Welcome to AEM 2850 / 5850!

Week 1

AEM 2850 / 5850 : R for Business Analytics

Cornell Dyson

Spring 2024

Acknowledgements: Andrew Heiss, Claus Wilke, Grant McDermott

Plan for today

Why take R for Business Analytics?

Summary of key class details

Teaser example

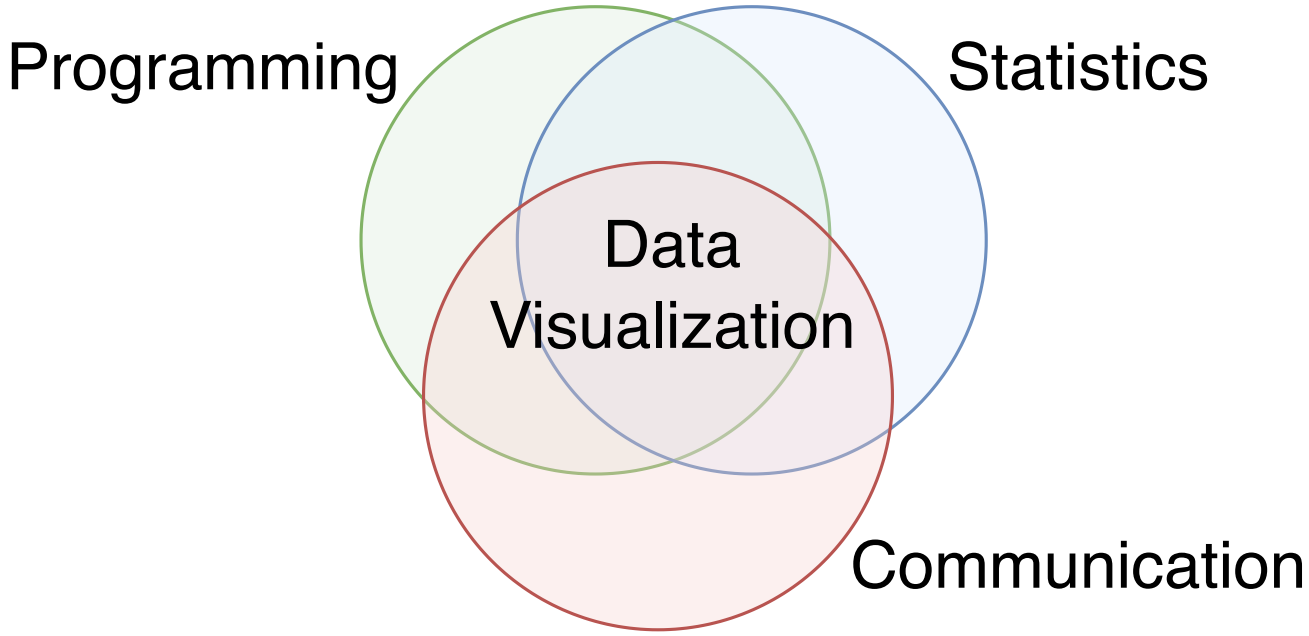
Just show me the data!

What makes a great visualization?

Basic base R (time permitting)

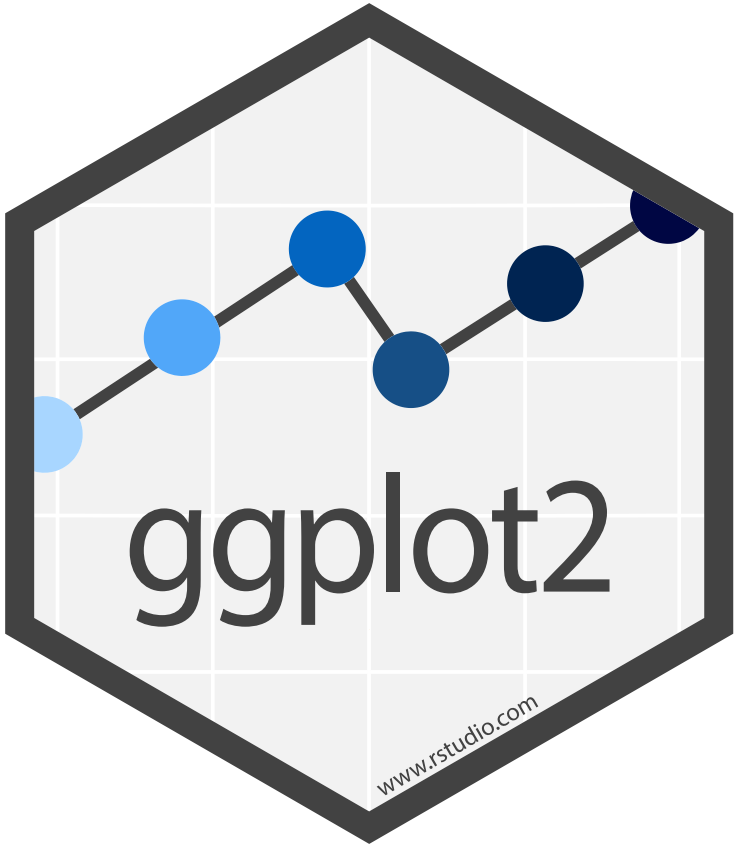
Why take R for Business Analytics?

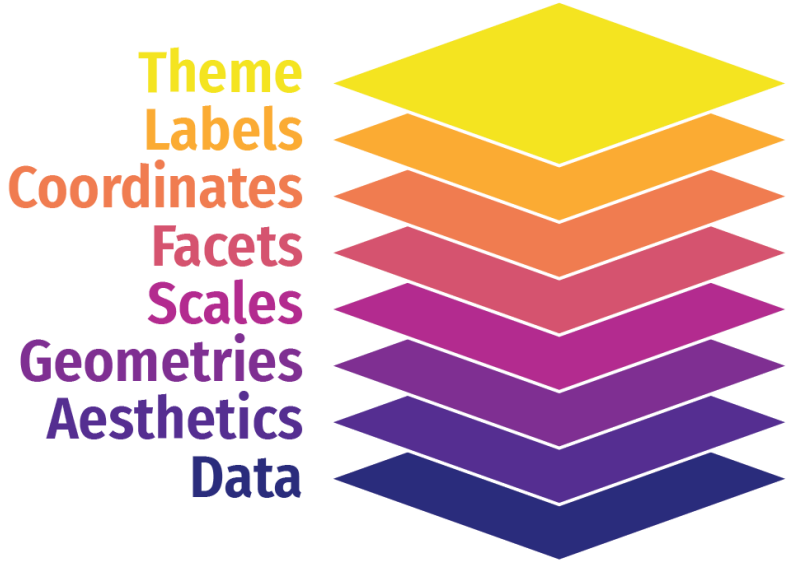
Why take R for Business Analytics?



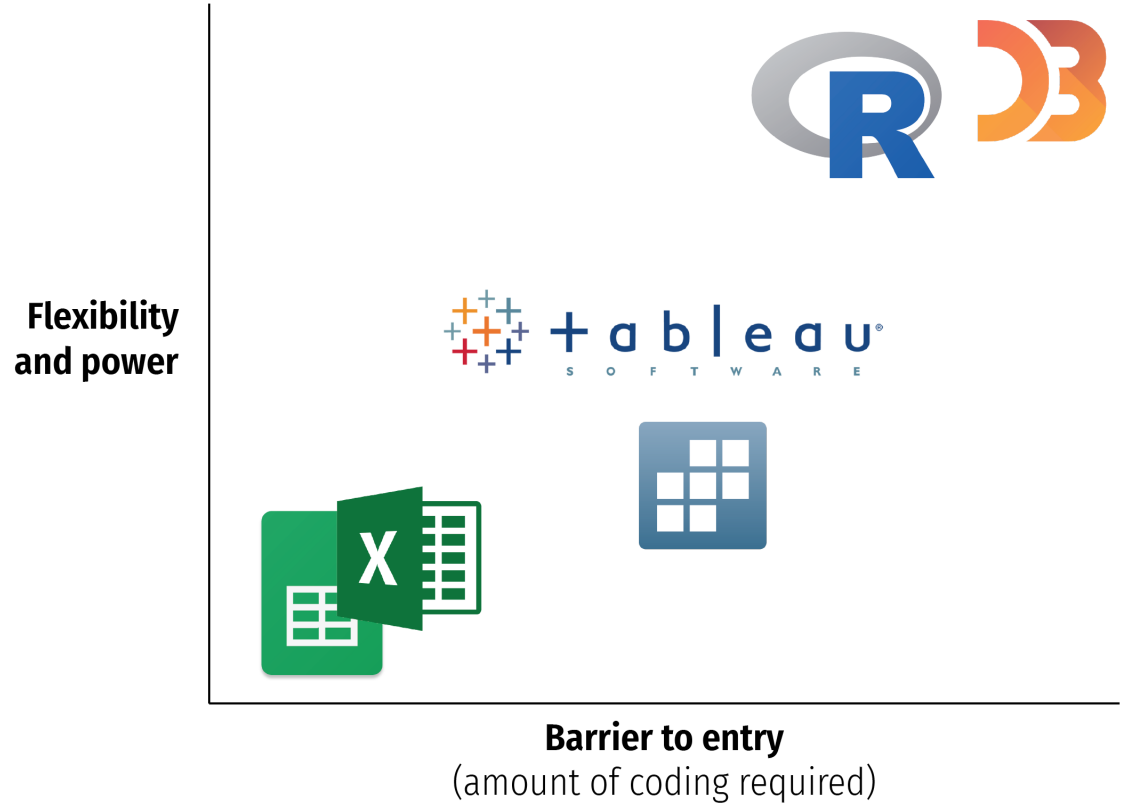
Why R for Business Analytics?

