

APPLICATIONS



OF DATA SCIENCE

The Tidyverse

Applications of Data Science - Class 1

Giora Simchoni

gsimchoni@gmail.com and add #dsapps in subject

Stat. and OR Department, TAU

2021-12-29

APPLICATIONS



OF DATA SCIENCE

I don't need to know about
wrangling data, I get by.

APPLICATIONS



OF DATA SCIENCE

So, what's wrong with Excel?

(MS Excel is one amazing software. But it lacks:)

- Structure (or rather, structure is up to the user)
- Types to variables
- Automation (you could learn VBA Excel, but the horror)
- Reproducibility
- Open Source
- Extensibility
- Speed and Scale
- Modeling (there *is* a t-test, but the horror)

MS Excel might be the most dangerous software on the planet (Tim Worstall, Forbes)

So, what's wrong with base R?

(Base R is one amazing software. But it lacks:)

- Consistency:
 - Function names
 - Function arguments names
 - Function arguments order
 - Function return types (sometimes the same function!)
- Meaningful errors and warnings
- Good choices of default values to arguments
- Speed
- Good and easy visualizations
- One other thing

(In) Consistency - Example 1: Strings

```
# split a string by pattern: strsplit(string, pattern)
strsplit("Who dis?", " ")
```

```
## [1]
## [1] "Who"  "dis?"
```

```
# find if a pattern exists in a string: grepl(pattern, string)
grepl("di", "Who dis?")
```

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

```
# substitute a pattern in a string: sub(pattern, replace, string)
sub("di", "thi", "Who dis?")
```

```
## [1] "Who this?"
```

```
# length of a string: nchar(string); length of object: length(obj)
c(nchar("Who dis?"), length("Who dis?"))
```

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

(In) Consistency - Example 2: Models

```
n <- 10000  
x1 <- runif(n)  
x2 <- runif(n)  
t <- 1 + 2 * x1 + 3 * x2  
y <- rbinom(n, 1, 1 / (1 + exp(-t)))
```

```
glm(y ~ x1 + x2, family = "binomial")
```

```
glmnet(as.matrix(cbind(x1, x2)), as.factor(y), family = "binomial")
```

```
randomForest(as.factor(y) ~ x1 + x2)
```

```
gbm(y ~ x1 + x2, data = data.frame(x1 = x1, x2 = x2, y = y))
```



(Un) Meaningful Errors - Example

```
df <- data.frame(Education = 1:5, Ethnicity = c(2, 4, 5, 2, 1))
table(df$Education, df$Ethnicity)

## Error in table(df$Education, df$Ethnicity): all arguments must have the same length
```

(Bad) Default Values - Example

	A	B	C	
1	col1	col2	col3	
2		1	0.2	a
3		2	0.3	b
4		3	0.4	c
5		4	0.5	d
6				

```
# In R 3.6... in R 4.0 this was fixed!
df <- read.csv("../data/bad_args_test.csv")
df$col3
```

```
## [1] a b c d
## Levels: a b c d
```

```
df <- read.csv("../data/bad_args_test.csv", stringsAsFactors = FALSE)
df$col3
```

```
## [1] "a" "b" "c" "d"
```

(No) Speed - Example

```
file_path <- "../data/mediocre_file.csv"
df <- read.csv(file_path)
dim(df)
```

```
## [1] 9180    14
```

```
library(microbenchmark)

microbenchmark(
  read_base = read.csv(file_path),
  read_tidy = read_csv(file_path, col_types = cols()),
  read_dt = data.table::fread(file_path),
  times = 10)
```

```
## Unit: milliseconds
##          expr      min       lq     mean   median      uq     max neval
##  read_base 37.2907 38.3556 40.74187 40.63785 41.5849 47.3025    10
##  read_tidy 26.6109 28.2304 37.22088 29.03210 30.2974 112.7317    10
##  read_dt   8.3234  8.4668 14.33845  8.64185 10.4102  61.5332    10
```

Detour: The OKCupid Dataset

APPLICATIONS



OF DATA SCIENCE

The OKCupid Dataset

- ~60K active OKCupid users scraped on June 2012
- 35K Male, 25K Female (less awareness for non-binary back then)
- Answers to questions like:
 - Body Type
 - Diet
 - Substance Abuse
 - Education
 - Do you like pets?
 - Open questions, e.g. "On a typical Friday night I am..."
 - And the more boring demographic details like age, height, location, sign, religion etc.
- See [here](#) for the full codebook

BTW

Letter to the Editor

A Letter to the *Journal of Statistics and Data Science Education* — A Call for Review of “OkCupid Data for Introductory Statistics and Data Science Courses” by Albert Y. Kim and Adriana Escobedo-Land

Tiffany Xiao & Yifan Ma

Pages 214-215 | Published online: 18 Jun 2021

 Download citation  <https://doi.org/10.1080/26939169.2021.1930812>

See [here](#).

