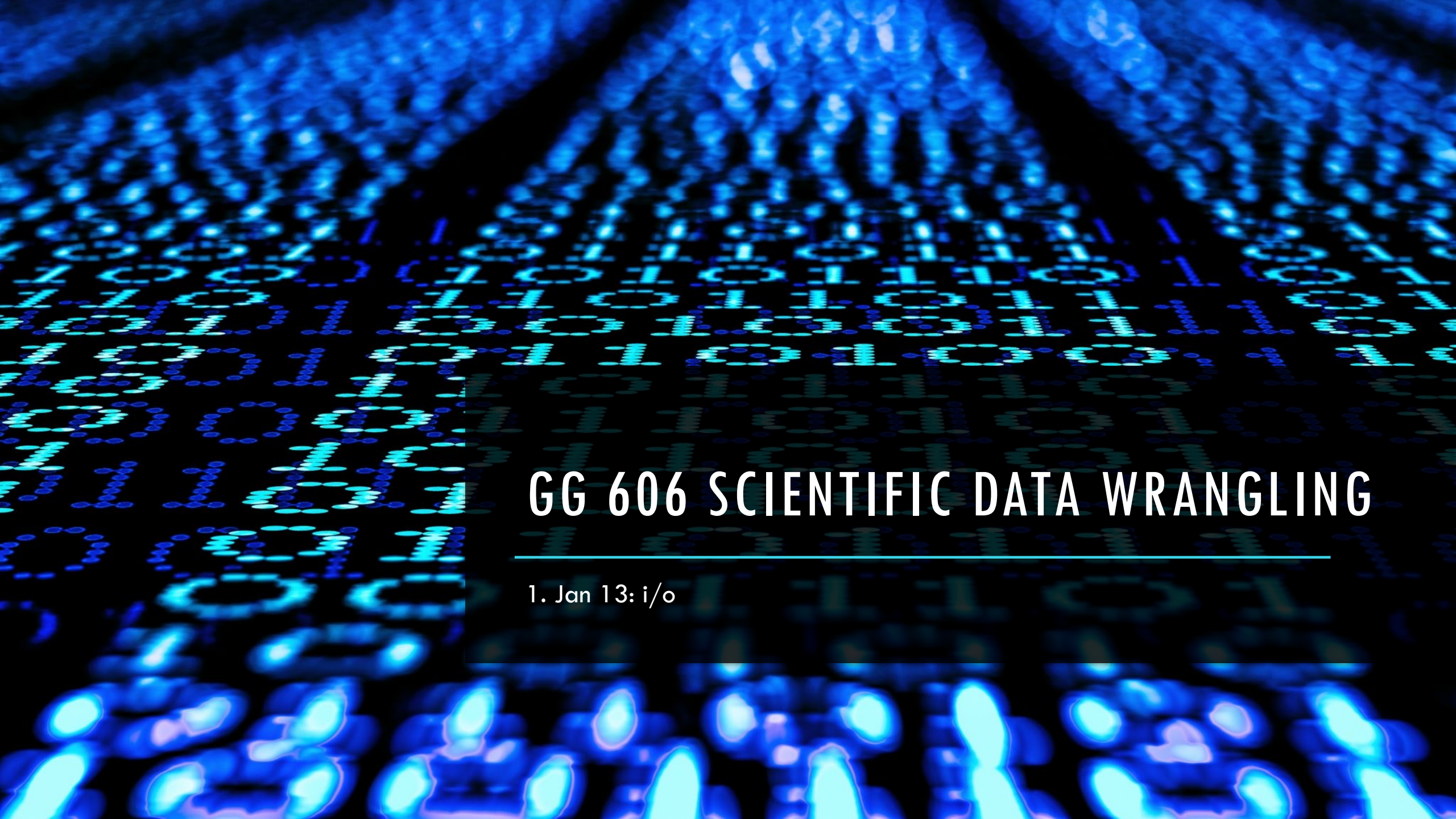


GG 606 SCIENTIFIC DATA WRANGLING

1. Jan 13: i/o

```
> 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 ca
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <c
 1  2013     1     1      517            515         2      830            819        11 UA
 2  2013     1     1      533            529         4      850            830        20 UA
 3  2013     1     1      542            540         2      923            850        33 AA
 4  2013     1     1      544            545        -1     1004           1022       -18 B6
 5  2013     1     1      554            600        -6      812            837       -25 DL
 6  2013     1     1      554            558        -4      740            728        12 UA
 7  2013     1     1      555            600        -5      913            854        19 B6
 8  2013     1     1      557            600        -3      709            723       -14 EV
 9  2013     1     1      557            600        -3      838            846        -8 B6
```

5.2.4 **Exercises**

1. Find all flights that
   1. Had an arrival delay of two or more hours
   2. Flew to Houston (`IAH` or `HOU`)
   3. Were operated by United, American, or Delta
   4. Departed in summer (July, August, and September)
   5. Arrived more than two hours late, but didn't leave late
   6. Were delayed by at least an hour, but made up over 30 minutes in flight
   7. Departed between midnight and 6am (inclusive)
2. Another useful dplyr filtering helper is `between()`. What does it do? Can you use it to simplify the code needed to answer the previous challenges?
3. How many flights have a missing `dep_time`? What other variables are missing? What might these rows represent?
4. Why is `NA ^ 0` not missing? Why is `NA | TRUE` not missing? Why is `FALSE & NA` not missing? Can you figure out the general rule? (`NA * 0` is a tricky counterexample!)

REVIEW

```r
#Find all flights that had an arrival delay of two or more hours
filter(flights, arr_delay > 120)

#Flew to Houston (IAH or HOU)
filter(flights, dest %in% c('IAH','HOU'))

#Were operated by United, American, or Delta
filter(flights, carrier %in% c('UA','AA', 'DL')) #%>% nrow()

#Departed in summer (July, August, and September)
filter(flights, month %in% c(7,8,9))

#Arrived more than two hours late, but didn't leave late
filter(flights, arr_delay > 120 & dep_delay <=0)

#Were delayed by at least an hour, but made up over 30 minutes
in flight
filter(flights, dep_delay >=60 & arr_delay <=30)

#Departed between midnight and 6am (inclusive)
filter(flights, dep_time <=600 | dep_time == 2400)


#Another useful dplyr filtering helper is between(). What does
it do? Can you use it to simplify the code needed to answer the
previous challenges?
filter(flights, month %in% c(7,8,9)) %>% nrow() ==
filter(flights, between(month, 7,9)) %>% nrow()


#How many flights have a missing dep_time? What other variables
are missing? What might these rows represent?
sum(is.na(flights$dep_time))
flights[is.na(flights$dep_time),] #probably cancelled flights
filter(flights, is.na(dep_time))
```

5.2.4 Exercises

1. Find all flights that
    1. Had an arrival delay of two or more hours
    2. Flew to Houston (`IAH` or `HOU`)
    3. Were operated by United, American, or Delta
    4. Departed in summer (July, August, and September)
    5. Arrived more than two hours late, but didn't leave late
    6. Were delayed by at least an hour, but made up over 30 minutes in flight
    7. Departed between midnight and 6am (inclusive)
2. Another useful dplyr filtering helper is `between()`. What does it do? Can you use it to simplify the code needed to answer the previous challenges?
3. How many flights have a missing `dep_time`? What other variables are missing? What might these rows represent?
4. Why is `NA ^ 0` not missing? Why is `NA | TRUE` not missing? Why is `FALSE & NA` not missing? Can you figure out the general rule? (`NA * 0` is a tricky counterexample!)

```r
#Why is NA ^ 0 not missing? Why is NA | TRUE not missing? Why is FALSE & NA
not missing? Can you figure out the general rule? (NA * 0 is a tricky
counterexample!)

anything ^0 is 1, we're evaluating if expressions are TRUE so NA|TRUE is
TRUE, while FALSE | NA is NA because we don't know it could be TRUE
```

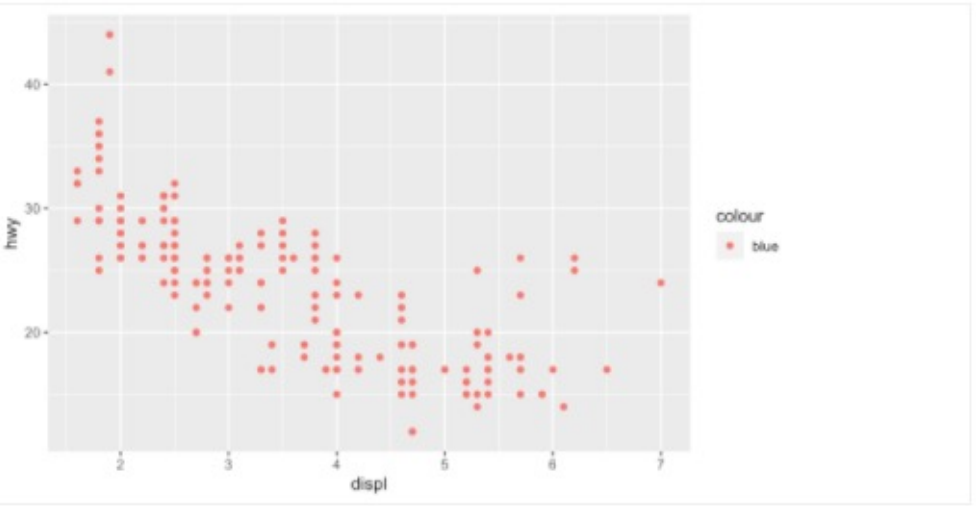
REVIEW

3.3.1 Exercises

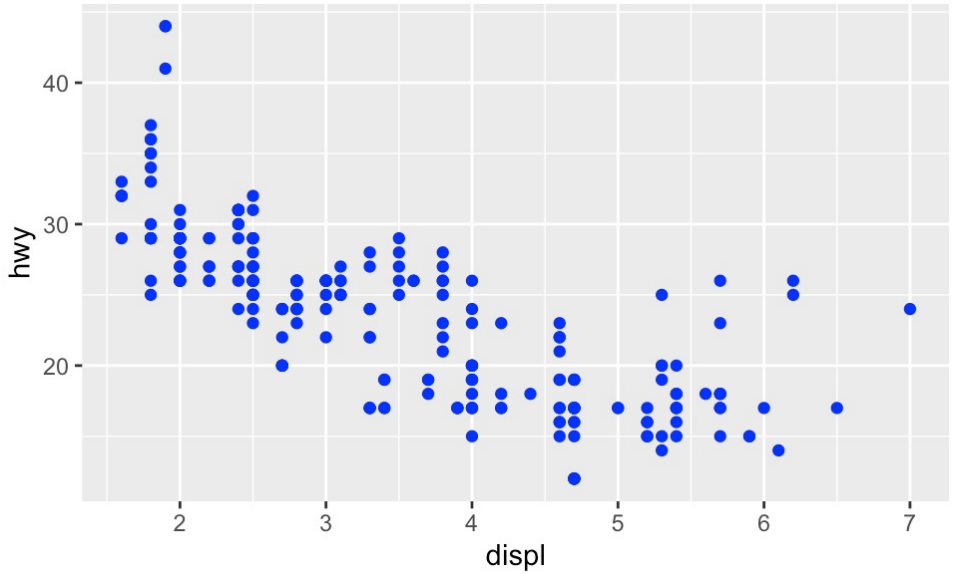
1. What's gone wrong with this code? Why are the points not blue?