Why R for Data Visualization?





Why R for Data Visualization?



Why R for Life?

Practical tool that could help you get a job and then do said job

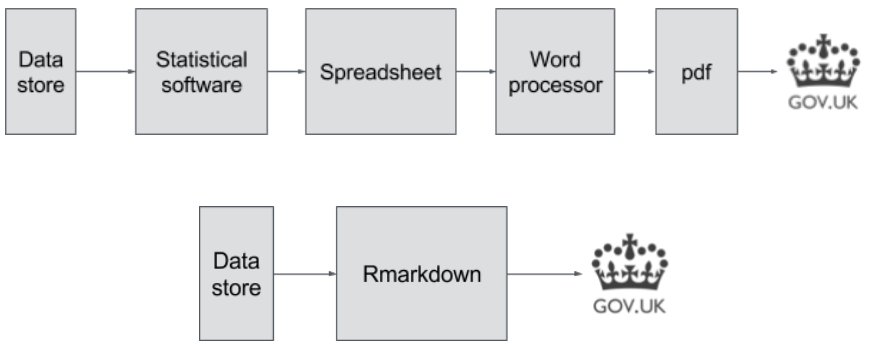
3.1.2 Data Visualization

We use ggplot2 as our main package to create ad-hoc exploratory graphics as well as polished-looking customized visualizations. When combined with tools to clean and transform data, ggplot2 allows analysts to quickly translate insights into high quality, compelling visualizations. In addition to the static graphics of `ggplot2`, we often make interactive visualizations or dashboards using R packages such as `plotly` (Sievert et al. 2017), `leaflet` (Cheng et al. 2017), `dygraphs` (Vanderkam et al. 2017), `DiagrammeR` (Sveidqvist et al. 2017), and `shiny` (Chang et al. 2017).

3.1.3 Reproducible Research

At Airbnb, all R analyses are documented in `rmarkdown`, where code and visualizations are combined within a single written report. Posts are carefully reviewed by experts in the content area and techniques used, both in terms of methodologies and code style, before publishing and sharing with the business partners. The peer review process is

Airbnb, ggplot, and rmarkdown



The UK's reproducible analysis pipeline

Why R for Life?

Practical tool that could help you get a job and then do said job

Or start making money now!

Why R for Life?

Practical tool that could help you get a job and then do said job

Open source

Huge community of users and package developers

Here are a few examples of other things you can do using R:

- Make slides like the ones you're looking at right now
- Build websites like our course site
- Write books like R for Data Science
- Make interactive web apps

Skills from this course can also be used for other programming languages

Class details

Preface

1. Your success in this class is important to me

2. This course is a work in progress

3. Get the semester off to a good start: **read the syllabus**!

A bit about me

- Prof. Todd Gerarden
- Economist
- Came to Cornell in 2018
- Interested in:
  - Energy economics
  - Climate tech
  - Working with data

A bit about our TAs

Graduate TA

Victor Simoes Dornelas

Undergraduate TAs

Jonathan Gotian

We will post office hours and contact info on the course site and canvas

A bit about you

Do you have any programming experience? (None is required or even expected!)

What programming language(s) have you used before?

- R
- Python
- SQL
- VBA
- MATLAB
- Stata
- Other

First course assignment will be to fill out a survey to tell us more about you

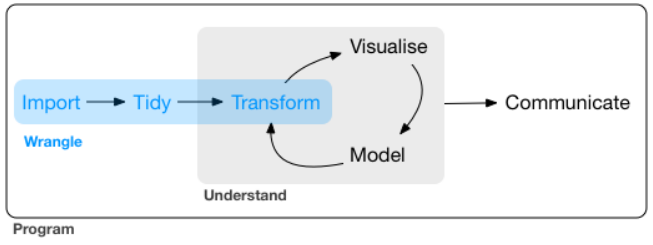
Course objectives

1. Develop basic proficiency in R programming
2. Understand data structures and manipulation
3. Describe effective techniques for data visualization and communication
4. Construct effective data visualizations
5. Utilize course concepts and tools for business applications

Plan for the semester

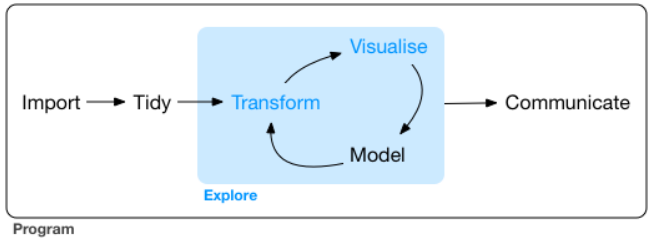
Programming Foundations

R, RStudio, Quarto, the tidyverse



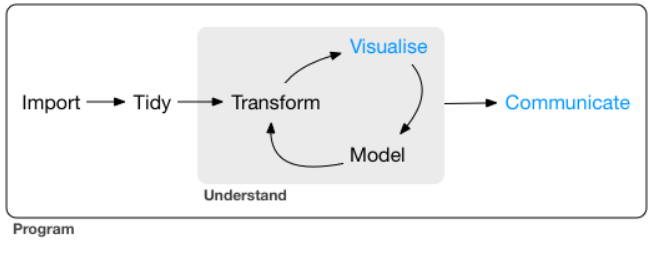
Data Visualization Foundations

the grammar of graphics, ggplot2



Special Topics

annotations, time, space, etc.



Plan for each week

We will follow the same general process each week:

- Do readings listed on the course site before Tuesday (example: Week 1)
- **Tuesday:** come to class, where we will discuss material for that week's topic
- **Thursday:** come to class, where we will work through hands-on examples
- Work on the lab, attend office hours as needed
- **Monday (following):** submit lab on canvas by 11:59pm (starting with Week 1)

Assignments

- **Labs** are short weekly homework assignments to practice programming
- **Prelims** are intended to assess programming and data visualization proficiency
- The **group project** is intended to synthesize and reinforce skills in real-world applications
- **Class participation** is the best way to learn the material, attendance and completion of in-class examples is expected

Assignment	Percent
Labs	35%
Prelim 1	20%
Prelim 2	20%
Group project	20%
Class participation	5%
Total	100%

Contacting us

Office hours:

- TAs: TBD
- Tuesdays 11:30am - 12:30pm: Prof. Gerarden in Warren 464
- Other times by appointment: Prof. Gerarden, at aem2850.youcanbook.me

Email:

You can also reach us by email. The best approach is to email both me and our grad TA Victor Simoes Dornelas at the same time. You can do that with one click here. **Please** read the syllabus for tips on how to make the most of email.

Course websites

Site for accessing course materials: (↓)

aem2850.toddgerarden.com

Site for submitting work: (↑)

canvas.cornell.edu/courses/62697

- viewing announcements
- viewing grades
- you can also view and navigate the course site through canvas

Sucking

> "The bad news is whenever you're learning a new tool, for a long time you're going to suck. It's going to be very frustrating.
>
> But, the good news is that that is typical, it's something that happens to everyone, and it's only temporary...
>
> Remember, when you're getting frustrated, that's a good thing, that's temporary, keep pushing through, and in time [it] will become second nature."

Hadley Wickham, author of `ggplot2`, *R for Data Science*, and much more

I *know* you can succeed in this class. Don't hesitate to get help from me, TAs, office hours, and your peers.

Questions about the class?

Teaser example

How does 2024 compare to 2023 so far?