End of Detour

APPLICATIONS

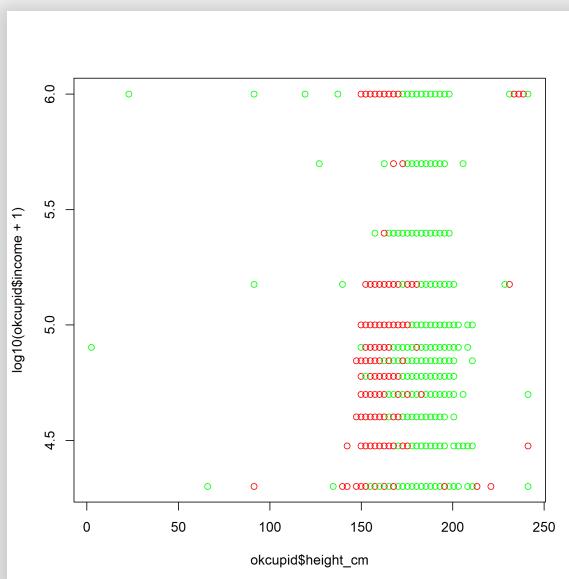


OF DATA SCIENCE

(Not) Good Vizualizations - Example

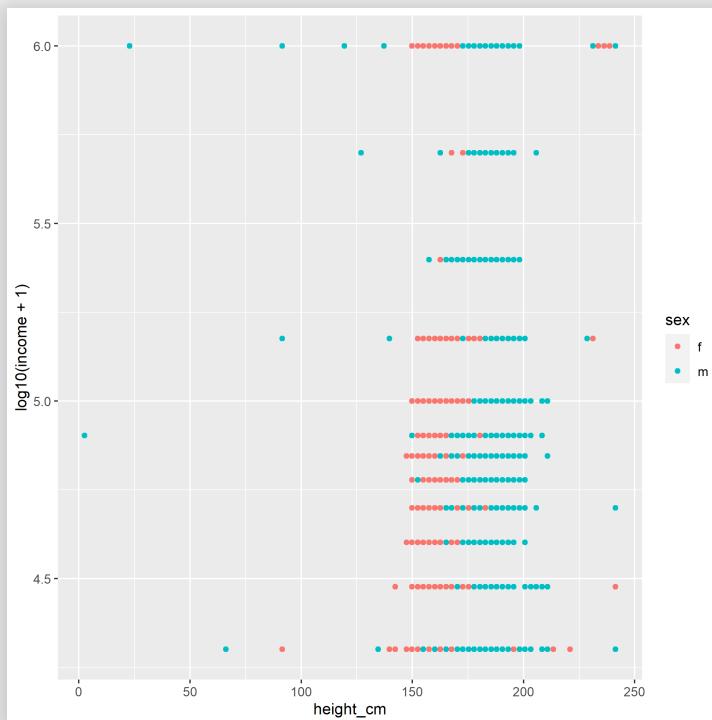
```
okcupid <- read_csv("~/okcupid.csv.zip", col_types = cols())
okcupid$income[okcupid$income == -1] <- NA
okcupid$height_cm <- okcupid$height * 2.54
```

```
plot(okcupid$height_cm, log10(okcupid$income + 1),
      col = c("red", "green") [as.factor(okcupid$sex) ])
```



```
ggplot(okcupid, aes(height_cm, log10(income + 1), color = sex)) +  
  geom_point()
```

Warning: Removed 48442 rows containing missing values (geom_point).



One other thing

Manager: "Give me the average income of women respondents above age 30 grouped by sexual orientation!"

You:

```
mean_bi <- mean(okcupid$income[okcupid$sex == "f" & okcupid$age >
mean_gay <- mean(okcupid$income [okcupid$sex == "f" & okcupid$age >
mean_straight <- mean(okcupid$income[okcupid$sex == "f" & okcupid$age >

data.frame(orientation = c("bisexual", "gay", "straight"),
           income_mean = c(mean_bi, mean_gay, mean_straight))

##   orientation income_mean
## 1    bisexual    133421.05
## 2          gay     86489.36
## 3    straight     85219.74
```

Or the slightly better you:

```
mean_income_function <- function(orientation) {  
  mean(okcupid$income[okcupid$sex == "f" & okcupid$age > 30 & okcupid$income > 0])  
}  
  
mean_bi <- mean_income_function("bisexual")  
mean_gay <- mean_income_function("gay")  
mean_straight <- mean_income_function("straight")  
  
data.frame(orientation = c("bisexual", "gay", "straight"),  
           income_mean = c(mean_bi, mean_gay, mean_straight))
```

```
##   orientation income_mean  
## 1    bisexual 133421.05  
## 2        gay   86489.36  
## 3    straight  85219.74
```

Or the even better you:

```
orientations <- c("bisexual", "gay", "straight")
income_means <- numeric(3)

for (i in seq_along(orientations)) {
  income_means[i] <- mean_income_function(orientations[i])
}

data.frame(orientation = orientations, income_mean = income_means)

##   orientation income_mean
## 1    bisexual    133421.05
## 2          gay     86489.36
## 3    straight     85219.74
```

Or the best you:

```
okupid_females_over30 <- with(okupid, okupid[sex == "f" & age >  
aggregate(okupid_females_over30$income,  
by = list(orientation = okupid_females_over30$orientati  
FUN = mean, na.rm = TRUE)
```

```
##   orientation      x  
## 1    bisexual 133421.05  
## 2        gay   86489.36  
## 3    straight   85219.74
```

Manager: "What? Why would bisexual women have a higher income than straight or gay women? Could you add the median, trimmed mean, standard error and n?"

You: 🤷

The Tidyverse

APPLICATIONS



OF DATA SCIENCE

What *is* The Tidyverse?