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = "blue"))
```

Copy




```
#OLD
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = "blue"))
#NEW
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy), color = "blue")
```



REVIEW

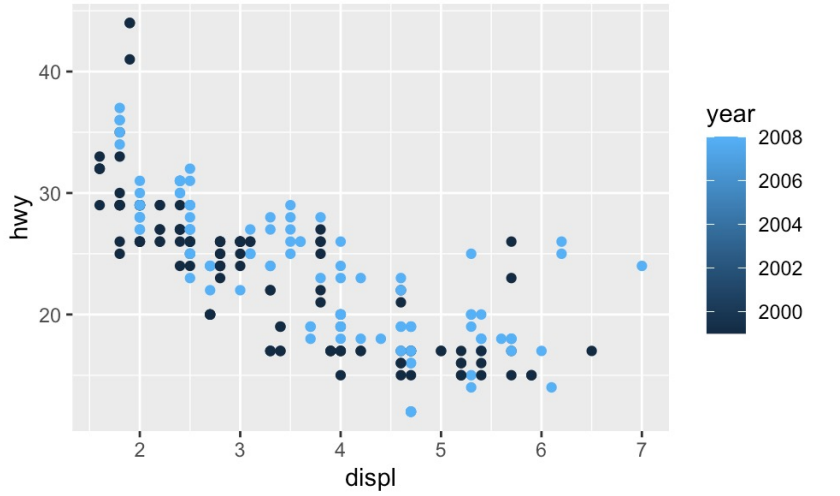
2. Which variables in `mpg` are categorical? Which variables are continuous? (Hint: type `?mpg` to read the documentation for the dataset). How can you see this information when you run `mpg`?

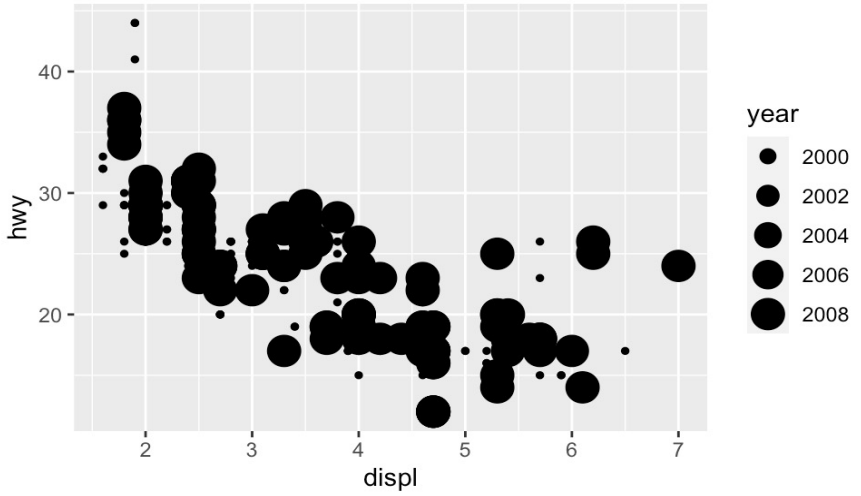
```
> mpg
# A tibble: 234 × 11
   manufacturer model      displ  year   cyl trans      drv     cty   hwy fl    class
   <chr>        <chr>      <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
 1 audi         a4           1.8  1999     4 auto(l5)   f        18    29 p     compact
 2 audi         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
```

REVIEW

3. Map a continuous variable to `color`, `size`, and `shape`. How do these aesthetics behave differently for categorical vs. continuous variables?

```
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, color=year))
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, size=year))
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, shape=year))
```

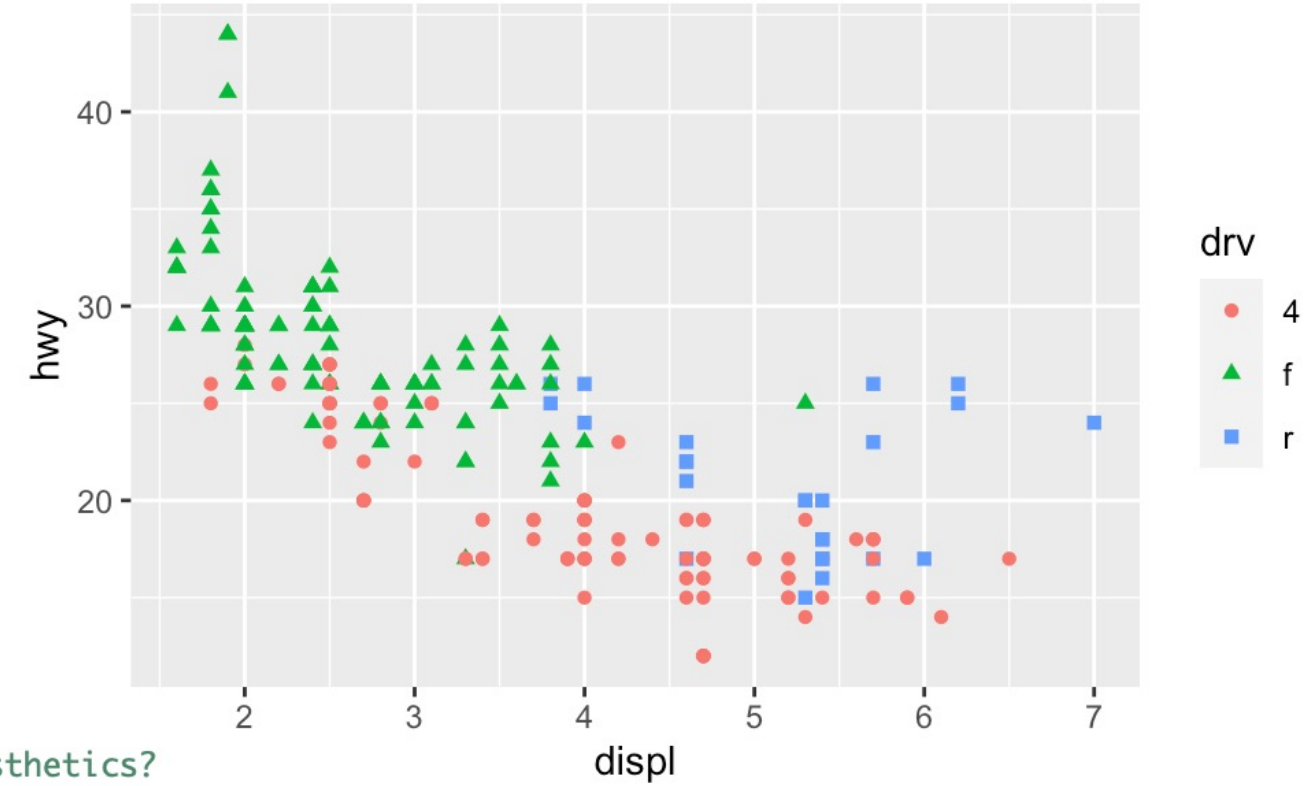





```
> ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, shape=year))
Error: A continuous variable can not be mapped to shape
Run `rlang::last_error()` to see where the error occurred.
```

REVIEW

4. What happens if you map the same variable to multiple aesthetics?




```
#4. What happens if you map the same variable to multiple aesthetics?
ggplot(data = mpg) + geom_point(aes(x = displ, y = hwy, color=drv, shape=drv))
```

REVIEW

5. What does the `stroke` aesthetic do? What shapes does it work with? (Hint: use `?geom_point`)

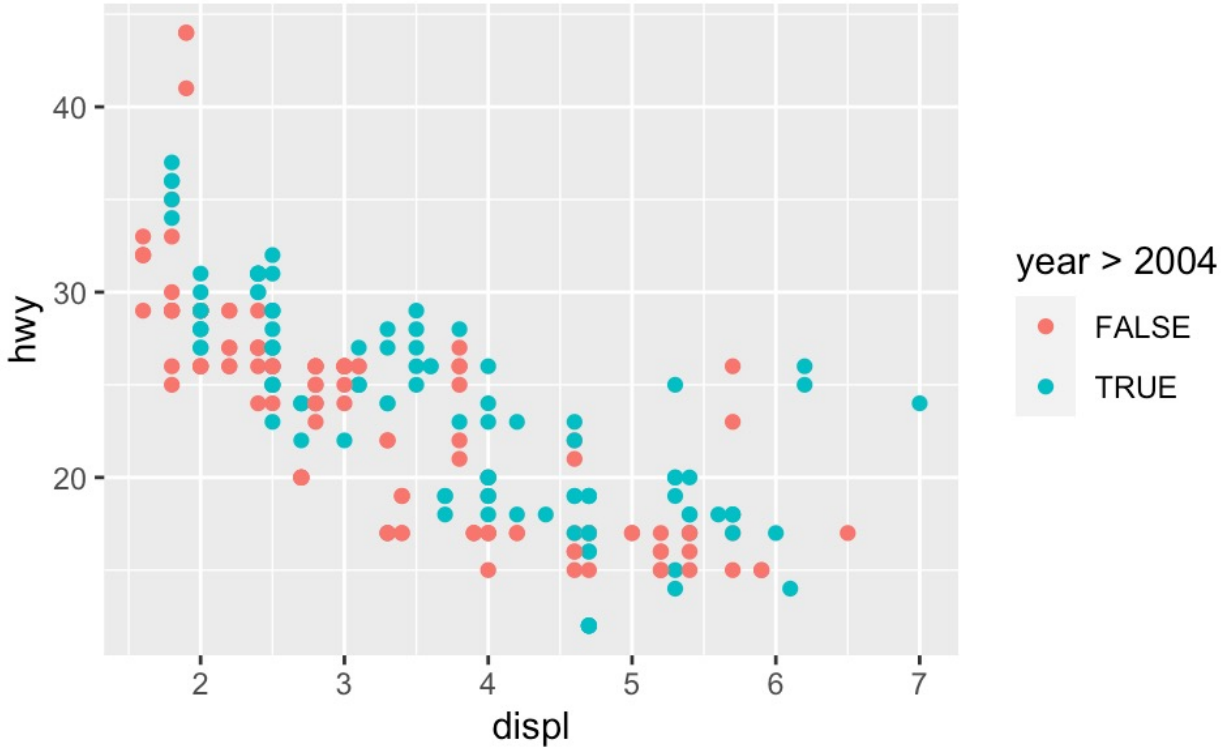
https://ggplot2.tidyverse.org/reference/geom_point.html

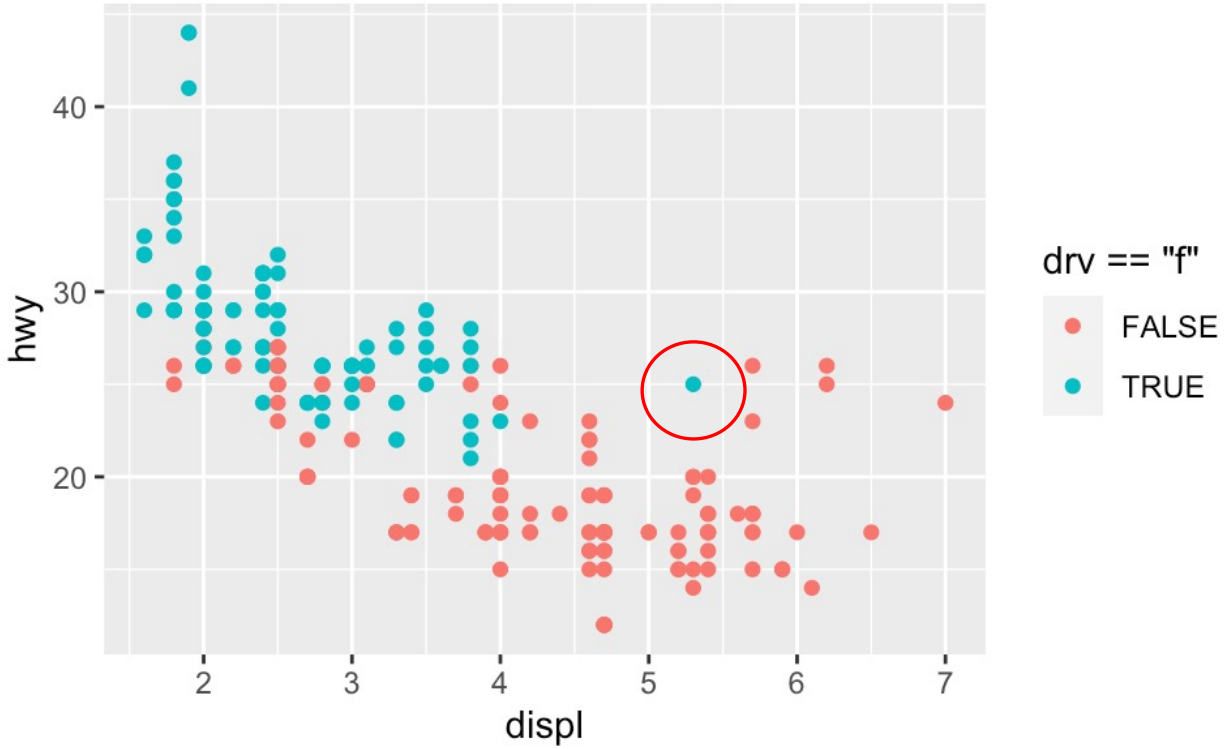
REVIEW

How could we find out what that outlier is?

6. What happens if you map an aesthetic to something other than a variable name, like `aes(colour = displ < 5)` ? Note, you'll also need to specify x and y.

it will evaluate the expression and plot results as the aesthetic mapping





REVIEW

How could we find out what that outlier is?