Go to aem2850.toddgerarden.com/content/01-content

Click the links to download the following files:

- Weather stations in NY
- Weather in NY in 2023
- Weather in NY in 2024

Let's make a plot that compares the evolution of daily max temps (TMAX) over January in 2023 and 2024

Time permitting: do it on your own using software of your choice

How does 2024 compare to 2023 so far?

One way to do this in R. First, we'll need to import and prep the data:

```
# load packages
library(tidyverse); library(lubridate)
# identify the Cornell station
stations <- read_csv("data/01-slides/ny-stations.csv")
cornell <- stations |> filter(str_detect(NAME, "CORNELL"))
# read in and bind relevant data
clean_data <- function(y, s, m) {
  str_glue("data/01-slides/ny-weather-", y, ".csv") |>
    read_csv() |>
    inner_join(s, by = "STATION") |>
    mutate(date = mdy(DATE),
           mon = month(date),
           day = day(date),
           year = year(date)) |>
    filter(mon == m)
}
years <- c(2023, 2024)
cornell_temps <- map(years, clean_data, cornell, 1) |> bind_rows()
```

How does 2024 compare to 2023 so far?

What do the data look like?

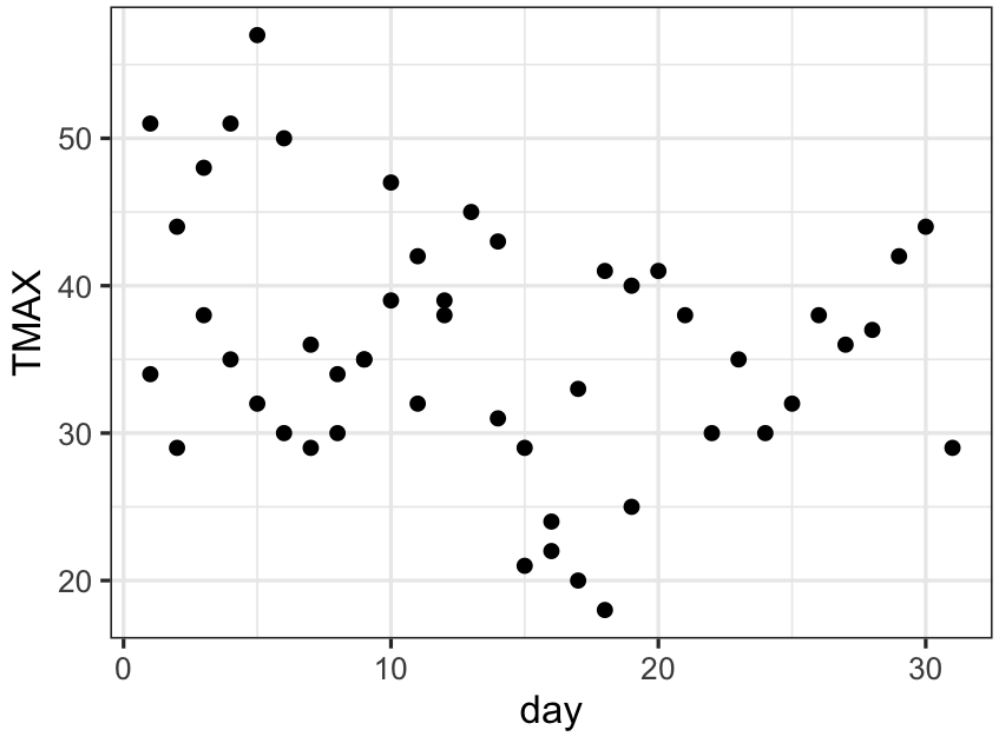
```
head(cornell_temps)
```

```
## # A tibble: 6 × 19
##   STATION     DATE    DAPR  MDPR  PRCP  SNOW  SNWD  TAVG  TMAX  TMIN  TOBS NAME
##   <chr>       <chr>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
## 1 USC00304174 1/1/23    NA    NA  0.01     0     0    NA    51    38    38 ITHA…
## 2 USC00304174 1/2/23    NA    NA  0        0     0    NA    44    38    39 ITHA…
## 3 USC00304174 1/3/23    NA    NA  0        0     0    NA    48    29    41 ITHA…
## 4 USC00304174 1/4/23    NA    NA  0.4      0     0    NA    51    39    40 ITHA…
## 5 USC00304174 1/5/23    NA    NA  0.2      0     0    NA    57    34    35 ITHA…
## 6 USC00304174 1/6/23    NA    NA  0        0     0    NA    50    31    34 ITHA…
## # ℹ 7 more variables: LATITUDE <dbl>, LONGITUDE <dbl>, ELEVATION <dbl>,
## #   date <date>, mon <dbl>, day <int>, year <dbl>
```

How does 2024 compare to 2023 so far?

```
# plot data
cornell_temps |>
  ggplot(aes(x = day,
             y = TMAX)) +
  geom_point() +
  theme_bw()
```

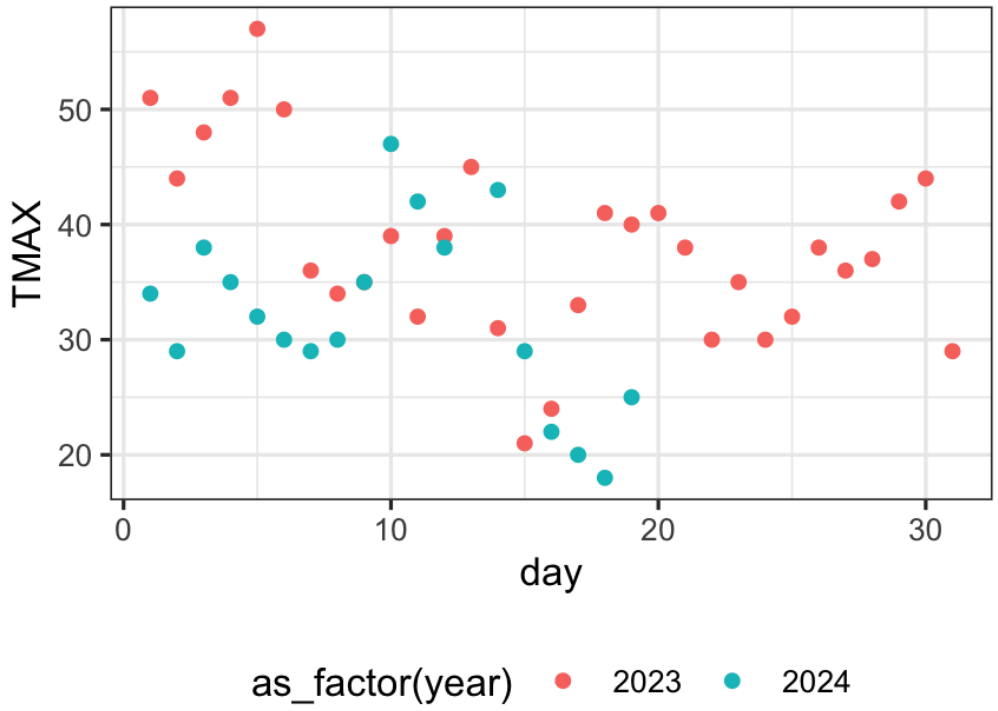
What's wrong with this plot?



How does 2024 compare to 2023 so far?

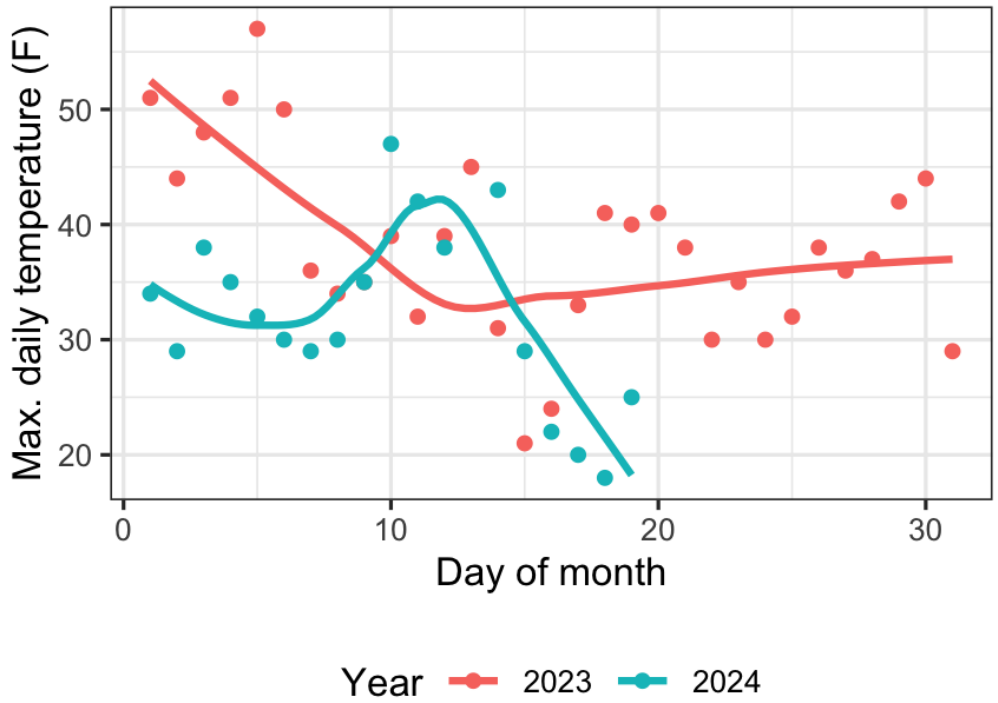
```
# plot data
cornell_temps |>
  ggplot(aes(x = day,
             y = TMAX,
             color = as_factor(year))) +
  geom_point() +
  theme_bw() +
  theme(legend.position = "bottom")
```

What's wrong with this plot?