The [tidyverse](#) is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

- **tibble:** the `data.frame` re-imagined
- **readr:** importing/exporting (mostly rectangular) data for humans
- **dplyr + tidyrr:** a grammar of data manipulation
- **purrr:** functional programming in R
- **stringr:** string manipulation
- **ggplot2:** a grammar of graphics

The above can all be installed and loaded under the `tidyverse` package:

```
library(tidyverse)
```

Many more:

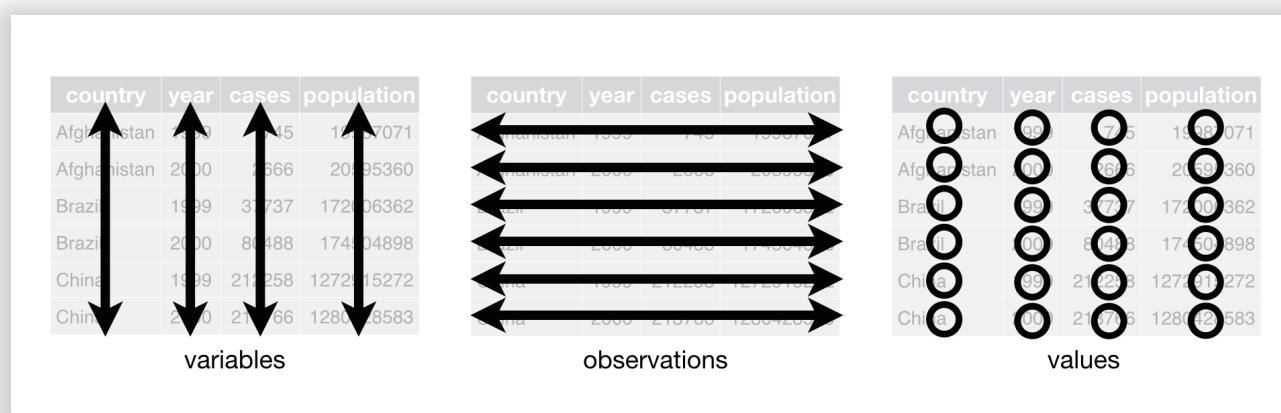
- `lubridate`: manipulating dates
- `tidymodels`: tidy modeling/statistics
- `rvest`: web scraping
- `tidytext`: tidy text analysis (life saver)
- `tidygraph + ggraph`: manipulating and plotting networks
- `glue`: print like a boss
- countless gg extensions (`ggmosaic`, `ggbbeeswarm`, `ganimate`, `ggridges` etc.)

What's so great about the Tidyverse?

- Tidy Data
- Consistency (in function names, args, return types, documentation)
- The Pipe
- Speed (C++ under the hood)
- ggplot2
- The Community

Tidy Data

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.



Which one of these datasets is tidy? (I)

table1

```
## # A tibble: 315 x 4
##   religion      yob n_straight n_total
##   <chr>        <dbl>      <dbl>     <dbl>
## 1 atheist       1950        26      29
## 2 buddhist      1950         6       6
## 3 christian     1950        28      32
## 4 hindu         1950         0       0
## 5 jewish         1950        21      24
## 6 muslim         1950         0       0
## 7 unspecified    1950        71      76
## 8 atheist        1951        31      33
## 9 buddhist       1951        11      11
## 10 christian     1951       23      24
## # ... with 305 more rows
```

Which one of these datasets is tidy? (II)

table2

```
## # A tibble: 630 x 4
##   religion    yob type     n
##   <chr>      <dbl> <chr>   <dbl>
## 1 atheist     1950 straight 26
## 2 atheist     1950 total   29
## 3 buddhist    1950 straight 6
## 4 buddhist    1950 total   6
## 5 christian   1950 straight 28
## 6 christian   1950 total   32
## 7 hindu       1950 straight 0
## 8 hindu       1950 total   0
## 9 jewish      1950 straight 21
## 10 jewish     1950 total   24
## # ... with 620 more rows
```

Which one of these datasets is tidy? (III)

```
table3
```

```
## # A tibble: 315 x 3
##   religion      yob pct_straight
##   <chr>        <dbl> <chr>
## 1 atheist      1950 26/29
## 2 buddhist     1950 6/6
## 3 christian    1950 28/32
## 4 hindu        1950 0/0
## 5 jewish       1950 21/24
## 6 muslim        1950 0/0
## 7 unspecified  1950 71/76
## 8 atheist      1951 31/33
## 9 buddhist     1951 11/11
## 10 christian   1951 23/24
## # ... with 305 more rows
```

Which one of these datasets is tidy? (IV)

table4

```
## # A tibble: 7 x 91
##   religion    n_total_1950 n_total_1951 n_total_1952 n_total_1953 n_total_1954
##   <chr>          <dbl>        <dbl>        <dbl>        <dbl>
## 1 atheist         29          33          34          37
## 2 buddhist        6           11          14          16
## 3 christian       32          24          37          47
## 4 hindu           0           0           0           1
## 5 jewish          24          29          27          23
## 6 muslim           0           0           0           0
## 7 unspecified     76          79          83          97
## # ... with 85 more variables: n_total_1955 <dbl>, n_total_1956 <dbl>,
## #   n_total_1957 <dbl>, n_total_1958 <dbl>, n_total_1959 <dbl>,
## #   n_total_1960 <dbl>, n_total_1961 <dbl>, n_total_1962 <dbl>,
## #   n_total_1963 <dbl>, n_total_1964 <dbl>, n_total_1965 <dbl>,
## #   n_total_1966 <dbl>, n_total_1967 <dbl>, n_total_1968 <dbl>,
## #   n_total_1969 <dbl>, n_total_1970 <dbl>, n_total_1971 <dbl>,
## #   n_total_1972 <dbl>, n_total_1973 <dbl>, n_total_1974 <dbl>, ...
```

Why Tidy?

