If more than one argument to function(), pass those in after

Simple pipe examples

```
5 %>% sqrt()
```

```
mtcars$cyl %>% as.character()
```

```
mtcars %>%
        nrow() %>%
        sqrt()
```

No temporary objects

Easy to read

or

```
sqrt(5)
```

```
as.character(mtcars$cyl)
```

```
n <- nrow(mtcars)
sqrt(n)
```

```
sqrt(nrow(mtcars))
```

Makes unnecessary variables

Harder to read

Piping usefulness: stringing together operations

```
595  therm %>%
596    filter(Plot %in% 48) %>%
597    filter(!Tag %in% 1117) %>%
598    select(Date, Tag, Infl_spread) %>%
599    spread(Date, Infl_spread)
600  # 1068 is plusible...
601  therm %>%
602    filter(Plot %in% 48) %>%
603    filter(!Tag %in% 1117) %>%
604    select(Date, Tag, Infl_done) %>%
605    spread(Date, Infl_done)
606  # 1068 is the only realistic option
```

Neatly, cleanly perform multiple operations on data frame

No temporary objects made

Easy to read (if you know what you're looking for)

```
640  therm %>%
641    filter(Plot %in% 48) %>%
642    filter(!Tag %in% 1117) %>%
643    select(Date, Tag, Infl_spread, Infl_done) %>%
644    unite(col = Infl, c(Infl_spread, Infl_done), sep = '_') %>%
645    spread(Date, Infl)
```

Another analogy for piping: cake baking

Cake recipe:

- Take flour
- Add eggs, oil, water
- Mix with spoon for two minutes
- Bake at 350 degrees F for 35 minutes
- Let cool

Base R:

```
dough <- add(flour, oil, water)

batter <- mix(dough, utensil =
              'spoon', time = 2)

cake <- bake(batter, temp = 350,
             unit = 'F', time = 35)

cake <- let_cool(cake)
```

dough, batter are made once, never used again

Another analogy for piping: cake baking

Cake recipe:

- Take flour
- Add eggs, oil, water
- Mix with spoon for two minutes
- Bake at 350 degrees F for 35 minutes
- Let cool

Base R:

```
cake <- bake(mix(add(flour, oil, water),
   utensil = 'spoon', time = 2)
            temp = 350, unit = 'F',
            time = 35)
)

cake <- let_cool(cake)
```

this really hard to read!

Another analogy for piping: cake baking

Cake recipe:

- Take flour
- Add eggs, oil, water
- Mix with spoon for two minutes
- Bake at 350 degrees F for 35 minutes
- Let cool

Piping in tidyverse:

```
flour %>%
  add(eggs, oil, water) %>%
  mix(utensil = 'spoon', time = 20) %>%
  bake(temp = 350, unit = 'F', time = 35) %>%
  let_cool()
```

Tidyverse functions: data is first argument

- mutate(data, columns): add a new column(s) to a data frame

```
mutate(cars, speed.sq = speed^2)
```

- select(data, columns): selects column(s) from data frame

```
select(mtcars, mpg, wt, vs, am)
```

```
mtcars %>%
        select(mpg, wt, vs, am)
```

Add a column for weight in kg, then give me only mpg and weight in kilograms columns

```
mtcars %>%
        mutate(wtkg = wt*907.185) %>%
        select(mpg, wt.kg)
```

More tidyverse functions

- filter(data, logical): return only rows matching the logical

```
filter(cars, speed > 20)
```

- rename(data,

  newname = oldname,

  ...

): selects column(s) from data frame

```
mtcars %>%
        rename(
          transmission = am,
    engine = vs
        )
```

Renames the old am *column to* transmission*, renames old* vs *column to* engine

Grouping with tidyverse

dplyr functions allow you to perform operations on *groups* of data

group_by(variables) to group

summarise(), mutate(), etc. to operate

Base R equivalent is aggregate()

```
mtcars %>%
 group_by(am, vs) %>%
 summarise(
   mean.mpg = mean(mpg),
   sample.size = n(),
   se.mpg = sd(mpg) / sqrt(sample.size)
 )
```

	am	vs	mean.mpg	sample.size	se.mpg
1	0	0	15.0	12	0.801
2	0	1	20.7	7	0.934
3	1	0	19.8	6	1.64
4	1	1	28.4	7	1.80

High quality plots with ggplot

Base R does have plotting capabilities (with plot(), hist(), etc.)