How does 2024 compare to 2023 so far?

```
# plot data
cornell_temps |>
  ggplot(aes(x = day,
             y = TMAX,
             color = as_factor(year))) +
  geom_point() +
  geom_smooth(se = FALSE) +
  theme_bw() +
  theme(legend.position = "bottom") +
  labs(x = "Day of month",
       y = "Max. daily temperature (F)",
       color = "Year")
```



How does 2024 compare to 2023 so far?

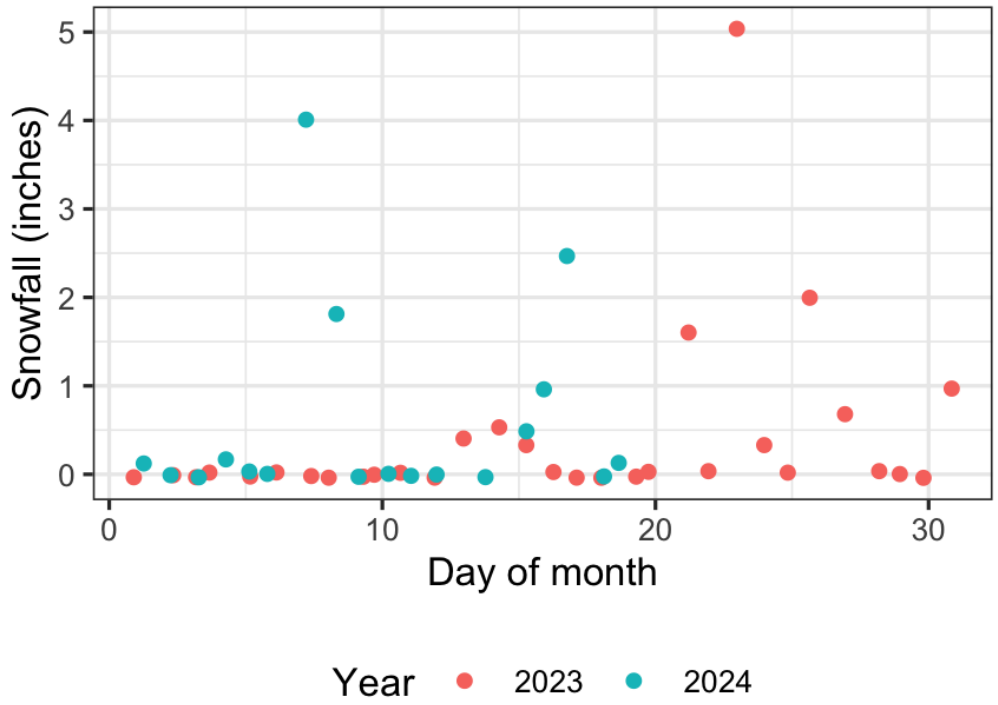
This approach has two advantages over manually creating figures using software such as excel or sheets:

1. we have a script to **reproduce** our work / share our methods with others
2. we can **generalize** and **scale** this much more easily than manual approach

How does 2024 compare to 2023 so far?

For example we can easily **generalize** this approach to other weather outcomes:

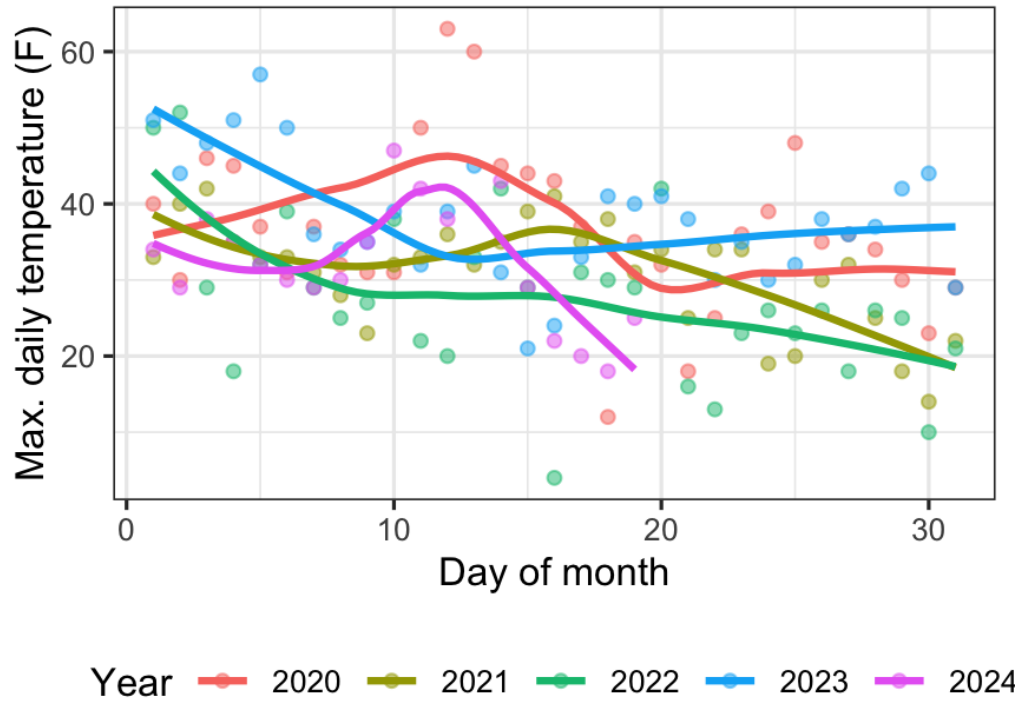
```
# plot data
cornell_temps |>
  ggplot(aes(x = day,
             y = SNOW,
             color = as_factor(year))) +
  geom_jitter() +
  theme_bw() +
  theme(legend.position = "bottom") +
  labs(x = "Day of month",
       y = "Snowfall (inches)",
       color = "Year")
```



How does 2024 compare to 2023 so far?

For example we can easily **scale** this approach to more years:

```
more_years <- 2020:2024
more_temps <- map(more_years, clean_data,
                  cornell, 1) |>
  bind_rows()
# use old code to plot new data!
more_temps |>
  ggplot(aes(x = day,
             y = TMAX,
             color = as_factor(year))) +
  geom_point(alpha = 0.5) +
  geom_smooth(se = FALSE) +
  theme_bw() +
  theme(legend.position = "bottom") +
  labs(x = "Day of month",
       y = "Max. daily temperature (F)",
       color = "Year")
```



Just show me the data!

Just show me the data!

Data is very powerful, but raw data is not usually enough

```
cornell_temps |>
  group_by(year) |>
  summarize(mean_max = mean(TMAX))
```

```
## # A tibble: 2 × 2
##    year mean_max
##   <dbl>    <dbl>
## 1  2023     38.2
## 2  2024     32
```

What's wrong with this calculation?

```
cornell_temps |>
  group_by(day) |>
  filter(n() != 1) |>
  group_by(year) |>
  summarize(mean_max = mean(TMAX))
```

```
## # A tibble: 2 × 2
##    year mean_max
##   <dbl>    <dbl>
## 1  2023     39.2
## 2  2024     32
```

Just show me the data!