Happy families are all alike; every unhappy family is unhappy in its own way. (Leo Tolstoy)

It allows R's vectorised nature to shine. (Hadley Wickham)

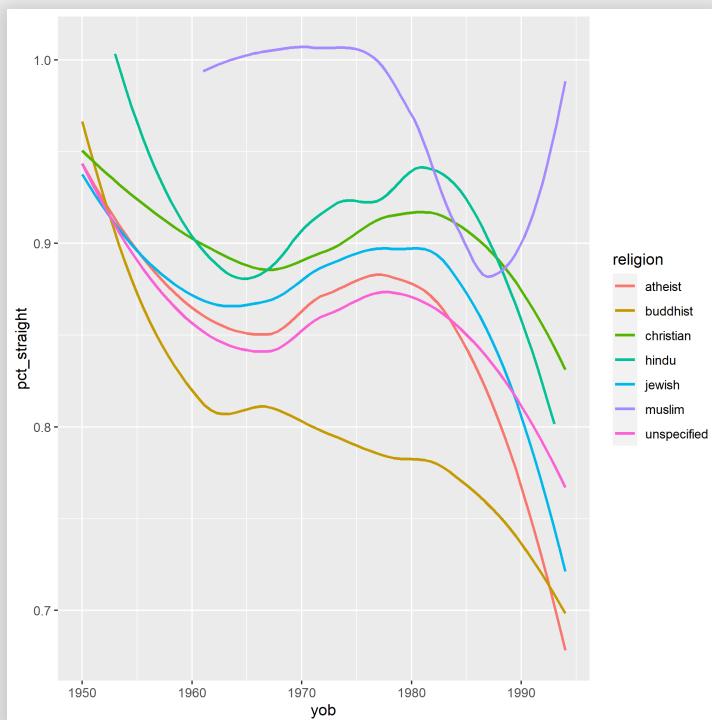
A Tidy dataset will be much easier to transform

```
table1$pct_straight = table1$n_straight / table1$n_total  
table1
```

```
## # A tibble: 315 x 5  
##   religion      yob n_straight n_total pct_straight  
##   <chr>        <dbl>     <dbl>     <dbl>        <dbl>  
## 1 atheist      1950       26       29       0.897  
## 2 buddhist     1950        6        6        1  
## 3 christian    1950       28       32       0.875  
## 4 hindu        1950        0        0       NaN  
## 5 jewish        1950       21       24       0.875  
## 6 muslim        1950        0        0       NaN  
## 7 unspecified   1950       71       76       0.934  
## 8 atheist       1951       31       33       0.939  
## 9 buddhist      1951       11       11        1  
## 10 christian    1951      23       24       0.958  
## # ... with 305 more rows
```

A Tidy dataset will be much easier to plot

```
ggplot(table1, aes(x = yob, y = pct_straight, color = religion)) +  
  geom_smooth(method = "loess", formula = y ~ x, se = FALSE)
```



Detour: The tibble

APPLICATIONS



OF DATA SCIENCE

The tibble: the data.frame re-imagined

- Prints nicer:

```
tib1 <- tibble(day = lubridate::today() + runif(1e3) * 30,  
               type = sample(letters, 1e3, replace = TRUE),  
               quantity = sample(seq(0, 100, 10), 1e3, replace = TRUE))  
tib1
```

```
## # A tibble: 1,000 × 3  
##   day       type  quantity  
##   <date>     <chr>    <dbl>  
## 1 2022-01-25 g        30  
## 2 2022-01-05 g        60  
## 3 2022-01-19 a        10  
## 4 2022-01-08 o        20  
## 5 2022-01-24 d        50  
## 6 2022-01-04 e        70  
## 7 2022-01-01 o        50  
## 8 2022-01-23 u        20  
## 9 2022-01-18 n         0  
## 10 2022-01-25 g       70  
## # ... with 990 more rows
```

```
df1 <- data.frame(day = lubridate::today() + runif(1e3) * 30,  
                   type = sample(letters, 1e3, replace = TRUE),  
                   quantity = sample(seq(0, 100, 10), 1e3, replace  
df1
```

```
##          day type quantity  
## 1 2022-01-04    p      70  
## 2 2022-01-03    r      20  
## 3 2022-01-23    v      60  
## 4 2022-01-07    p      30  
## 5 2022-01-01    r      50  
## 6 2022-01-18    u      90  
## 7 2022-01-09    i      50  
## 8 2022-01-02    n      40  
## 9 2022-01-27    c      90  
## 10 2022-01-14   o      0  
## 11 2022-01-03   w      50  
## 12 2022-01-20   q     100  
## 13 2022-01-22   r      80  
## 14 2022-01-07   g      90  
## 15 2022-01-20   u     100  
## 16 2022-01-09   e      50  
## 17 2022-01-05   f      60  
## 18 2022-01-07   y      10  
## 19 2022-01-01   k      70  
## 20 2022-01-25   p      20  
## 21 2021-12-31   y      50  
## 22 2022-01-24   u      60  
## 23 2022-01-20   q     10
```

- Warns you when you make mistakes (!):

```
tib1$quanittt
```

```
## Warning: Unknown or uninitialised column: `quanittt`.
```

```
## NULL
```

```
df1$quanittt
```

```
## NULL
```

- Can also create via `tribble()`:

```
tribble(  
  ~a, ~b, ~c,  
  "a", 1, 2.2,  
  "b", 2, 4.3,  
  "c", 3, 3.4  
)
```

```
## # A tibble: 3 × 3  
##   a         b     c  
##   <chr> <dbl> <dbl>  
## 1 a       1     2.2  
## 2 b       2     4.3  
## 3 c       3     3.4
```