However, customizing these plots can be difficult.

Many people instead use the package ggplot2.

Load the ggplot library with

library(ggplot2)

The syntax of ggplot

All ggplot plots start with a call to the ggplot() function

As with much of the tidyverse, the first argument is the data frame you want to plot.

```
ggplot(data = artClimate)

artClimate %>% ggplot()
```

Either of these will work!

The syntax of ggplot

Add features to your ggplot object with the plus sign (+)

Most of these features will be geometric (lines, points, shapes).

Functions to do this start with geom_, e.g.,

- geom_point()
- geom_line()
- geom_histogram()

aes() – you need this to extract features for your plots!

Columns from your data frame that specify features of the plot.

Usual suspects:

x

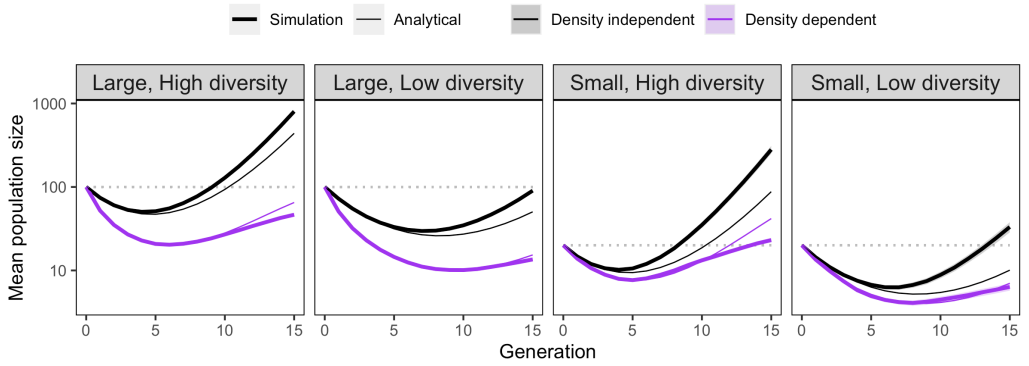
y

colour

shape

...

One or both of these is often used in aes()!




```
geom_line(
  aes(
        x = Generation,
        y = Growth.rate,
        colour = Density.dependence
      )
)
```

aes() goes *inside* geom_line() (not on its own) because it affects our lines

X, Y, and color of date that forms my line come from these columns

Example with our dataset:

Looking at citation – do longer journal articles get cited more?

```
climateArtArticles <– climateArt %>%
        # Select only journal articles
        filter(Publication.Type == 'J') %>%
        # Remove retractions, pre-prints, etc.
        filter(Document.Type == 'Article')
```

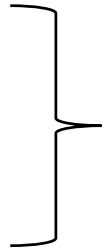
Let's first subset the journal articles.