Here's another example:

```
head(my_data, 10)
```

```
## # A tibble: 10 × 2
##        x     y
##    <dbl> <dbl>
##  1  55.4  97.2
##  2  51.5  96.0
##  3  46.2  94.5
##  4  42.8  91.4
##  5  40.8  88.3
##  6  38.7  84.9
##  7  35.6  79.9
##  8  33.1  77.6
##  9  29.0  74.5
## 10  26.2  71.4
```

```
mean(my_data$x)
```

```
## [1] 54.26327
```

Seems reasonable

```
mean(my_data$y)
```

```
## [1] 47.83225
```

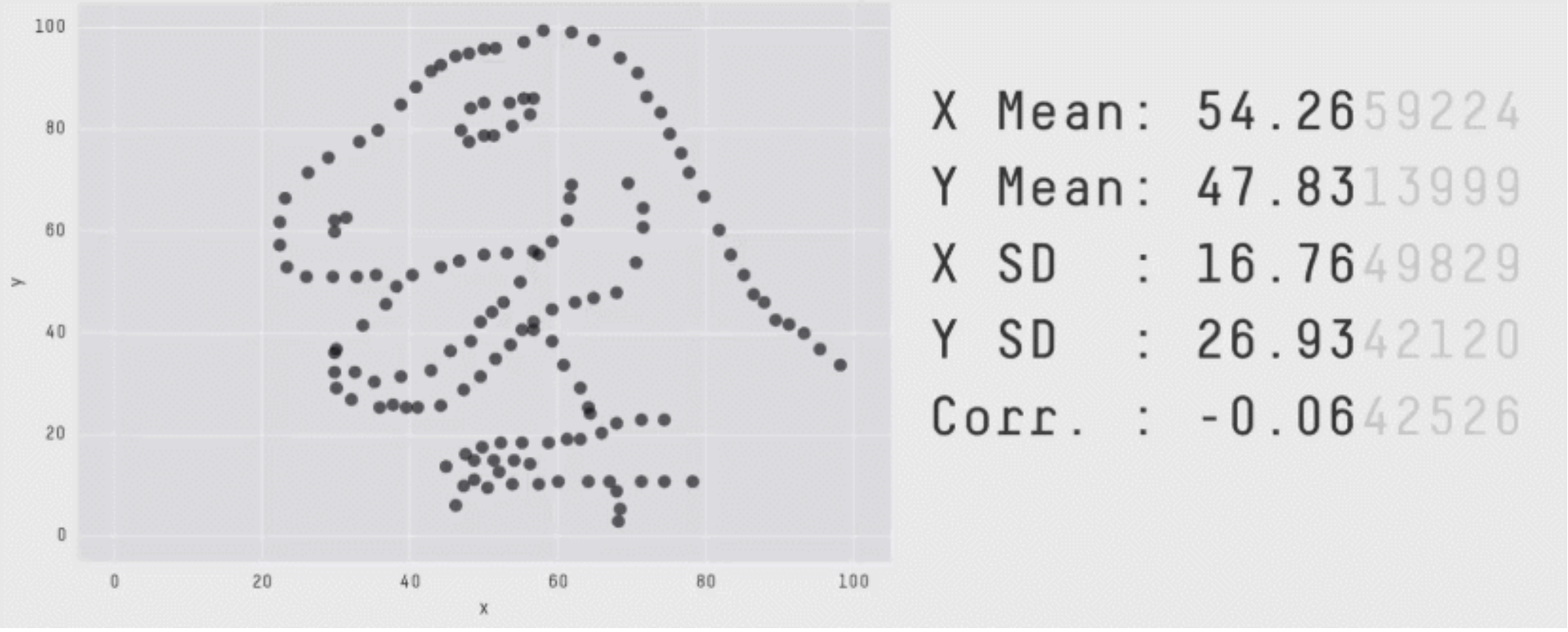
Seems reasonable

```
cor(my_data$x, my_data$y)
```

```
## [1] -0.06447185
```

No correlation

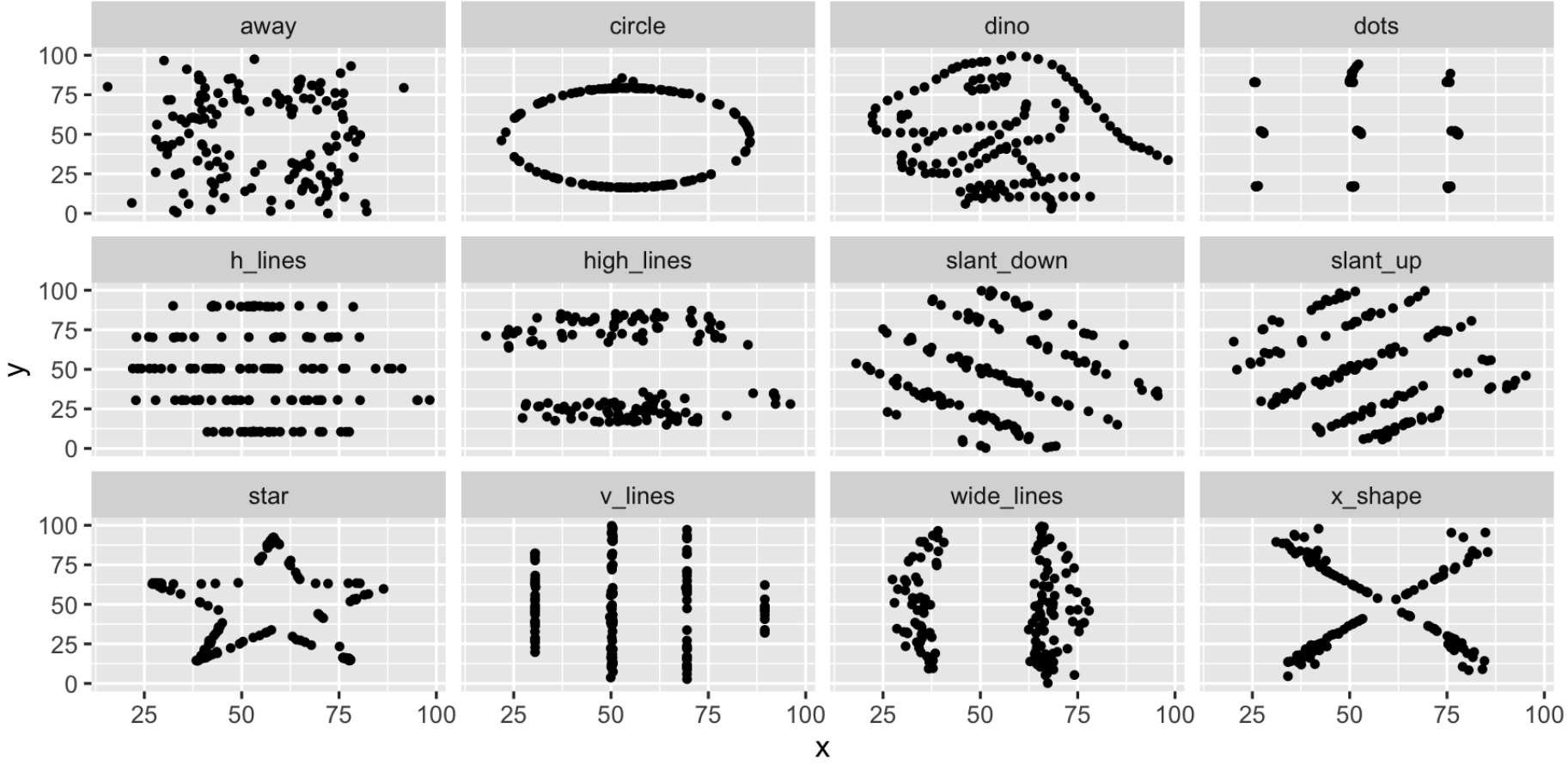
Oh no!



The Datasaurus Dozen

Raw data is not enough

Each of these has the same mean, standard deviation, variance, and correlation



What makes a great visualization?

What makes a great visualization?

Truthful

Functional

Beautiful

Insightful

Enlightening

Alberto Cairo, *The Truthful Art*

What makes a great visualization?

"Graphical excellence is the **well-designed presentation of interesting data**—a matter of substance, of statistics, and of design ... [It] consists of complex ideas communicated with clarity, precision, and efficiency. ... [It] is that which **gives to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space** ... [It] is nearly always multivariate ... And graphical excellence requires **telling the truth about the data**."

Edward Tufte, *The Visual Display of Quantitative Information*, p. 51

What makes a great visualization?