- Can build on top of variables during creation:

```
tibble(x = 1:5, y = x^2)
```

```
## # A tibble: 5 x 2
##       x     y
##   <int> <dbl>
## 1     1     1
## 2     2     4
## 3     3     9
## 4     4    16
## 5     5    25
```

```
data.frame(x = 1:5, y = x^2)
```

```
## Error in data.frame(x = 1:5, y = x^2): object 'x' not found
```

- Will never turn your strings into factors, will never change your column names:

```
tib1 <- readr::read_csv("../data/bad_args_test.csv", col_types = c  
colnames(tib1)
```

```
## [1] "col1" "col2" "col3"
```

```
tib1$col3
```

```
## [1] "a" "b" "c" "d"
```

```
df1 <- read.csv("../data/bad_args_test.csv")  
colnames(df1)
```

```
## [1] "i..col1" "col2"      "col3"
```

```
df1$col3
```

```
## [1] "a" "b" "c" "d"
```

Though one ought to remember a `tibble` **is still** a `data.frame`:

```
class(tib1)  
  
## [1] "spec_tbl_df" "tbl_df"        "tbl"          "data.frame"  
  
class(df1)  
  
## [1] "data.frame"
```

End of Detour

APPLICATIONS



OF DATA SCIENCE

Consistency - Example: stringr

a cohesive set of functions designed to make working with strings as easy as possible.

```
strings_vec <- c("I'm feeling fine", "I'm perfectly OK",
                 "Nothing is wrong!")
str_length(strings_vec)
```

```
## [1] 16 16 17
```

```
str_c(strings_vec, collapse = ", ")
```

```
## [1] "I'm feeling fine, I'm perfectly OK, Nothing is wrong!"
```

```
str_sub(strings_vec, 1, 3)
```

```
## [1] "I'm" "I'm" "Not"
```

```
str_detect(strings_vec, "I'm")
## [1] TRUE TRUE FALSE

str_replace(strings_vec, "I'm", "You're")
## [1] "You're feeling fine" "You're perfectly OK" "Nothing is wrong!"

str_split("Do you know regex?", " ")
## [[1]]
## [1] "Do"       "you"      "know"     "regex?"

str_extract(strings_vec, "[aeiou]")
## [1] "e" "e" "o"

str_count(strings_vec, "[A-Z]")
## [1] 1 3 1
```

The Pipe

Remember you?

```
mean_bi <- mean(okcupid$income[okcupid$sex == "f" & okcupid$age >
mean_gay <- mean(okcupid$income[okcupid$sex == "f" & okcupid$age >
mean_straight <- mean(okcupid$income[okcupid$sex == "f" & okcupid$age >

data.frame(orientation = c("bisexual", "gay", "straight"),
           income_mean = c(mean_bi, mean_gay, mean_straight))

##      orientation income_mean
## 1      bisexual    133421.05
## 2          gay     86489.36
## 3    straight     85219.74
```

Doesn't this make much more sense?

```
okcupid %>%
  filter(sex == "f", age > 30) %>%
  group_by(orientation) %>%
  summarize(income_mean = mean(income, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   orientation income_mean
##   <chr>          <dbl>
## 1 bisexual      133421.
## 2 gay            86489.
## 3 straight       85220.
```

- Read as:
 - Take the OKCupid data,
 - Filter only women above the age of 30,
 - And for each group of sexual orientation,
 - Give me the average income

- Make verbs, not nouns
- Can always access the dataset last stage with ".":

```
okupid %>%
  filter(str_count(essay0) > median(str_count(.\$essay0)), na.rm = TRUE)
```

- Operates not just on data frames or tibbles:

```
strings_vec %>% str_to_title()
```

```
## [1] "I'm Feeling Fine"  "I'm Perfectly Ok"   "Nothing Is Wrong!"
```

- No intermediate objects
- Don't strive to make the longest possible pipe (though it is a fun experiment)
- Tools exist for debugging

And, if you want to throw in the n, the median:

```
okcupid %>%
  filter(sex == "f", age > 30) %>%
  group_by(orientation) %>%
  summarize(income_mean = mean(income, na.rm = TRUE),
            income_median = median(income, na.rm = TRUE),
            n = n())
```

```
## # A tibble: 3 x 4
##   orientation income_mean income_median     n
##   <chr>          <dbl>           <dbl> <int>
## 1 bisexual      133421.         50000    652
## 2 gay            86489.          40000    664
## 3 straight       85220.          60000   10436
```

And if you want this for the age as well:

```
okcupid %>%
  filter(sex == "f", age > 30) %>%
  group_by(orientation) %>%
  summarize(across(c(income, age),
    list(mean = mean, median = median), na.rm = TRUE))
```



```
## # A tibble: 3 x 5
##   orientation income_mean income_median age_mean age_median
##   <chr>        <dbl>        <dbl>       <dbl>       <dbl>
## 1 bisexual     133421.      50000       37.8       36
## 2 gay          86489.       40000       40.4       38
## 3 straight     85220.       60000       40.7       38
```

Now *this* is a language for Data Science.

But we're getting ahead of ourselves.

In fact!