```
> filter(mpg, drv=='f' & displ > 5)
# A tibble: 1 × 11
  manufacturer model      displ  year   cyl trans    drv     cty   hwy fl    class
  <chr>        <chr>      <dbl> <int> <int> <chr>    <chr> <int> <int> <chr> <chr>
1 pontiac      grand prix   5.3  2008     8 auto(s4) f        16    25 p     midsize
>
```

LEARNING OBJECTIVES

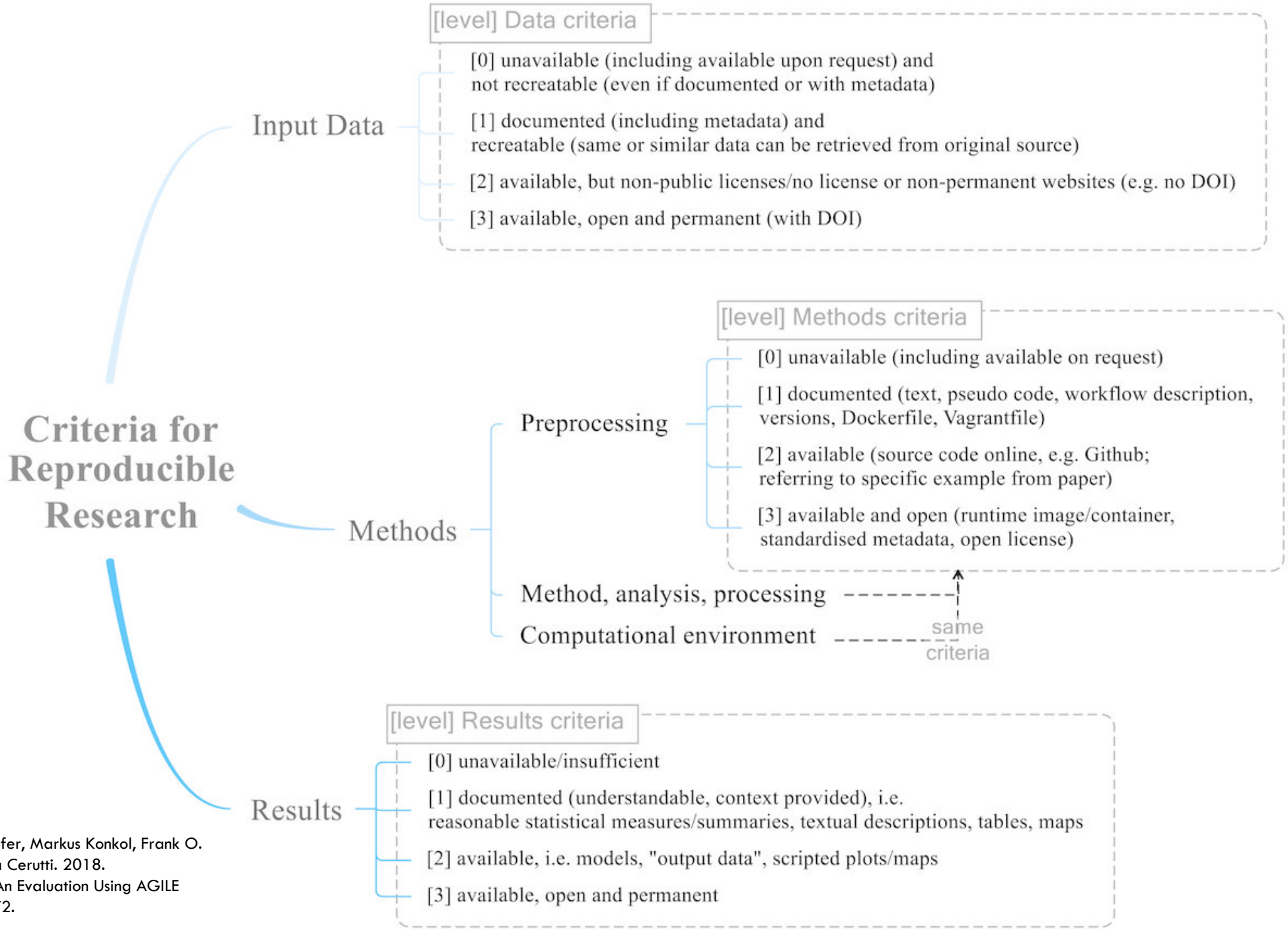
- Read and write common forms of data from R/R-Studio
- Understand text parsing and its relation to data importing in R
- List some of the commonly used data formats in environmental data analytics

TOPICS TO COVER THIS WEEK

- readr package – reading in data tidyverse way
- tidy data – concepts and techniques
- non-rectangular or alternate data types
  - rgdal – spatial
  - json
  - image/raster data – raster, rgeos, stacs
  - online data – curl, apis, etc.

REPRODUCIBLE SCIENTIFIC WORKFLOWS

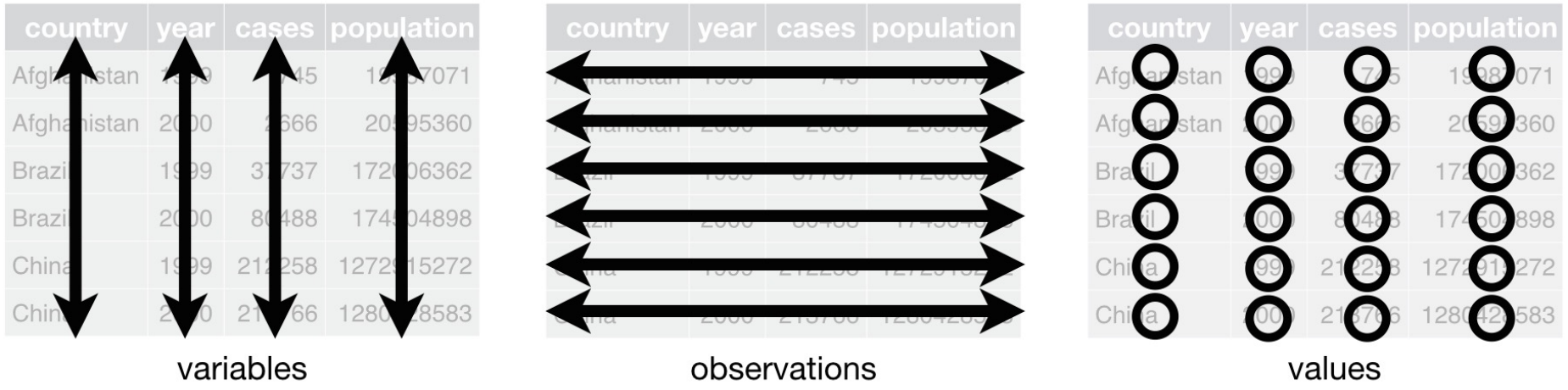
- lack of reproducibility in scientific studies is problem – especially in environmental science
- consistent and reproducible reading and writing of data used for scientific work is critical



Nüst, Daniel, Carlos Granell, Barbara Hofer, Markus Konkol, Frank O. Ostermann, Rusne Sileryte, and Valentina Cerutti. 2018. "Reproducible Research and GIScience: An Evaluation Using AGILE Conference Papers." *PeerJ* 6 (July): e5072. https://doi.org/10.7717/peerj.5072.

THE THREE RULES OF TIDY DATA

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.



MESSY DATA IS COMMON

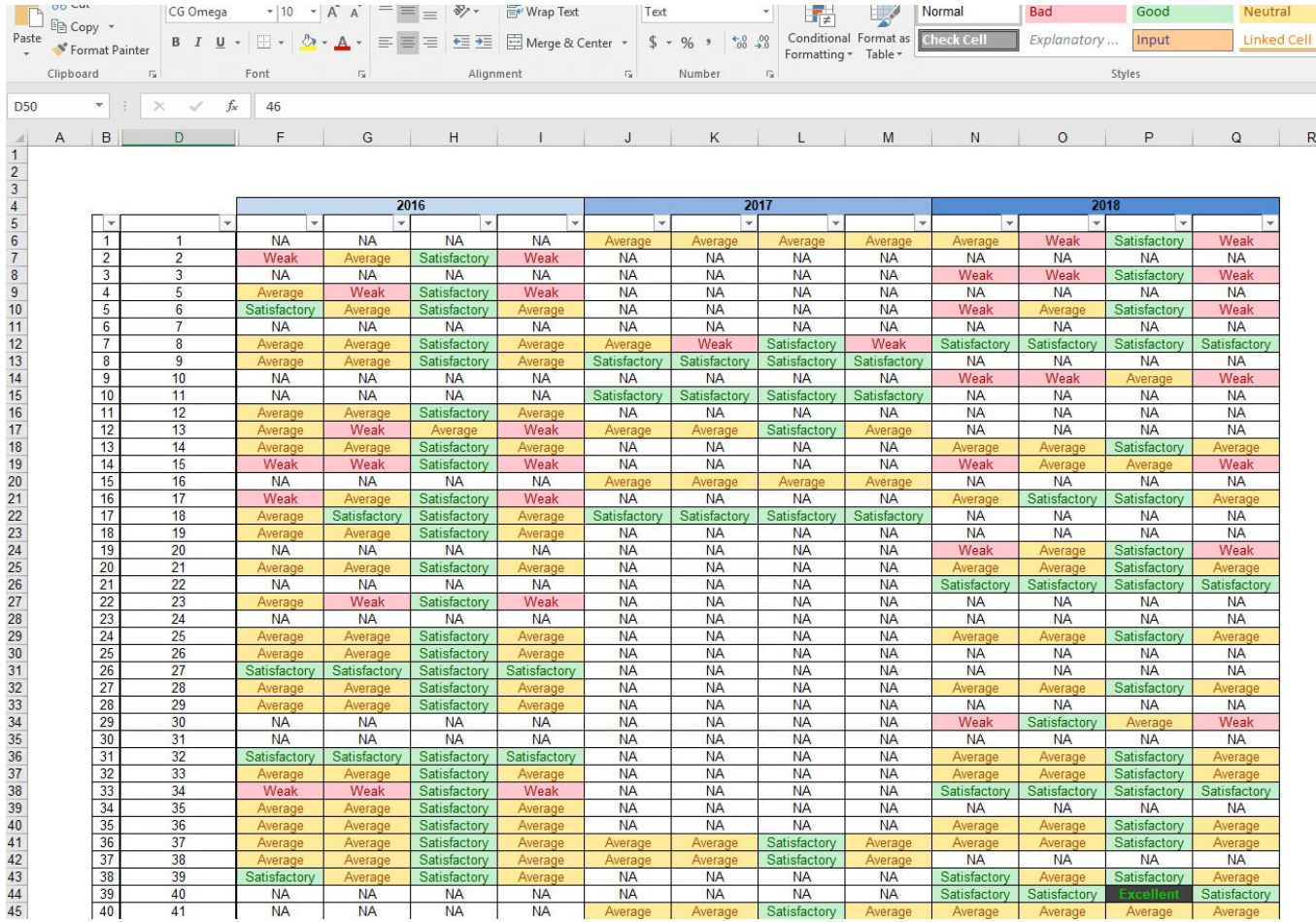
Wide data format

Time	A	B	C
0	1.1	4.2	5.6
1	1.0	4.5	5.8

Tidy data format

Time	Sample	Value	id
0	A	1.1	1
1	A	1.0	1
0	B	4.2	1
1	B	4.5	1
0	C	5.6	1
1	C	5.8	1

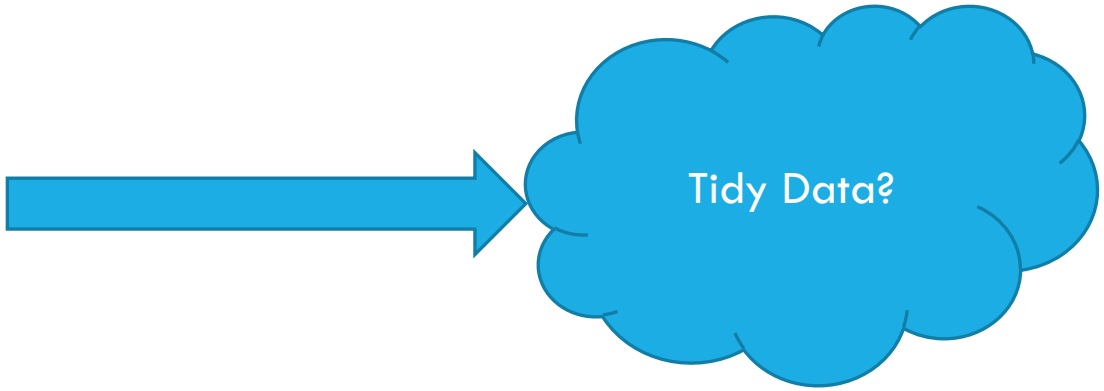
Joachim Goedhart



		2016				2017				2018			
1	1	NA	NA	NA	NA	Average	Average	Average	Average	Average	Weak	Satisfactory	Weak
2	2	Weak	Average	Satisfactory	Weak	NA	NA	NA	NA	NA	NA	NA	NA
3	3	NA	NA	NA	NA	NA	NA	NA	NA	Weak	Weak	Satisfactory	Weak
4	5	Average	Weak	Satisfactory	Weak	NA	NA	NA	NA	NA	NA	NA	NA
5	6	Satisfactory	Average	Satisfactory	Average	NA	NA	NA	NA	Weak	Average	Satisfactory	Weak
6	7	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
7	8	Average	Average	Satisfactory	Average	Average	Weak	Satisfactory	Weak	Satisfactory	Satisfactory	Satisfactory	Satisfactory
8	9	Average	Average	Satisfactory	Average	Satisfactory	Satisfactory	Satisfactory	Satisfactory	NA	NA	NA	NA
9	10	NA	NA	NA	NA	NA	NA	NA	NA	Weak	Weak	Average	Weak
10	11	NA	NA	NA	NA	Satisfactory	Satisfactory	Satisfactory	Satisfactory	NA	NA	NA	NA
11	12	Average	Average	Satisfactory	Average	NA	NA	NA	NA	NA	NA	NA	NA
12	13	Average	Weak	Average	Weak	Average	Average	Satisfactory	Average	NA	NA	NA	NA
13	14	Average	Average	Satisfactory	Average	NA	NA	NA	NA	Average	Average	Satisfactory	Average
14	15	Weak	Weak	Satisfactory	Weak	NA	NA	NA	NA	Weak	Average	Average	Weak
15	16	NA	NA	NA	NA	Average	Average	Average	Average	NA	NA	NA	NA
16	17	Weak	Average	Satisfactory	Weak	NA	NA	NA	NA	Average	Satisfactory	Satisfactory	Average
17	18	Average	Satisfactory	Satisfactory	Average	Satisfactory	Satisfactory	Satisfactory	Satisfactory	NA	NA	NA	NA
18	19	Average	Average	Satisfactory	Average	NA	NA	NA	NA	NA	NA	NA	NA
19	20	NA	NA	NA	NA	NA	NA	NA	NA	Weak	Average	Satisfactory	Weak
20	21	Average	Average	Satisfactory	Average	NA	NA	NA	NA	Average	Average	Satisfactory	Average