Good aesthetics

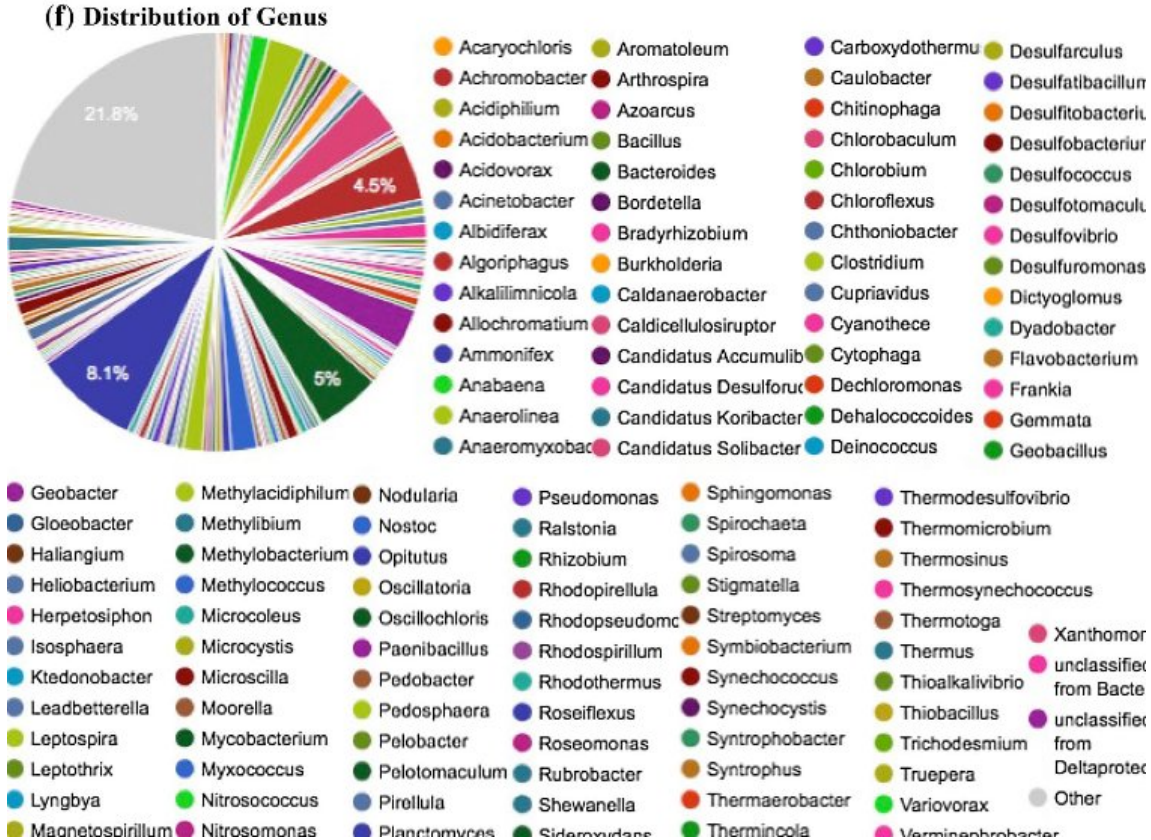
No substantive issues

No perceptual issues

Honesty + good judgment

Kieran Healy, *Data Visualization: A Practical Introduction*

What's wrong?



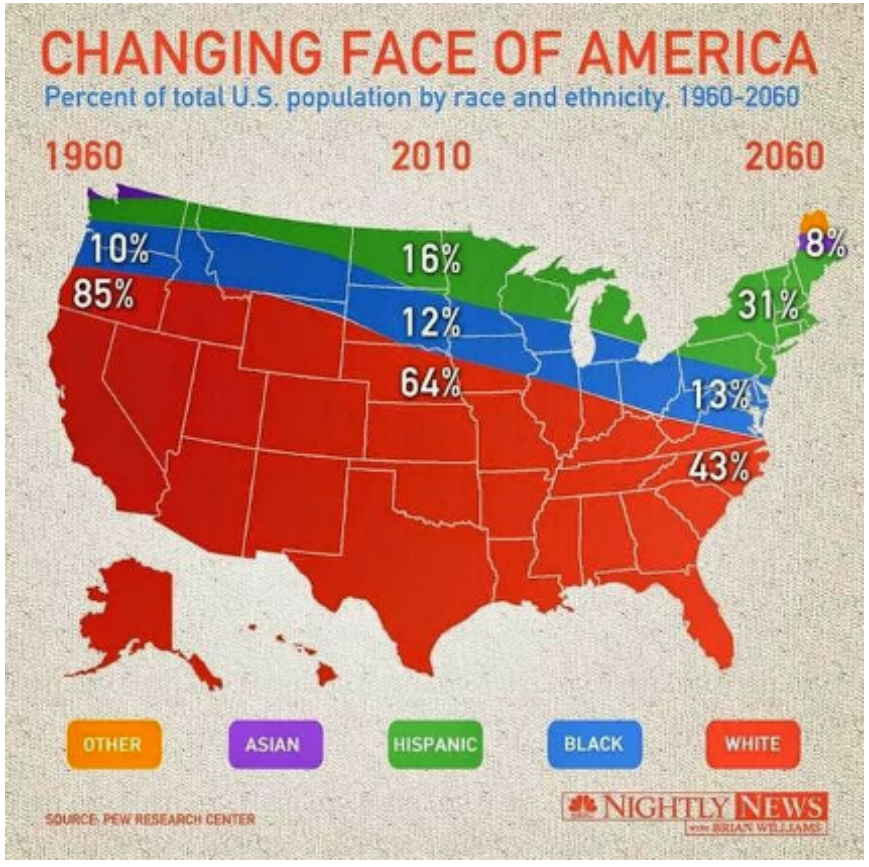
Good aesthetics?

No substantive issues?

No perceptual issues?

Honesty + good judgment?

What's wrong?



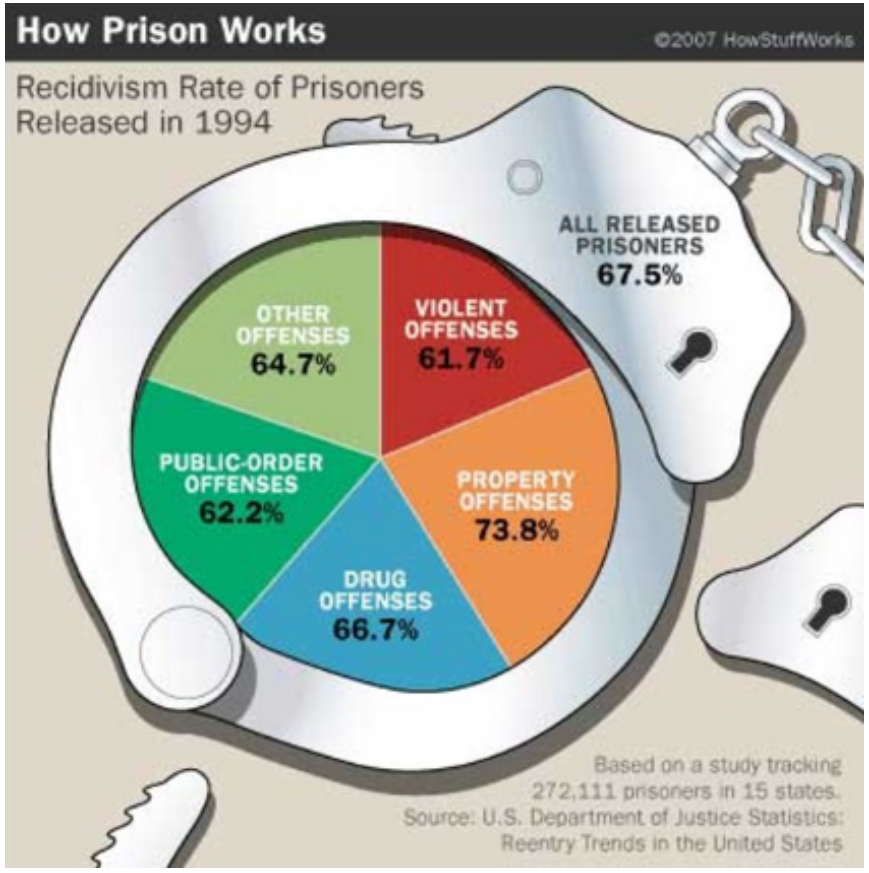
Good aesthetics?

No substantive issues?

No perceptual issues?

Honesty + good judgment?

What's wrong?