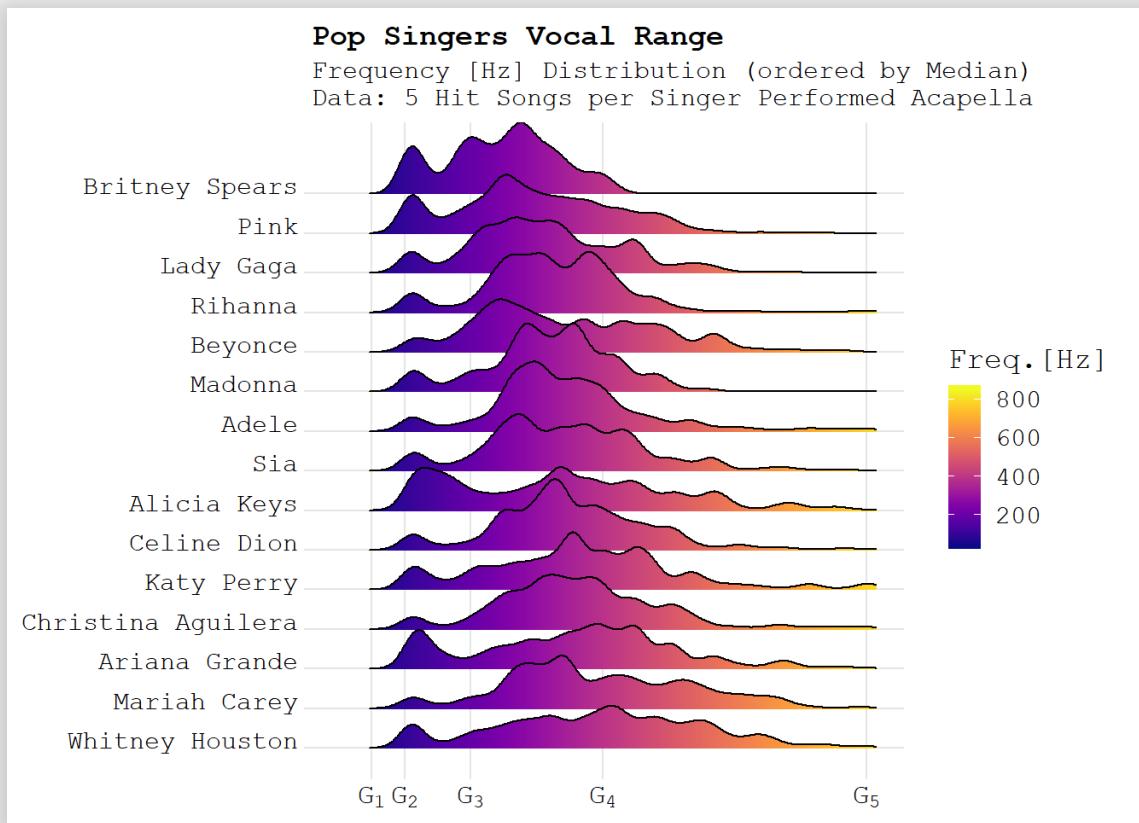
This became so popular that since R 4.1 there is a built-in pipe operator:

```
okcupid |>
  filter(sex == "f", age > 30) |>
  group_by(orientation) |>
  summarize(across(c(income, age),
    list(mean = mean, median = median), na.rm = TRUE))
```



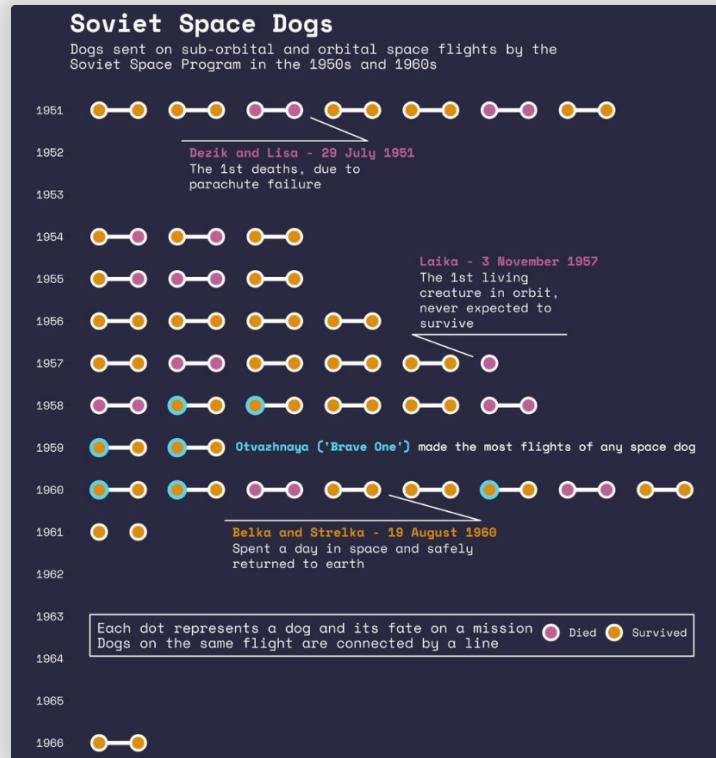
```
## # A tibble: 3 x 5
##   orientation income_mean income_median age_mean age_median
##   <chr>          <dbl>        <dbl>      <dbl>       <dbl>
## 1 bisexual      133421.      50000     37.8        36
## 2 gay            86489.       40000     40.4        38
## 3 straight       85220.       60000     40.7        38
```