21	22	NA	NA	NA	NA	NA	NA	NA	NA	Satisfactory	Satisfactory	Satisfactory	Satisfactory
22	23	Average	Weak	Satisfactory	Weak	NA	NA	NA	NA	NA	NA	NA	NA
23	24	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
24	25	Average	Average	Satisfactory	Average	NA	NA	NA	NA	Average	Average	Satisfactory	Average
25	26	Average	Average	Satisfactory	Average	NA	NA	NA	NA	NA	NA	NA	NA
26	27	Satisfactory	Satisfactory	Satisfactory	Satisfactory	NA	NA	NA	NA	NA	NA	NA	NA
27	28	Average	Average	Satisfactory	Average	NA	NA	NA	NA	Average	Average	Satisfactory	Average
28	29	Average	Average	Satisfactory	Average	NA	NA	NA	NA	NA	NA	NA	NA
29	30	NA	NA	NA	NA	NA	NA	NA	NA	Weak	Satisfactory	Average	Weak
30	31	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
31	32	Satisfactory	Satisfactory	Satisfactory	Satisfactory	NA	NA	NA	NA	Average	Average	Satisfactory	Average
32	33	Average	Average	Satisfactory	Average	NA	NA	NA	NA	Average	Average	Satisfactory	Average
33	34	Weak	Weak	Satisfactory	Weak	NA	NA	NA	NA	Satisfactory	Satisfactory	Satisfactory	Satisfactory
34	35	Average	Average	Satisfactory	Average	NA	NA	NA	NA	NA	NA	NA	NA
35	36	Average	Average	Satisfactory	Average	NA	NA	NA	NA	Average	Average	Satisfactory	Average
36	37	Average	Average	Satisfactory	Average	Average	Average	Satisfactory	Average	Average	Average	Satisfactory	Average
37	38	Average	Average	Satisfactory	Average	Average	Average	Satisfactory	Average	NA	NA	NA	NA
38	39	Satisfactory	Average	Satisfactory	Average	NA	NA	NA	NA	Satisfactory	Average	Satisfactory	Average
39	40	NA	NA	NA	NA	NA	NA	NA	NA	Satisfactory	Satisfactory	Excellent	Satisfactory
40	41	NA	NA	NA	NA	Average	Average	Satisfactory	Average	Average	Average	Average	Average

MESSY DATA IS COMMON

	Population Estimates			
District	**MOH area**	**2009**	**2010**	**2011**
Colombo	Dehiwala	233664	236809	240018
	Piliyandala	168958	171232	173553
	Homagama	204699	207454	210266
	Kaduwela	233612	236757	239966
	Kolonnawa	178675	181080	183534
	Kotte	69302	70234	71186
	Maharagama	157089	159203	161361
	MC-Colombo	715249	724877	734702
	Moratuwa	197357	200013	202724
	Nugegoda	103230	104619	106037
	Padukka	60589	61407	62234
	Boralesgamuwa	61944	62777	63628
	Hanwella	104218	105621	107053
Gampaha	Attanagalla	180397	183907	187521
	Biyagama	188435	192102	195877
	Divulapitiya	149474	152382	155377
	Gampaha	199015	202888	206875
	Ja-Ela	148509	151399	154374



tibble or data frame

TIBBLES VS DATA FRAMES

- **Input type remains unchanged** - data.frame is notorious for treating strings as factors; this will not happen with tibbles
- **Variable names remain unchanged** - In base R, creating data.frames will remove spaces from names, converting them to periods or add "x" before numeric column names. Creating tibbles will not change variable (column) names.
- **There are no row.names() for a tibble** - Tidy data requires that variables be stored in a consistent way, removing the need for row names.
- **Tibbles print first ten rows and columns that fit on one screen** - Printing a tibble to screen will never print the entire huge data frame out. By default, it just shows what fits to your screen.

TABULAR DATA INTO R

- Use `read_csv` or `read.csv` functions to read in CSV data
- It is possible to read Exel or Google Sheets data directly, however this is more prone to error, when possible convert then read in
  - Spreadsheet → CSV → R

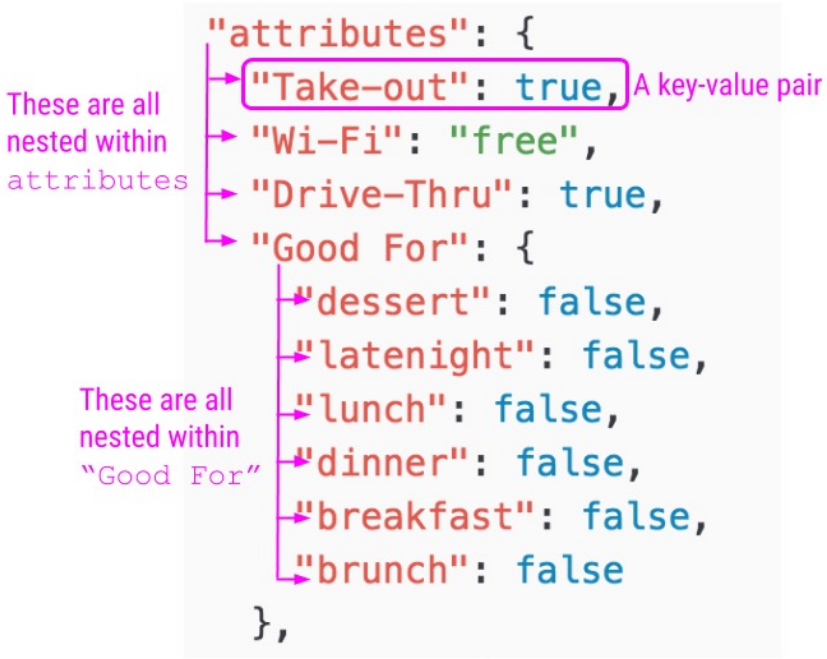
this is a guess that needs to be checked

read.