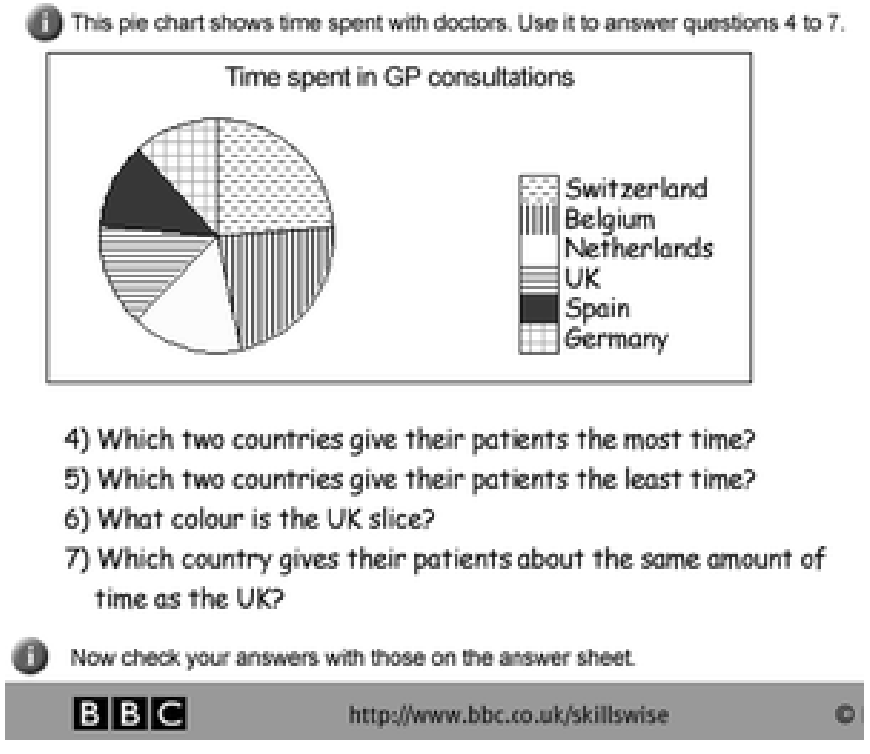
Good aesthetics?

No substantive issues?

No perceptual issues?

Honesty + good judgment?

What's wrong?



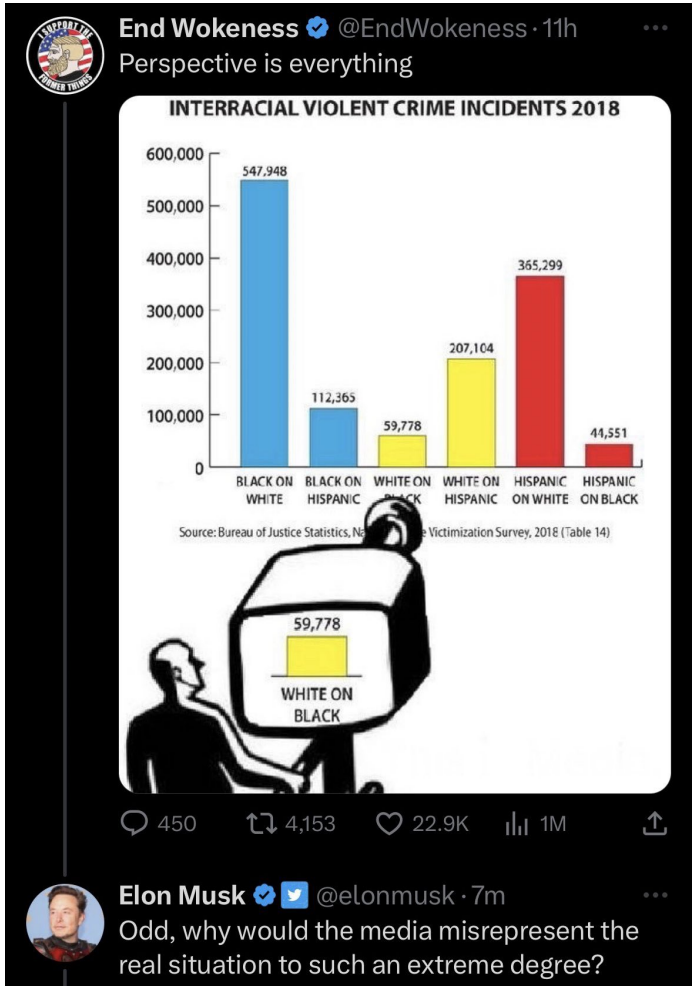
Good aesthetics?

No substantive issues?

No perceptual issues?

Honesty + good judgment?

What's wrong?



Good aesthetics?

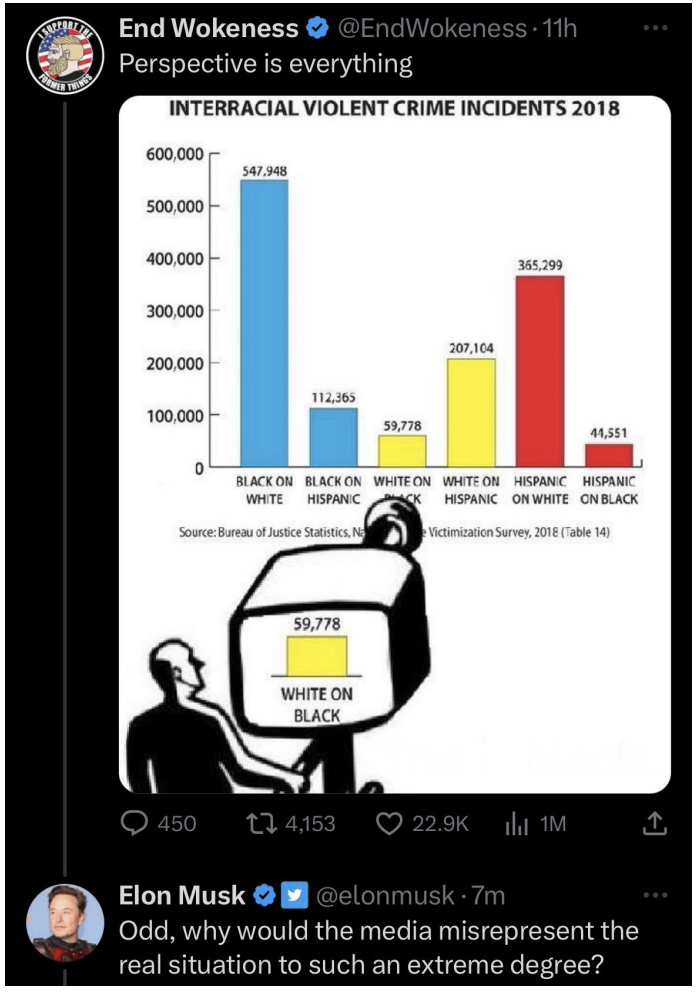
No substantive issues?

No perceptual issues?

Honesty + good judgment?

None of the above?

What's wrong?



Missing context: base rate fallacy (most Americans are white)

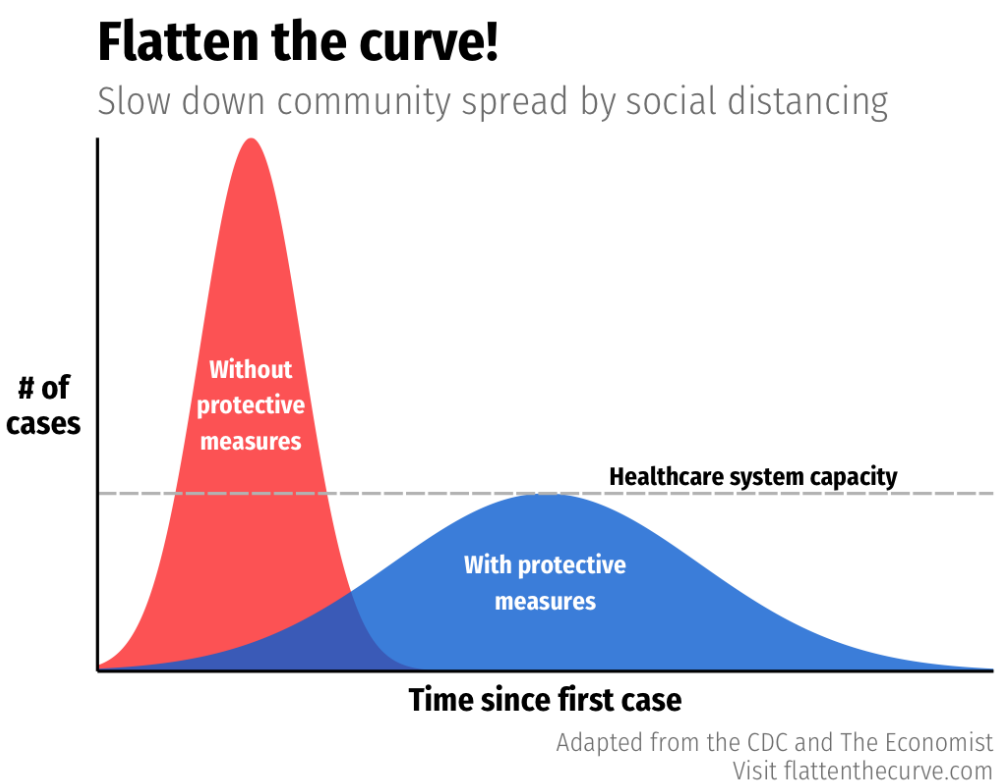
Missing context: omits same-race crime

Comparisons confounded by differences in age, wealth, etc.