ggplot2



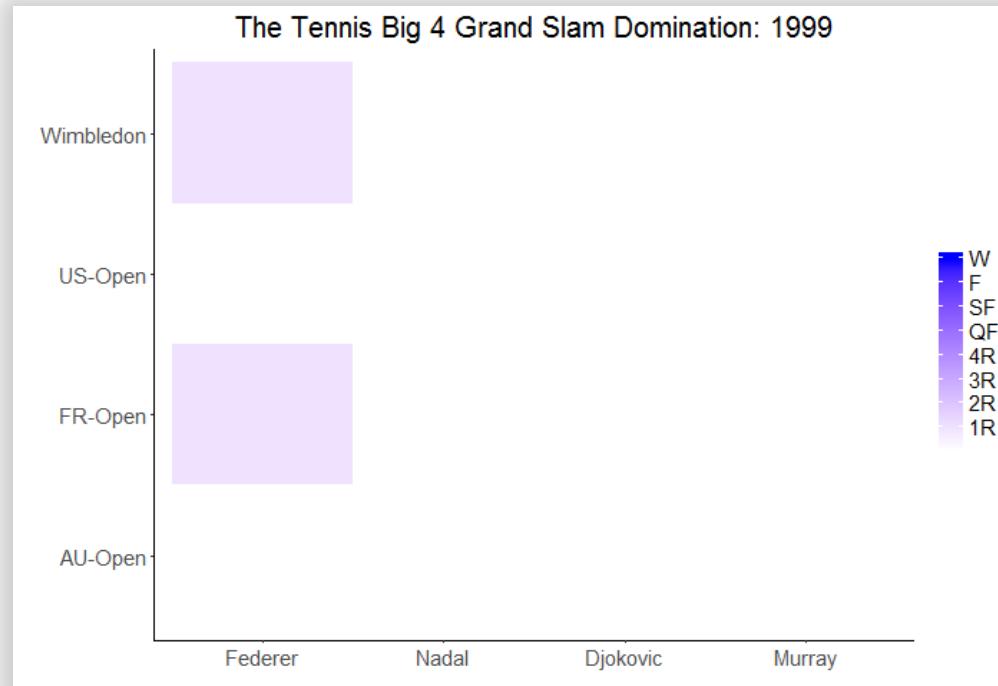
[Ave Mariah / Giora Simchoni](#)

ggplot2



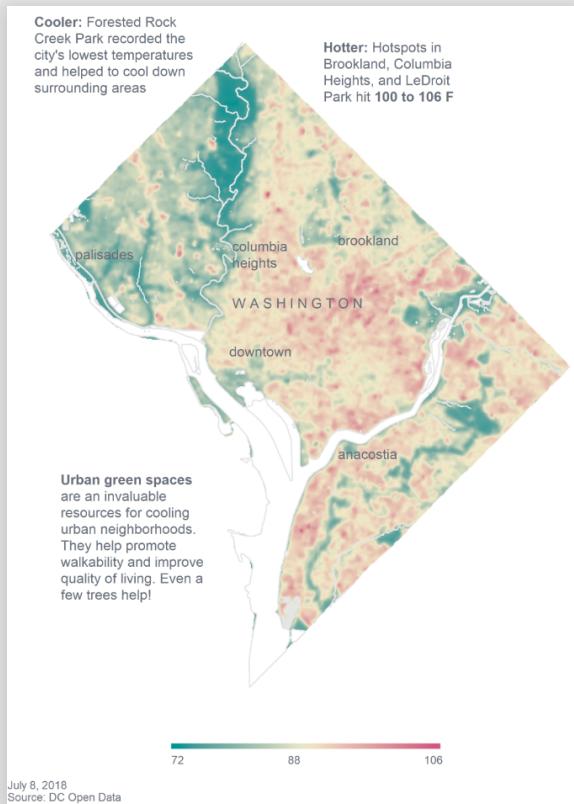
[Soviet Space Dogs / David Smale](#)

ggplot2



[Federer, Nadal, Djokovic and Murray, Love. / Giora Simchoni](#)

ggplot2



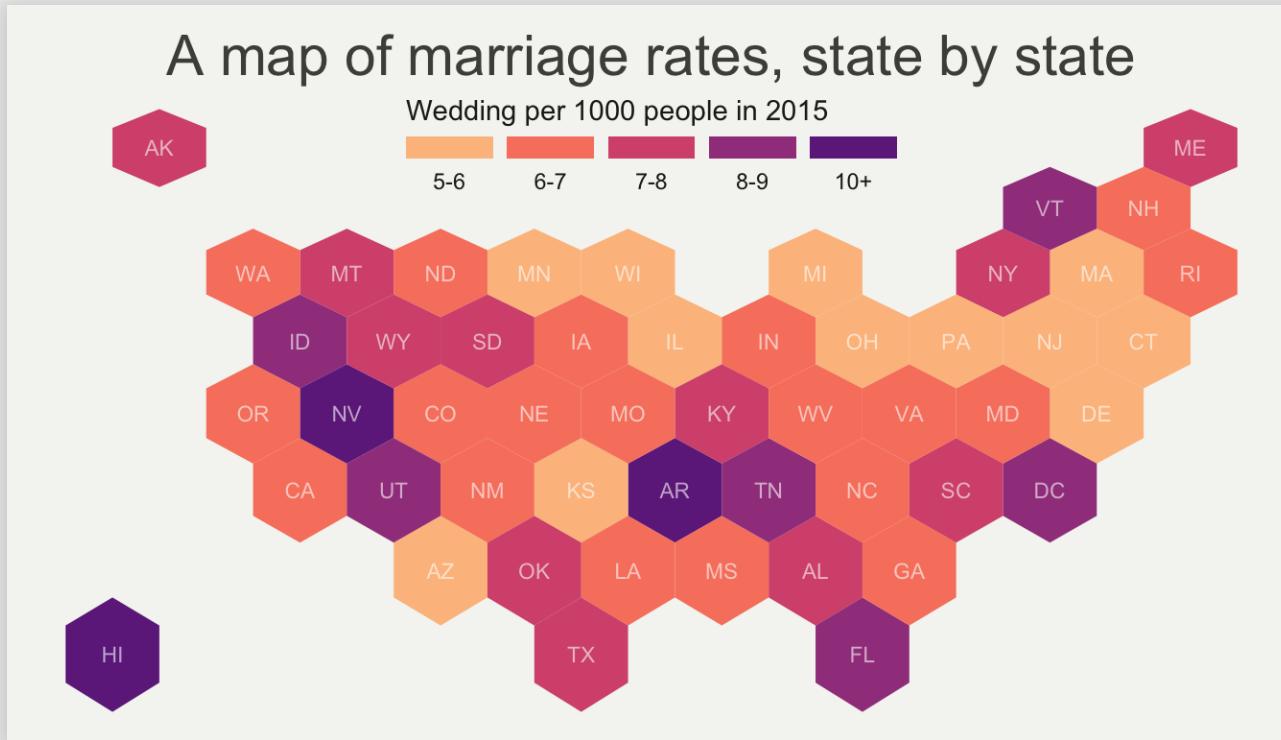
[NYT-style urban heat island maps / Katie Jolly](#)