```
heights <- read_csv("data/heights.csv")
#>
#> ── Column specification ────────────────────────────────────────
#> cols(
#>   earn = col_double(),
#>   height = col_double(),
#>   sex = col_character(),
#>   ed = col_double(),
#>   age = col_double(),
#>   race = col_character()
#> )
```

JSON – JAVASCRIPT OBJECT NOTATION

- used for movement of text data on the internet, between web browser and server for example
- JSON data is composed of simple key-value pairs
- Geographic version of JSON called GeoJSON – widely used in web mapping frameworks
  - including mapping using the Leaflet library from R



https://blog.exploratory.io/working-with-json-data-in-very-simple-way-ad7ebcc0bb89

GEOJSON

Type	Examples
Point	
LineString	
Polygon	
	

```
{
    "type": "Point",
    "coordinates": [30, 10]
}
```

```
{
    "type": "LineString",
    "coordinates": [
        [30, 10], [10, 30], [40, 40]
    ]
}
```

```
{
    "type": "Polygon",
    "coordinates": [
        [[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]
    ]
}
```

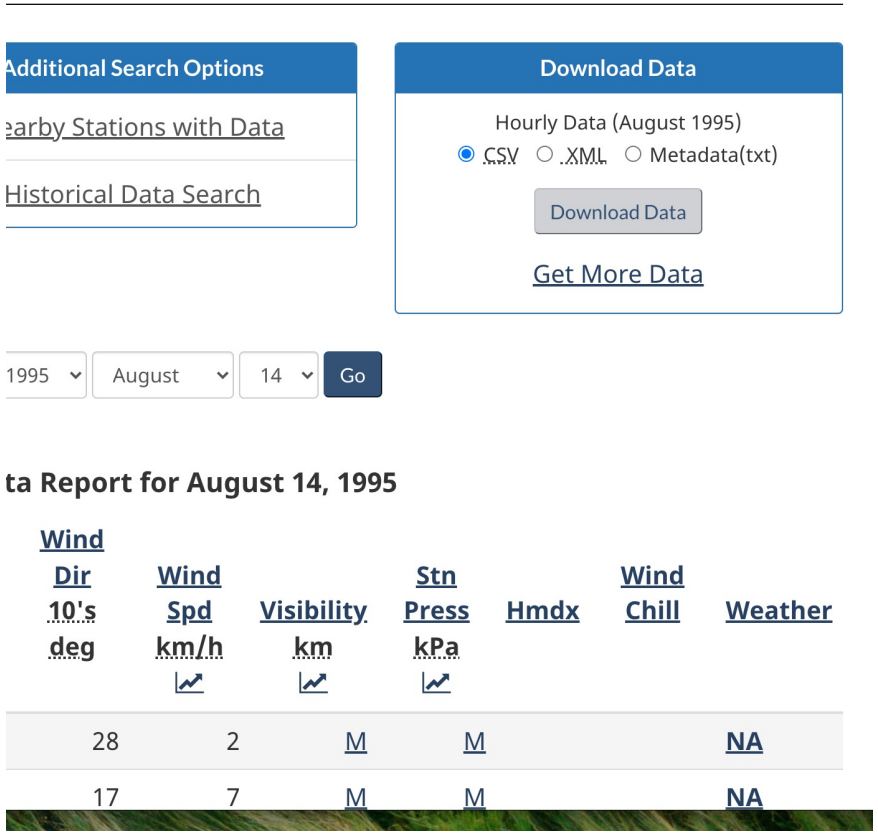
```
{
    "type": "Polygon",
    "coordinates": [
        [[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],
        [[20, 30], [35, 35], [30, 20], [20, 30]]
    ]
}
```


```
{
"type": "FeatureCollection",
"crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:EPSG::26943" } },

"features": [
{ "type": "Feature", "properties": { "id": 3, "Name": "Mills Hall", "MapKey": null },
"geometry": { "type": "Polygon", "coordinates": [ [ [ 6075433.155601346865296,
2110761.308253711555153 ], [ 6075460.17075748834759, 2110799.385876494459808 ],
[ 6075525.398678226396441, 2110753.996098623145372 ], [ 6075530.066493064165115,
2110760.627852647565305 ], [ 6075547.09583081305027, 2110748.24298348557204 ],
[ 6075560.516077329404652, 2110767.678209110628814 ], [ 6075532.0845170551911,
2110788.71110668964684 ], [ 6075559.82685037329793, 2110827.702769040130079 ],
[ 6075546.013382173143327, 2110837.609512744937092 ], [ 6075567.439776981249452,
2110868.925579437520355 ], [ 6075562.105833441950381, 2110872.67049785843119 ],
[ 6075590.059257703833282, 2110911.674293594900519 ], [ 6075608.784731969237328,
2110900.8104556822218 ], [ 6075579.425908595323563, 2110861.488368191290647 ],
[ 6075624.832116444595158, 2110825.141613696236163 ], [ 6075603.823744205757976,
2110794.194464405998588 ], [ 6075618.229612559080124, 2110783.569777271244675 ],
[ 6075591.277481381781399, 2110742.832968316972256 ], [ 6075569.571757420897484,
2110757.954965581186116 ], [ 6075557.239970221184194, 2110742.712741161696613 ],
[ 6075568.258573904633522, 2110733.795875346288085 ], [ 6075561.176165237091482,
2110724.888911794871092 ], [ 6075601.109866349957883, 2110695.693394737783819 ],
[ 6075668.972125135362148, 2110797.201805088669062 ], [ 6075689.300683847628534,
2110782.855504938866943 ], [ 6075668.673246178776026, 2110752.368591027799994 ],
[ 6075676.007501310668886, 2110747.093861604575068 ], [ 6075645.560291484929621,
2110702.813309468328953 ], [ 6075653.379581064917147, 2110695.545510984957218 ],
[ 6075631.689454830251634, 2110669.406318536028266 ], [ 6075627.247816104441881,
2110677.18475692300126 ], [ 6075600.595512701198459, 2110635.554167079273611 ],
[ 6075574.988727924413979, 2110656.502569102216512 ], [ 6075579.885425406508148,
2110662.954826896544546 ], [ 6075539.57698761485517, 2110692.928919678088278 ],
[ 6075536.173024679534137, 2110685.689812638331205 ], [ 6075498.047465051524341,
2110712.971842519938946 ], [ 6075503.890731132589281, 2110720.009502995293587 ],
[ 6075467.082116083241999, 2110747.895543638151139 ], [ 6075461.849721318110824,
2110740.056419994682074 ], [ 6075433.155601346865296, 2110761.308253711555153 ] ] ] } },
{ "type": "Feature", "properties": { "id": 1, "Name": "Music Building", "MapKey": null },
"geometry": { "type": "Polygon", "coordinates": [ [ [ 6074644.011824694462121,
2111479.179281256627291 ], [ 6074697.852151760831475, 2111535.013694510329515 ],
```


http://djjr-courses.wikidot.com/

CSV FILES – SIMPLE

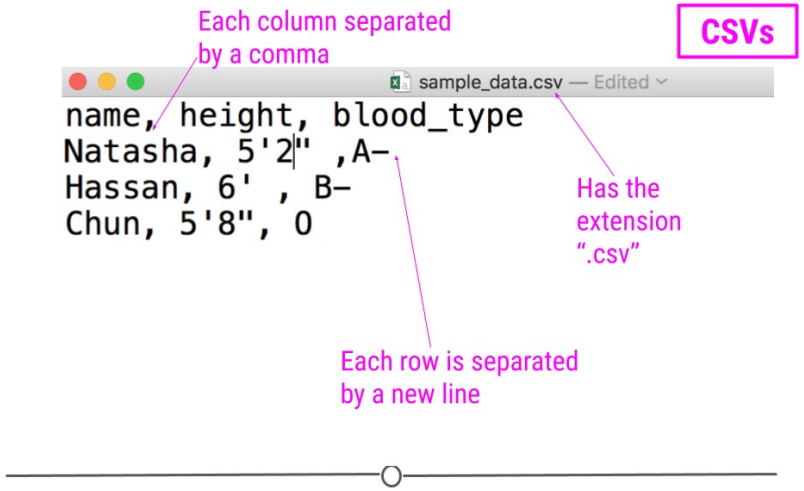
- comma separate value ("CSV") files are the most commonly used data interchange format – used mostly for moving data from one system or program to another
- plain-text files (i.e., data are stored as text, not proprietary binary formats)
  - universally readable
  - not the most efficient or fast
- e.g., historical weather data from Environment Canada: csv



https://climate.weather.gc.ca/

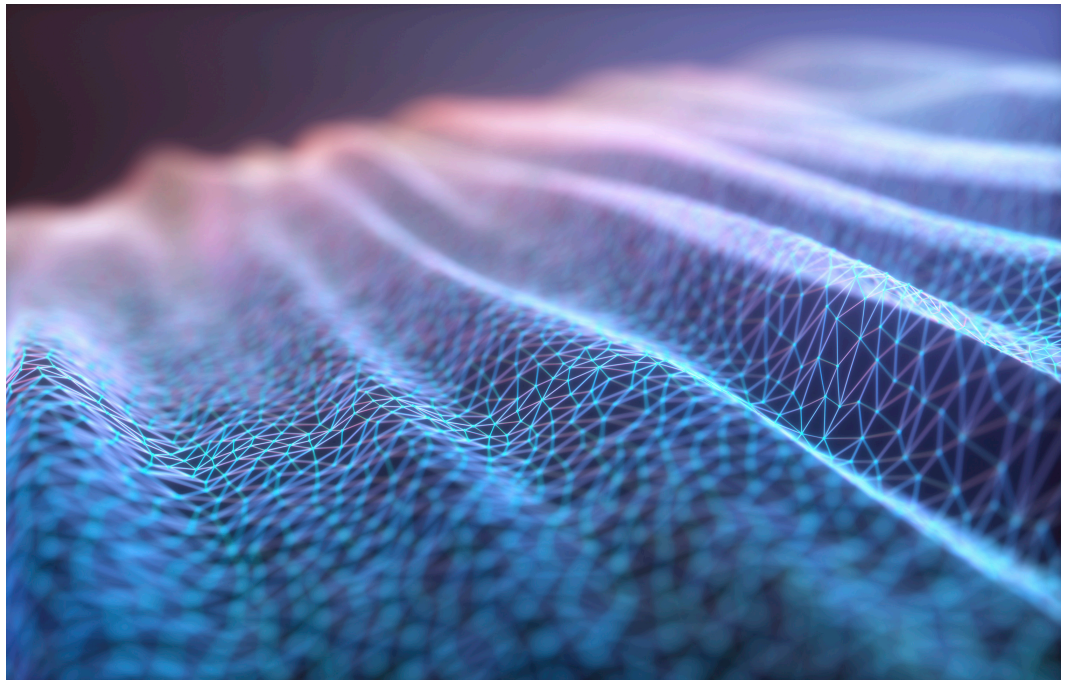
CSV FILES - ISSUES

- can cause problems when used to store large blocks of text data
- cannot handle hierarchical / nested data (true of all rectangular formats)
- no data types



CSV DIRECT FROM URL

- Very useful workflow is to skip downloading data completely and read from URL directly..
  - especially useful if data are changing and updating often and you want the latest data every time you run your analysis



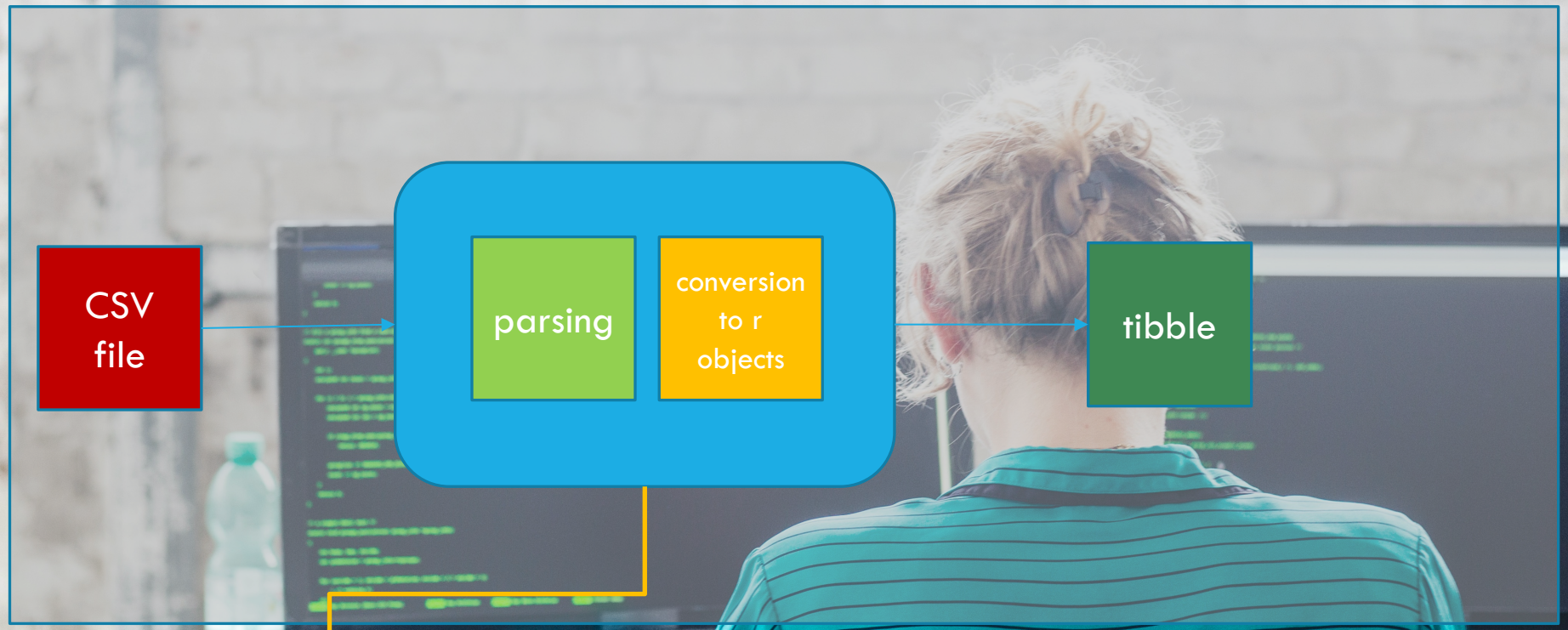
```
df <- read_csv("https://raw.githubusercontent.com/fivethirtyeight/data/master/steak-survey/steak-risk-survey.csv")
```

function in readr

url to CSV file (i.e., not a webpage CSV is listed on)

https://readr.tidyverse.org/reference/read_delim.html

WHAT HAPPENS WHEN YOU READ IN A FILE



reading in a character data type and coverting to another more specialized data type

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
#>  logi [1:3] TRUE FALSE NA
str(parse_integer(c("1", "2", "3")))
#>  int [1:3] 1 2 3
str(parse_date(c("2010-01-01", "1979-10-14")))
```