Measurement of crime may be biased

Source: Kareem Carr (click for more)

What's right?



Thread by Carl T. Bergstrom

Plan for the rest of this week

Office hours:

- Tuesdays 11:30am - 12:30pm: Prof. Gerarden in Warren 464
- Other times by appointment: Prof. Gerarden, at aem2850.youcanbook.me

Thursday:

- Intro to `R`, `RStudio`, and `R Markdown` / `Quarto`
- You will need your computer for coding exercises every Thursday
- See canvas announcement for instructions to get set up on posit.cloud

Plan for the rest of today

The plan for the rest of today is to introduce ourselves to **base** R

Introduction to base R

Object-oriented programming in R

"Everything is an object"

Reference material (cut for time):

- "Everything has a name" (reserved words and namespace conflicts)
- Indexing
- Cleaning up

Introduction to base R

(Some of this is just for reference, since we also cover it in example-01)

Basic arithmetic

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

```
1+2 # add / subtract
```

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

```
5/2 # divide
```

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

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

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

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 TRUE
```

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

Logic

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

To see whether an object is contained in a list of items, use `%in%`:

```
1:10
```

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

```
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 will happen if we evaluate `0.1 + 0.2 == 0.3`?

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

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

Problem: Computers represent numbers as binary (i.e., base 2) floating-points

- Fast and memory efficient, but can lead to unexpected behavior
- Similar to how decimals can't capture some fractions (e.g., $\frac{1}{3} = 0.3333...$)

Solution: Use `all.equal()` for evaluating floats (i.e., fractions)

```
all.equal(0.1 + 0.2, 0.3)
```

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

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 with =

You can also use = for assignment.

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

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

Which assignment operator should you use?

Many R users prefer <−, inserted using the keyboard shortcut Alt/Option + -

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

```
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
winter <- "ski season" # iykyk
```

Comments should be concise (unlike above)

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

Keyboard shortcut: use `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 or google sheet

```
# 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 x and y

Aside: built-in data frames

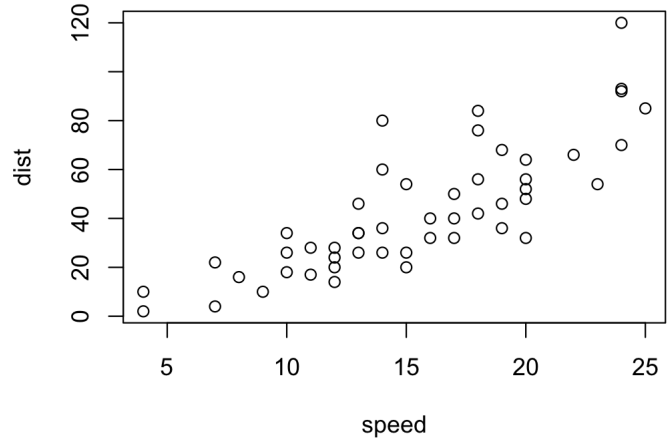
Base R and packages have built-in data frames with special names you can call on

For example, `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

```
d <- data.frame(x = 1:2, y = 3:4) # create a small data frame called "d"
d*10
```

```
##    x  y
## 1 10 30
## 2 20 40
```

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

```
mat <- as.matrix(d) # convert it to (i.e., create) a matrix call "mat"
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

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

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

```
# 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
```

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
```

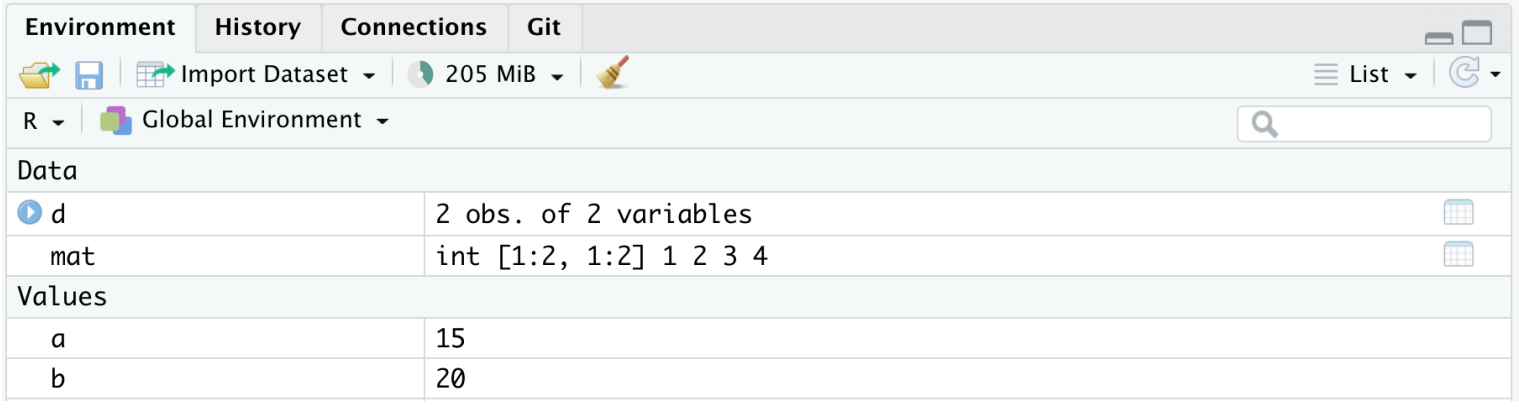
Uh-oh. What went wrong here?

Global environment

The error message provides the answer to our question:

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

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



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

We will come back to this

Reference material

(We don't have time for the rest of this today)

"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
```

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

Semi-reserved words (cont.)

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 distinguish between conflicting definitions! For example:

```
pi
```

```
## [1] 3.141593
```

```
pi <- 2
pi
```

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

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

Namespace conflicts

Try loading the `dplyr` package in RStudio

```
library(dplyr)
```

What warning gets reported?

```
The following objects are masked from ‘package:stats’:

    filter, lag

The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
```

The warning *masked from 'package:X'* is about a **namespace conflict**

Namespace conflicts

Whenever a namespace conflict arises, the most recently loaded package will gain preference

The `filter()` function now refers specifically to the `dplyr` variant

What if we want the `stats` variant?

1. Use `stats::filter()`
2. Assign `filter <- stats::filter`

Solving namespace conflicts

1. Use `package::function()`

Explicitly call a conflicted function from a package using the `package::function()` syntax

We can also use `::` to clarify the source of a function or dataset in our code

```
dplyr::starwars # print the starwars data frame from the dplyr package
scales::comma(c(1000, 1000000)) # use the comma function, which comes from the scales package
```

The `::` syntax also allows us to call functions without loading the package (as long as it is installed)

Solving namespace conflicts

2. Assign `function <- package::function`

A more persistent option is to assign a conflicted name to a particular package

```
filter <- stats::filter # note the lack of parentheses
filter <- dplyr::filter # change it back again
```

User-side namespace conflicts

Namespace conflicts don't just arise from loading packages

Users like you and me can (and probably will!) create them through assignment

Indexing

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
```

```
##  [1] -0.174443008  1.545317100  1.351017781  1.015205553  0.473798485
##  [6] -0.096533220 -0.322310310  0.153177244  0.239418137  0.156413413
## [11] -0.031493975  0.101216901 -0.223920645 -2.130356078  1.408712313
## [16] -0.663834759  1.148210744 -0.717833120  0.236425195  0.958974454
## [21]  0.895064434  0.477707538  0.352281509 -0.995005553  0.862133983
## [26]  0.650036197 -1.021848162 -1.415356256  0.119055675 -0.533926838
## [31]  0.002557935  0.680205965 -0.491916390  0.439463007  0.337348506
## [36]  0.004421690  0.967359169 -0.720593499  0.728681798 -1.320526053
## [41]  1.556530862 -0.723065150  0.741647289  0.248553919  1.358960055
## [46]  1.412690732  0.090484902 -0.596047126 -0.061523986  0.128014119
```

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 <- dplyr::starwars # assign for convenience
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 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 10
```

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