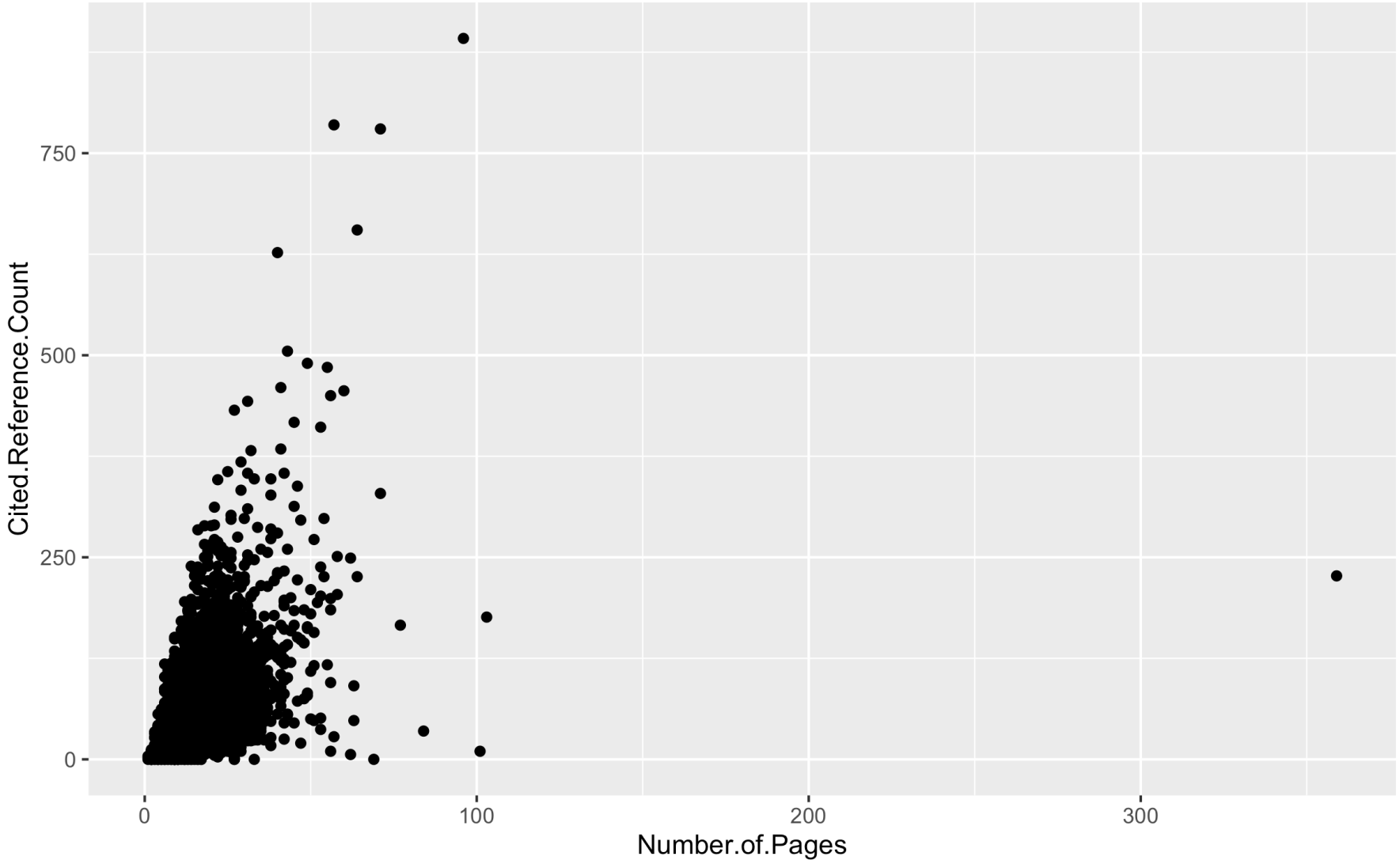
```
nrow(climateArtArticles)
[1] 8994
```

To plot citations over time:

```
climateArtArticles %>%
  ggplot() +
  geom_point(
        aes(x =  Number.of.Pages,
           y = Cited.Reference.Count)
        )
```

x and y inside aes() tell us what the *x* and *y* coordinates of our points should be!