Copy

PARSING: NUMBERS

- Numbers can be formatted in ways that make it easier for humans but more difficu for computers
  - $10,000 → 10000
  - 19% → 19 or 0.19
  - 1,000,000 → 1000000

```
# Used in America
parse_number("$123,456,789")
#> [1] 123456789

# Used in many parts of Europe
parse_number("123.456.789", locale = locale(grouping_mark = "."))
#> [1] 123456789

# Used in Switzerland
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
#> [1] 123456789
```

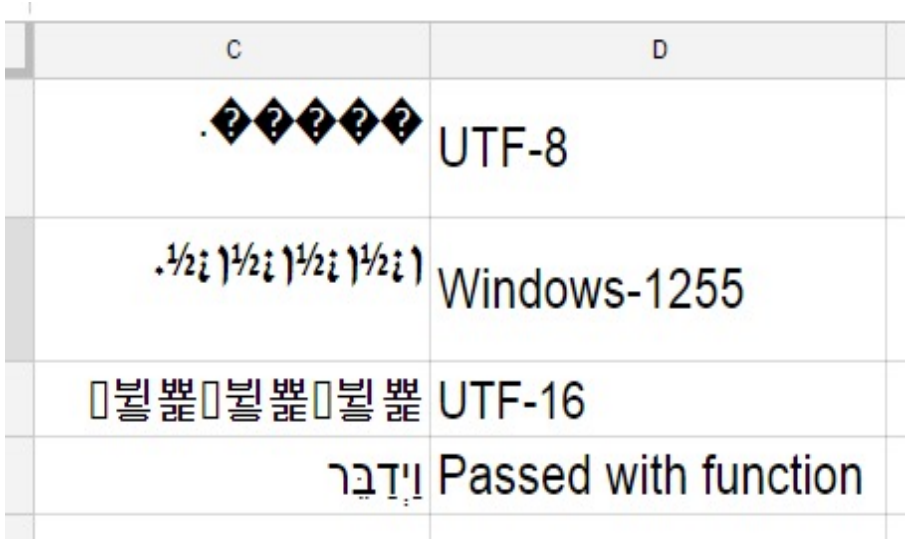
North America

Europe

```
parse_double("1.23")
#> [1] 1.23
parse_double("1,23", locale = locale(decimal_mark = ","))
#> [1] 1.23
```

PARSING: STRINGS

- Text data (called strings) are surprisingly the source of many issues, mainly due to character encoding issues
- Many older sensors and software programs encode text using ASCII – originally a 7-bit text encoding standard
  - 2^7 = 128 possible characters
  - good for English letters and punctuation, not good for any other languages, emojis, etc
  - extended versions of ASCII are 2^8 = 256 characters
- Modern character encoding is UTF-8 – can use up to 4 bytes (1 byte = 8 bits) per character
  - over a million possible characters
- For sensor data may need to find character encoding used and specify in `parse_character` function



C	D
.□□□□□	UTF-8
.½¿)½¿)½¿)½¿)	Windows-1255
□뵿뾽□뵿뾽□뵿뾽	UTF-16
וַיְדַבֵּר	Passed with function

PARSING: DATES

- Dates are notoriously difficult and often the source of problems –
- Need to specify codes to properly represent your da data – get familiar with the codes
- Other option is to split data data up into three integer columns
  - can make working with data for analysis much easier

`parse_date()` expects a four digit year, a `-` or `/`, the month, a `-` or `/`, then the day:

```
parse_date("2010-10-01")
#> [1] "2010-10-01"
```

Copy

Year

`%Y` (4 digits).

`%y` (2 digits); 00-69 -> 2000-2069, 70-99 -> 1970-1999.

Month

`%m` (2 digits).

`%b` (abbreviated name, like “Jan”).

`%B` (full name, “January”).

Day

`%d` (2 digits).

`%e` (optional leading space).

PARSING: DATES – THE OLD WAY

- base R classes for date and time handling in a variety of different ways
  - none of them great

https://www.r-bloggers.com/2013/08/date-formats-in-r/

readr parsing

- reads the first 1000 rows and uses some heuristics to guess the type of each column
- the data may be organized in a way such that the first 1000 rows do not give a good sample for determining the data type
- can specify the data type manually using `col_types` inside the `read_csv` function

```
df3 <- read_csv(
  readr_example("challenge.csv"),
  col_types = list(
    x = col_double(),
    y = col_date(format = "")
  )
)
```

```
guess_parser("2010-10-01")
#> [1] "date"
guess_parser("15:01")
#> [1] "time"
guess_parser(c("TRUE", "FALSE"))
#> [1] "logical"
guess_parser(c("1", "5", "9"))
#> [1] "double"
guess_parser(c("12,352,561"))
#> [1] "number"

str(parse_guess("2010-10-10"))
#>  Date[1:1], format: "2010-10-10"
```

readr parsing

- example using iris data

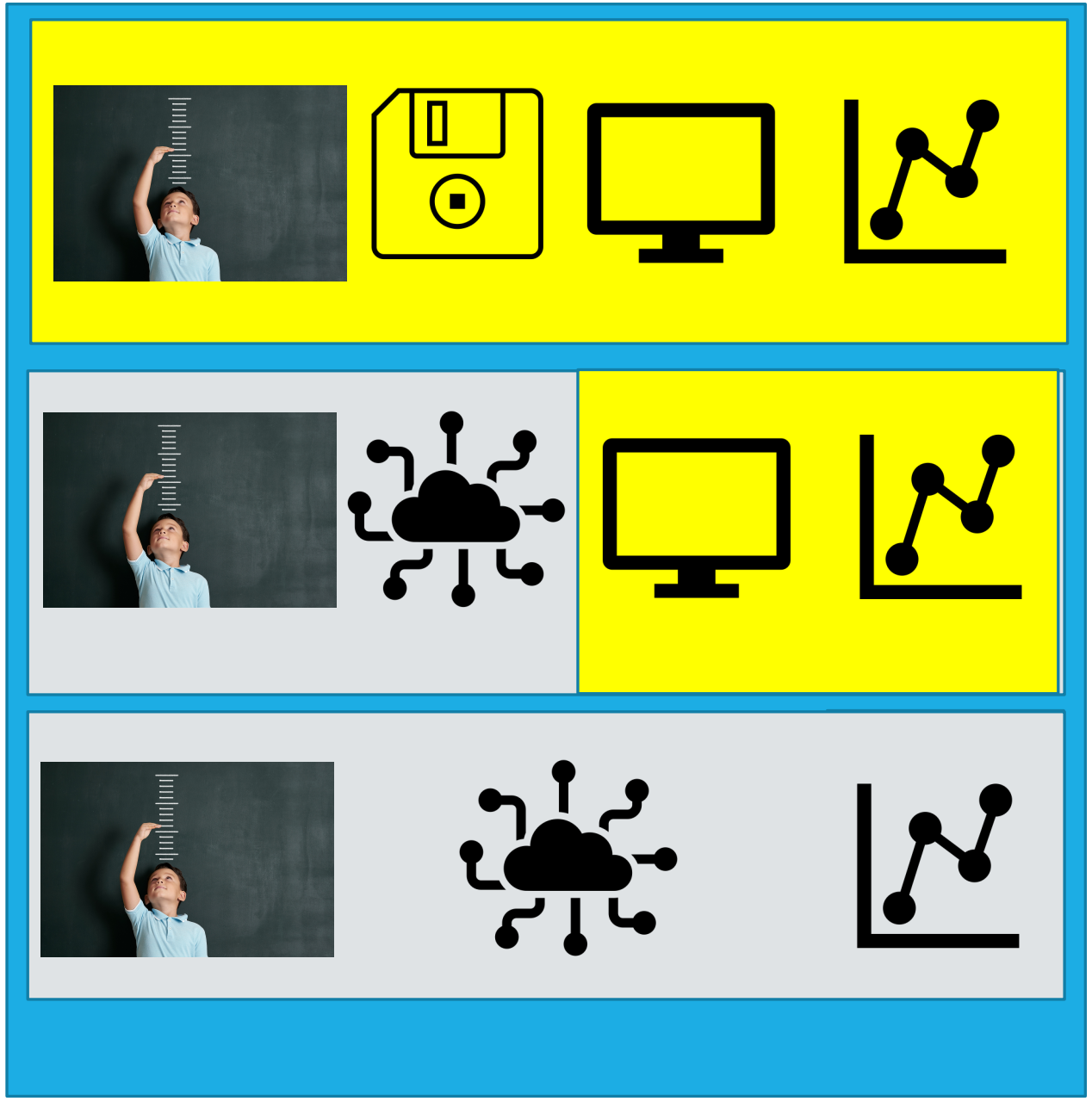
```
url <- "http://vincentarelbundock.github.io/Rdatasets/csv/datasets/iris.csv"
ird <- read_csv(url,
                col_types = list(
                  Sepal.Length = col_double(),
                  Sepal.Width = col_double(),
                  Petal.Length = col_double(),
                  Petal.Width = col_double(),
                  Species = col_factor(c("setosa", "versicolor", "virginica")) ))
```