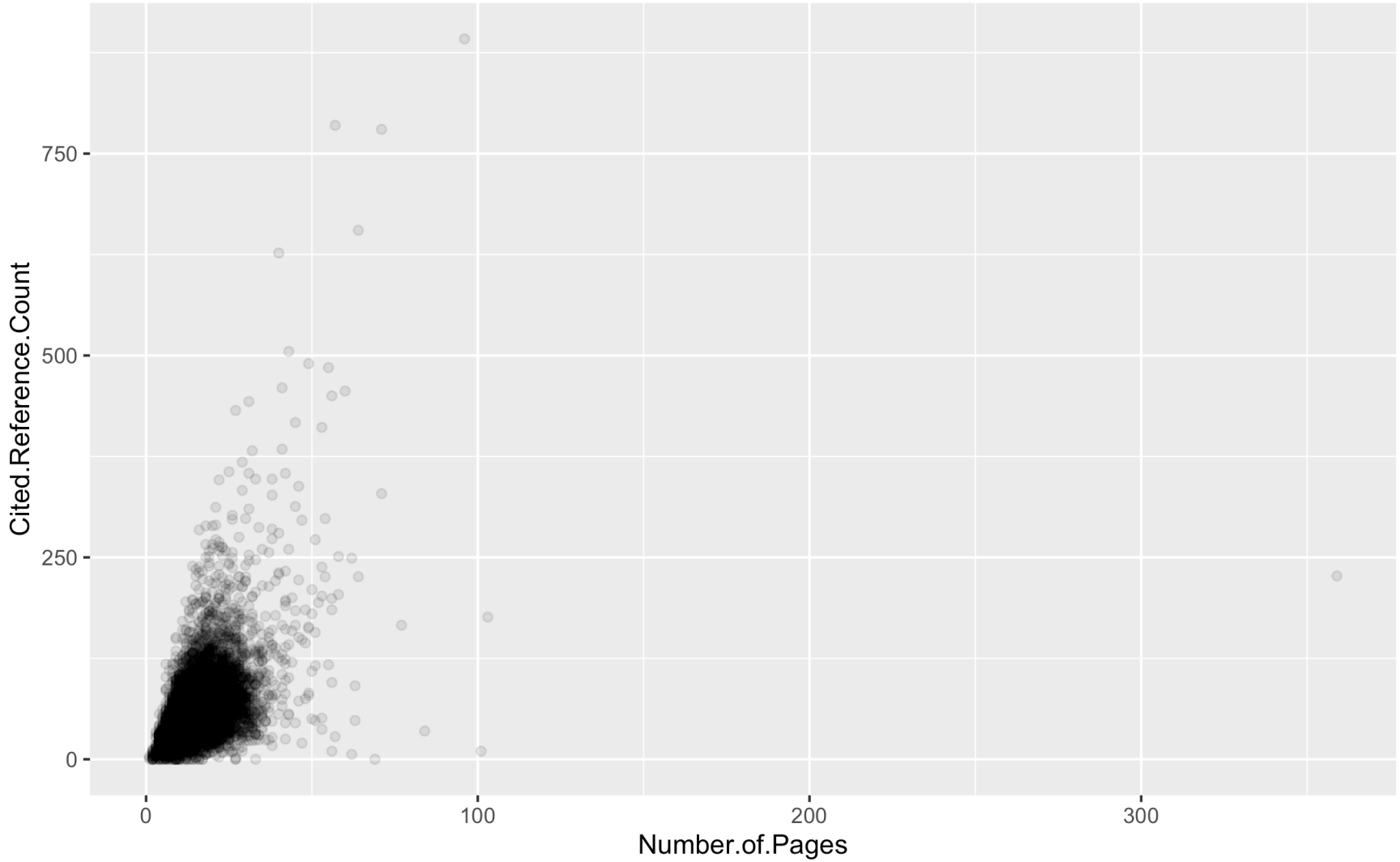


Interesting! Points are hard to see though…

alpha (transparency) – make it low and apply it to *all points*

```
climateArtArticles %>%
  ggplot() +
  geom_point(
        aes(x =  Number.of.Pages,
           y = Cited.Reference.Count),
           alpha = 0.1
        )
```

Outside of aes() because we don't want alpha to depend on any column!



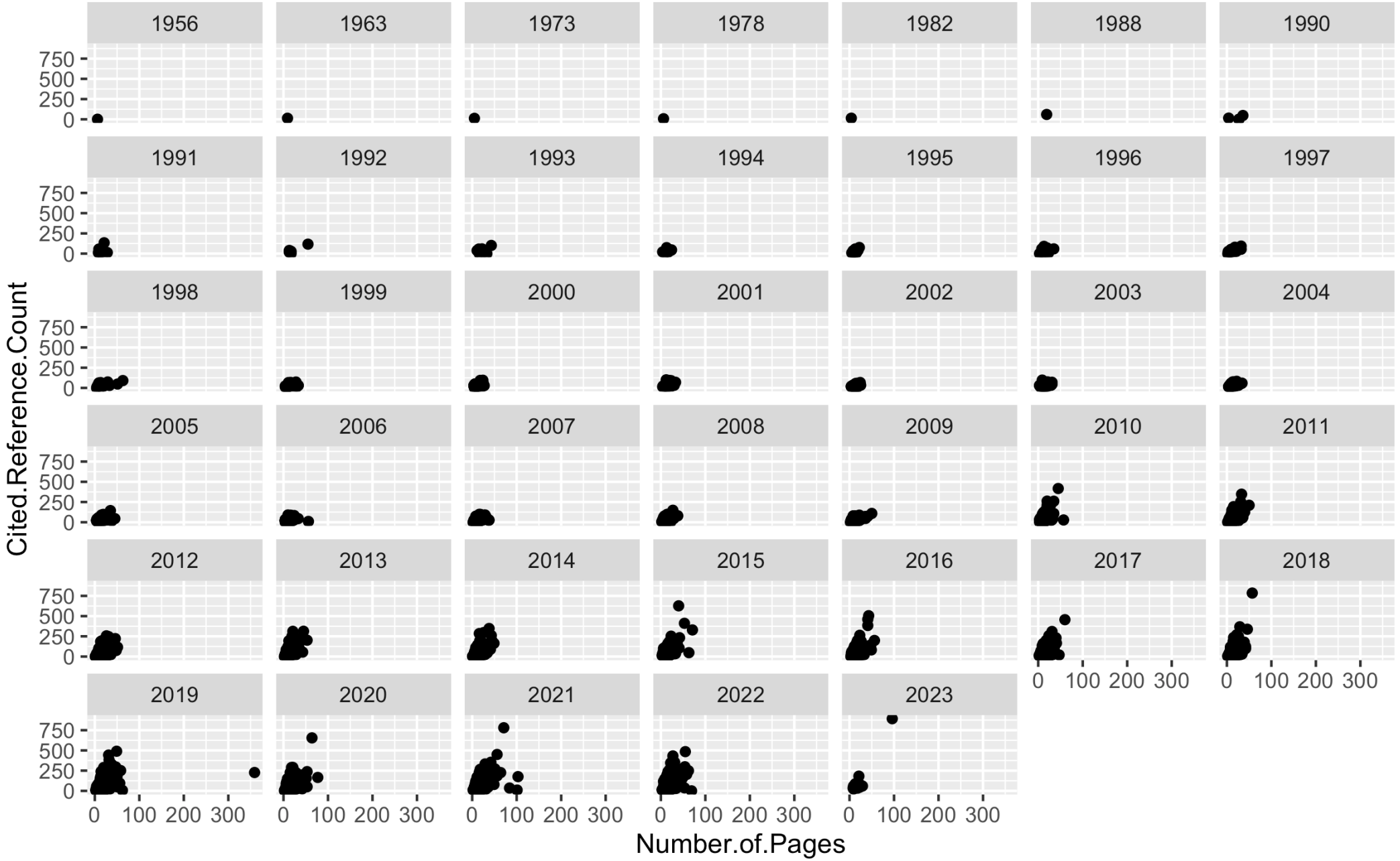
Thicker = more points

Does this vary by year?

Use facet_wrap() to build sub-plots, broken out according to one of your columns

```
climateArts %>%
  ggplot() +
  geom_point(
        aes(
    x = Number.of.Pages,
    y = Cited.Reference.Count
        )
  ) +
  facet_wrap(~ Publication.Year)
```

Makes one subplot (facet) for each year


1956
1963
1973
1978
1982
1988
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
0
250
500
750
0
100
200
300
Cited.Reference.Count
Number.of.Pages

More ggplot functions available at…

https://ggplot2.tidyverse.org/