tOrange.biz

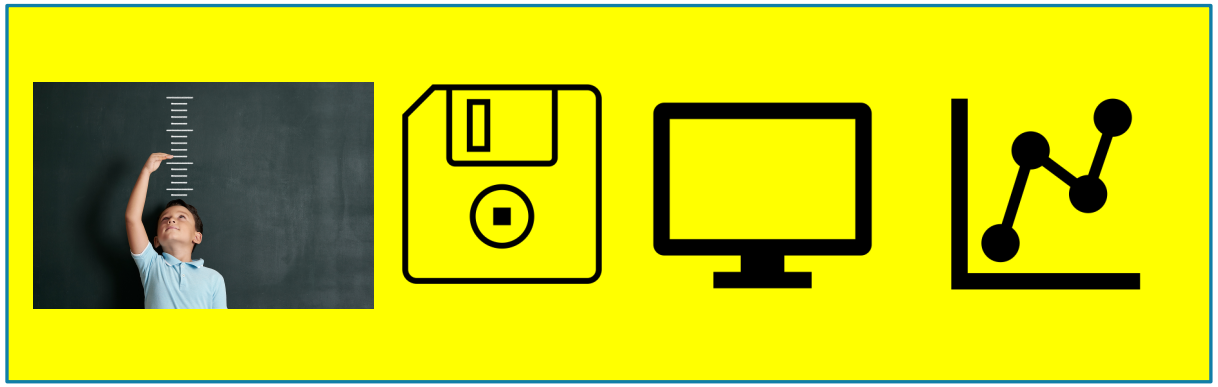
SPATIAL DATA

- many more formats for geospatial data
- Vector
  - shapefiles, geojson, autocad, gpx
- Raster
  - jpg, geotiff, mrsid, jpg2000, etc
- Other
  - las, kml/z,

Geospatial Data Workflows



READING AND WRITING GEOSPATIAL DATA IN R



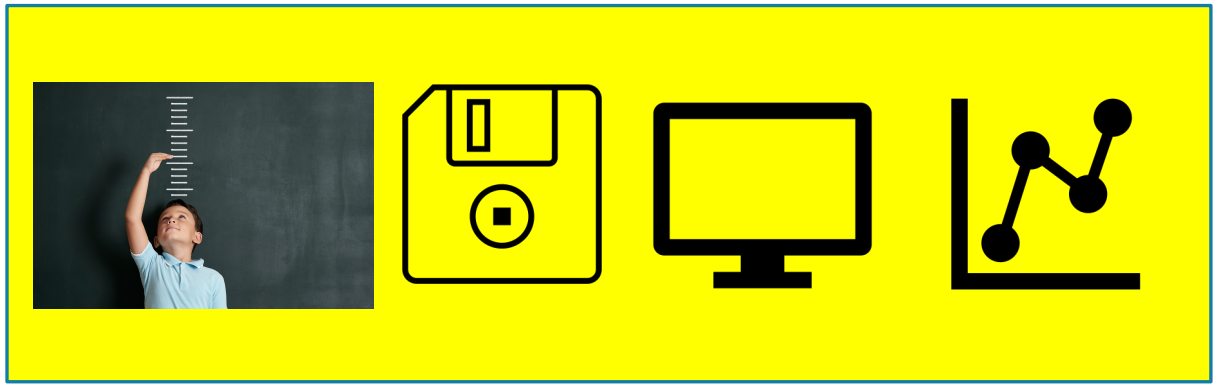
- Old Approach

```
#*********************************************************************
*******
library(rgdal)
#script for bycatch and STAL analysis
setwd("/Users/colinr23/Dropbox/personal/spatialResear/EC/data/projDat")
options(stringsAsFactors = FALSE)

#first read in unioned shapefile of annual hotspots
hs <- readOGR("./final/Annual_full", "hs_all")
bycatch <- readOGR("./", "byCatch")
poly <- readOGR(".", "fishingGroundsArea")
proj4string(hs) <- proj4string(bycatch)
```

- New Approach

```
nc_dfr = st_read(system.file("shape/nc.shp", package="sf"))
#> Reading layer `nc' from data source
#>   `/usr/local/lib/R/site-library/sf/shape/nc.shp' using driver `ESRI
Shapefile'
#> Simple feature collection with 100 features and 14 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -84.3 ymin: 33.9 xmax: -75.5 ymax: 36.6
#> Geodetic CRS:  NAD27
nc_tbl = read_sf(system.file("shape/nc.shp", package="sf"))
class(nc_dfr)
#> [1] "sf"         "data.frame"
class(nc_tbl)
#> [1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

READING AND WRITING GEOSPATIAL DATA IN R



- Old Approach

```
# load the raster packages
library(raster)
# set working directory to data folder #setwd("pathToDirHere")
wd <- ("~/Git/data/")
setwd(wd)
DEM <- raster(paste0(wd, "NEON-DS-Field-Site-Spatial-Data/SJER/DigitalT
```

- New Approach

```
library(terra)
multi_raster_file = system.file("raster/landsat.tif", packag
"spDataLarge")
multi_rast = rast(multi_raster_file)
multi_rast
#> class       : SpatRaster
#> dimensions  : 1428, 1128, 4  (nrow, ncol, nlyr)
#> resolution  : 30, 30  (x, y)
#> extent      : 301905, 335745, 4111245, 4154085  (xmin, xm
#> coord. ref. : WGS 84 / UTM zone 12N (EPSG:32612)
#> source      : landsat.tif
#> names       : landsat_1, landsat_2, landsat_3, landsat_4
#> min values  :      7550,      6404,      5678,      5252
#> max values  :     19071,     22051,     25780,     31961
```

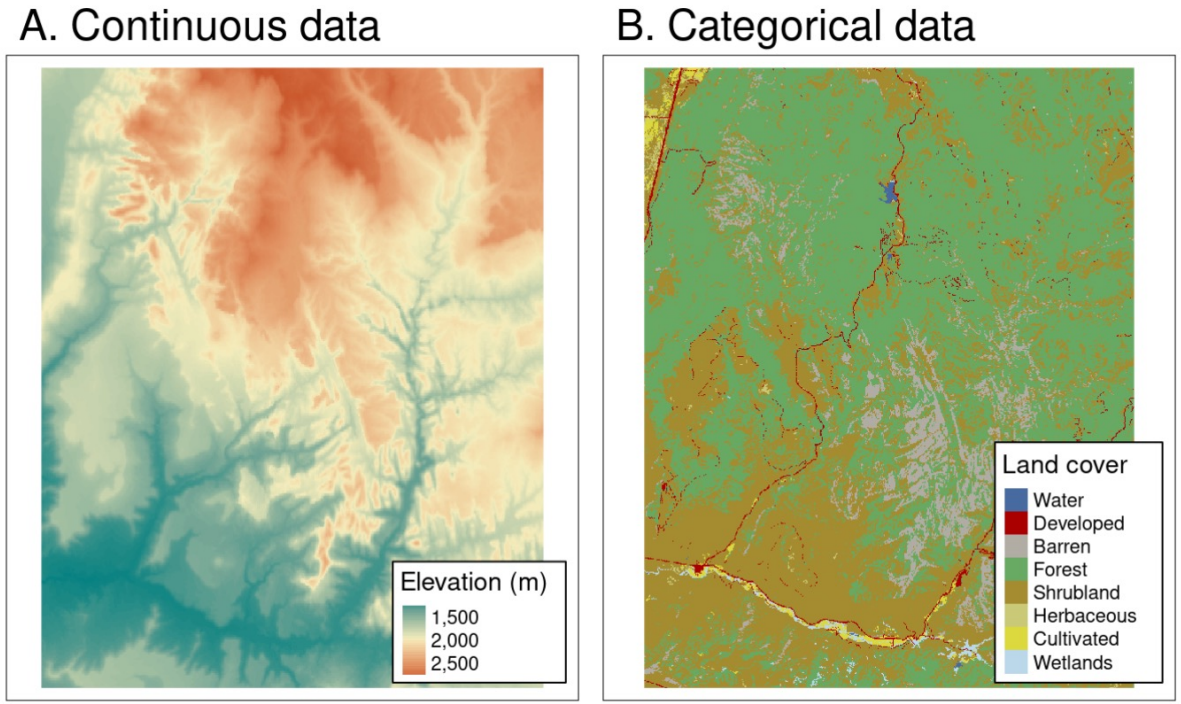
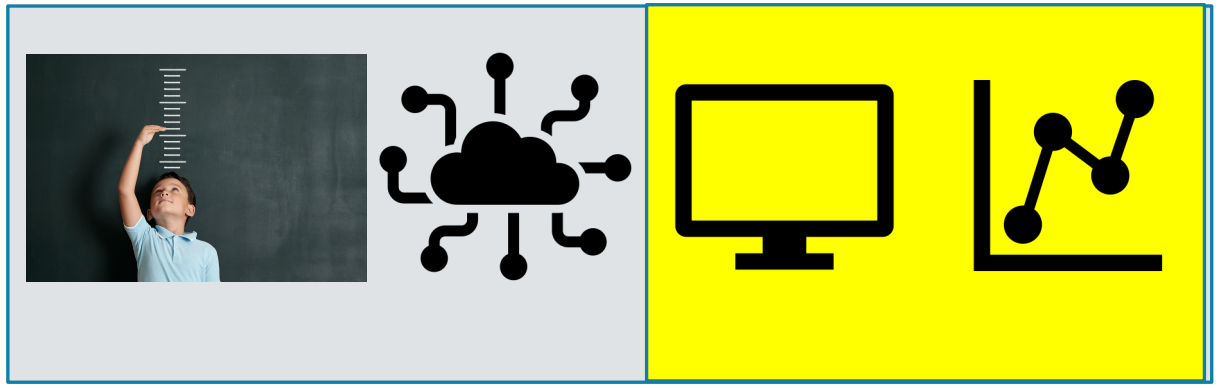


FIGURE 2.11: Examples of continuous and categorical rasters.

READING AND WRITING GEOSPATIAL DATA IN R



- Downloading data directly from your code is more reproducibility and precise
- Sometimes data is accessible within packages other times they provide an interface to an API

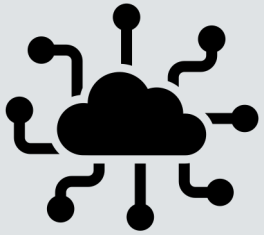
https://rdrr.io/cran/googleway/man/google_streetview.html

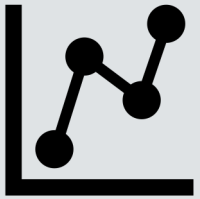
```
library(geodata)
worldclim_prec = worldclim_global("prec", res = 10,
path = tempdir())
class(worldclim_prec)
```

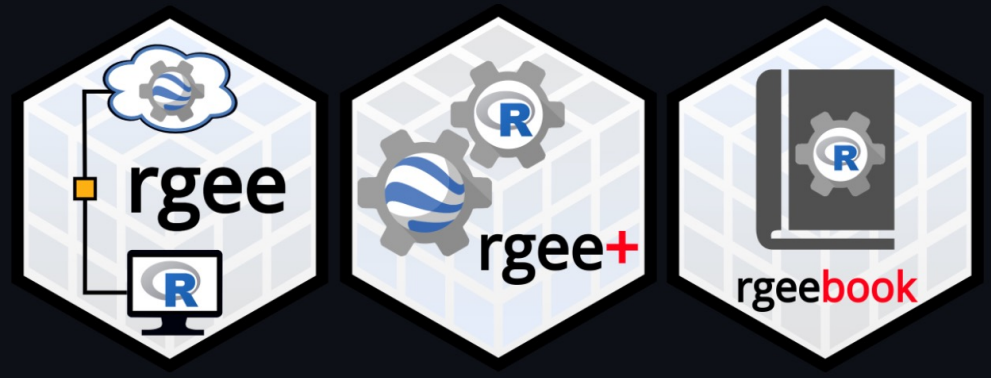
```
library(osmdata)
parks = opq(bbox = "leeds uk") %>%
  add_osm_feature(key = "leisure", value = "park") %>%
  osmdata_sf()
```

```
#*****************************************************
#GGGGGOOOOOOOOOOOOOOOOOOOOGGGGGGGGGLLLLLLLEEEEEEE API CALL
      jpeg(paste("images/user-", x$user_id[1], "-segment-",
coordinates(ptsL)[1,2], "-", coordinates(ptsL)[1,1], "-.jpeg",
sep=""), width=10, height=10, units="in", res=300)
        xx <- google_streetview(location = c(coordinates(ptsL)[1,2],
coordinates(ptsL)[1,1]),
                                size = c(400,400),
                                panorama_id = NULL,
                                output = "plot",
                                heading = Bearing,
                                fov = 60,
                                pitch = 0,
                                response_check = FALSE,
                                key = key)
      dev.off()
#*****************************************************
#GGGGGOOOOOOOOOOOOOOOOOOOOGGGGGGGGGLLLLLLLEEEEEEE API CALL
```

```
## getSoilMoisture.R
## C. Robertson
## Purpose: Fetch soil moisture data for specified areas, write to file
##
##
################################################

#run on Sunday - check to see when the API updates the file
library(raster)
library(RCurl)
library(rgdal)
setwd("/Users/colinr23/Dropbox/citsci/wht/AI-Portal/AI-RPortal")
# base fetch URL
baseURL <- "http://www.agr.gc.ca/atlas/data_donnees/geo/aafcPsssm/tif/anomaly/2017/weekly/"
baseName <- "SMDiffAvg_SMUDP2_2017_Week_"
# increment weekNum
dt2 <- as.POSIXlt(Sys.Date())
weekNum <- (dt2$yday %/%  7)
fetchURL <- paste(baseURL, baseName, weekNum, "_M5.tif", sep="")

#check its there
#Go get it
z <- ""
try(z <- getBinaryURL(fetchURL, failonerror = TRUE), silent=TRUE)
if (length(z) > 1) {download.file(fetchURL, destfile = paste("data/soilmoisture/", baseName, weekNum, "_M5.tif", sep=""))
} else {print(paste(fetchURL, " doesn't exist", sep =  ""))}

Sys.sleep(15) #take a rest

if (length(z) > 1) {
  #call function or script to process file
  weekNum <- "43"
  x <- raster(paste("data/soilmoisture/", baseName, weekNum, "_M5.tif", sep=""))
  sr <- readOGR('./data/basemap', "study-sites")
  srCRS <- proj4string(sr)
  sr <- spTransform(sr, proj4string(x))
  x1 <- crop(x, extent(sr)) #crop to extent of study sites
  srAB <- crop(x1, extent(sr[1,]))
  srBC <- crop(x1, extent(sr[2,]))
  srAB <- projectRaster(srAB, crs=srCRS)
  srBC <- projectRaster(srBC, crs=srCRS)
  writeRaster(srAB, paste("data/soilmoisture/current_SM-AB-", weekNum, "-.tif", sep=""), format="GTiff", overwrite=TRUE)
  writeRaster(srBC, paste("data/soilmoisture/current_SM-BC-", weekNum, "-.tif", sep=""), format="GTiff", overwrite=TRUE)
}
rm(z)
```

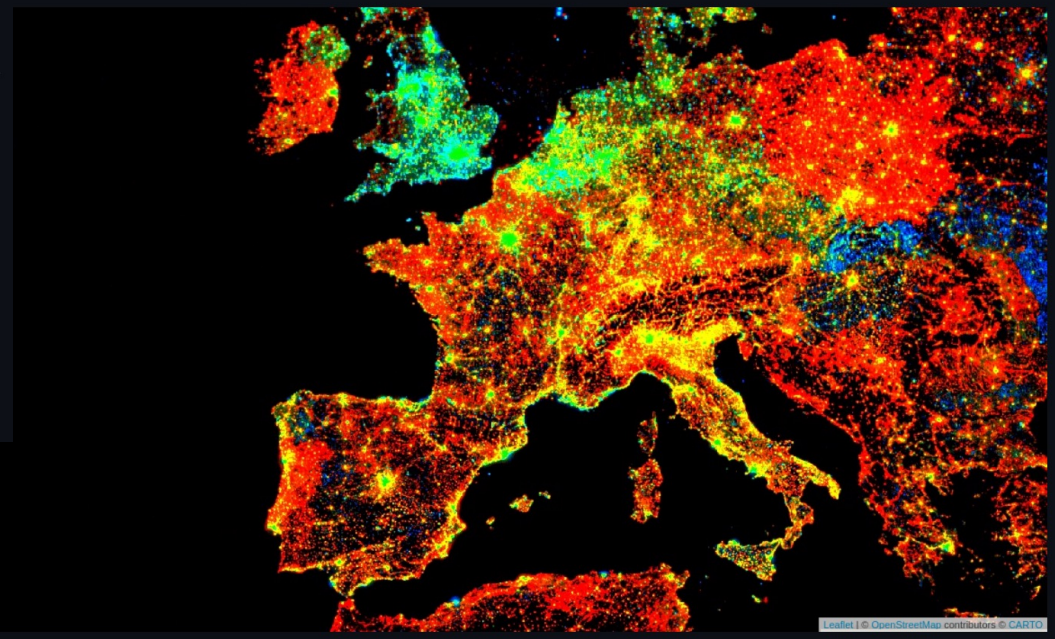






rgee: Google Earth Engine for R

rgee is an R binding package for calling Google Earth Engine API from within R. Various functions are implemented to simplify the connection with the R spatial ecosystem.



2. Extract precipitation values

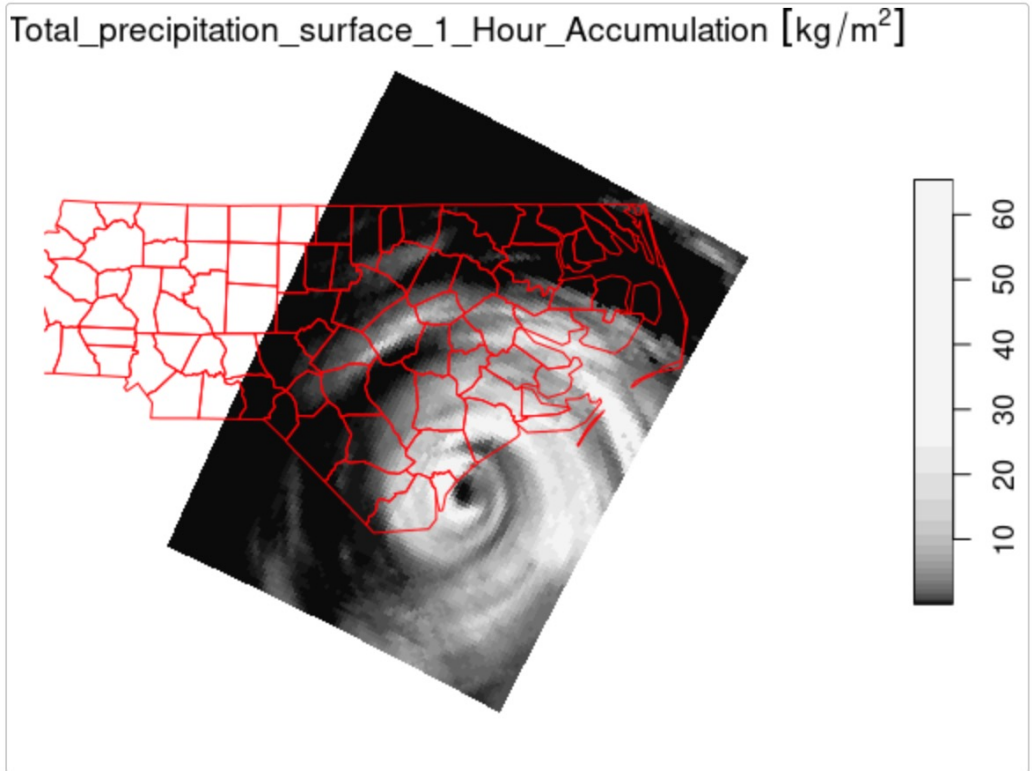
ADDITIONAL TOPICS

- working with online image collections via spatial-temporal asset catalogs
  - encapsulate queryable metadata for large volumes of image data
  - cloud-optimized geotiffs (COGS)
- NetCDF is a widely used format for storing array-based data as variables with dimensions and attrib utes
  - climate model output for example

tidync: A Tidy Approach to 'NetCDF' Data Exploration and Extraction

Tidy tools for 'NetCDF' data sources. Explore the contents of a 'NetCDF' source (file or URL) presented as varia interactive function translates the filter value or index expressions to array-slicing form. No data is read until exp hyper_array().

Version:	0.2.4
Depends:	R (≥ 3.5.0)
Imports:	dplyr (≥ 0.7.0), forcats, magrittr, ncdf4, ncmeta (≥ 0.2.0), purrr, RNetCDF (≥ 1.9-1), rlang, t
Suggests:	ggplot2, knitr, rmarkdown, testthat, covr, cubelyr
Published:	2020-05-12
Author:	Michael Sumner [aut, cre], Simon Wotherspoon [ctb], Tomas Remenyi [ctb], Ben Raymond
Maintainer:	Michael Sumner <mdsumner at gmail.com>
BugReports:	https://github.com/ropensci/tidync/issues
License:	GPL-3
URL:	https://docs.ropensci.org/tidync/
NeedsCompilation:	no
Materials:	I
CRAN checks:	t



https://cran.r-project.org/web/packages/stars/vignettes/stars1.html

SUMMARY

- Double check all imported data especially if working with a new dataset/source
- Even if it is harder, try to read data in programmatically – this leads to more transparent and reproducible workflows
- Be cautious of data type differences if you are writing data to external sources
  - e.g., databases
- Be extra cautious when handling date and time data
  - differences in time formats, timezones, database time vs. measurement time, sampling frequency changes, etc.

GG 606 SCIENTIFIC DATA WRANGLING

1. Jan 13: i/o