ggplot2

A map of marriage rates, state by state

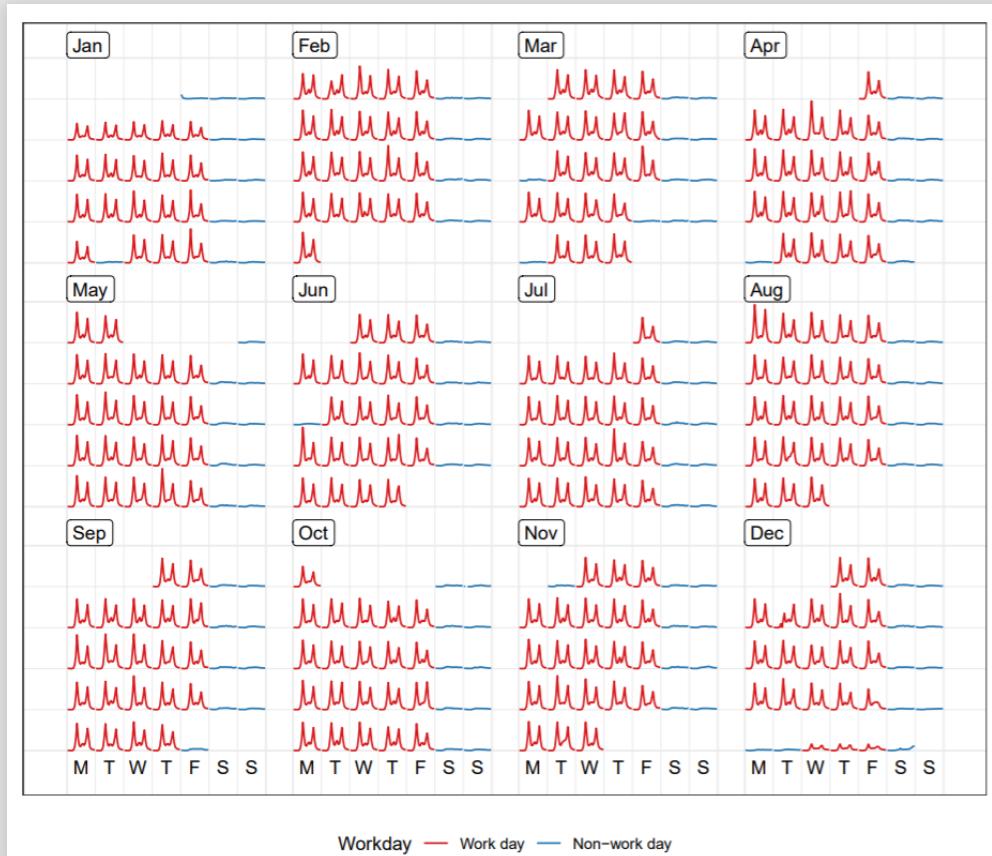
Wedding per 1000 people in 2015

5-6 6-7 7-8 8-9 10+



[A map of marriage rates, state by state / Unknown](#)

ggplot2



The Community

- 100% Open Source on Github
 - Cheatsheet for everything
 - Documentation for humans, Packages websites, Webinars, Free Books (start with [R4DS](#))
 - [Rstudio Community forum](#)
 - [RLadies](#) worldwide branches (who will pick up the  and create RLadies TLV?)
 - Very strong on Twitter
[#rstats](#)

String manipulation with string::CHEAT SHEET

The `string` package provides a set of internally-consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches

- str_detect(string, pattern)**: Detect if any of the elements in a string vector contains a matching pattern. Returns a logical vector.
- str_extract(string, pattern)**: Find the first element(s) of a string vector that match a regular expression pattern.
- str_count(string, pattern)**: Count the number of times a pattern occurs in a string or character vector.
- str_locate(string, pattern)**: Compute the positions of patterns in a string or character vector.
- str_extract_all(string, pattern)**: Extract all non-overlapping matches of a pattern from a string or character vector.

Subset Strings

- str_sub(string, start = 1, end = -1)**: Extract a subset of characters from a string or character vector.
- str_collapse(string, pattern)**: Remove the extra characters between consecutive matches of a regular expression pattern.
- str_substring(string, pattern)**: Return only the portion of a string that matches a regular expression pattern.
- str_peek(string, pattern)**: Return the first pattern matches that fit in each string, as a vector. Also returns the index of the first match for each pattern.
- str_matching(pattern)**: Returns the first pattern matches that fit in each string, as a vector. Also returns the index of the first match for each pattern.
- str_extract_group(string, pattern)**: Extracts multiple patterns with a column for each group in pattern. Returns a data frame.

Manage Lengths

- str_length(string)**: The length of a string or character vector, which generally equals the number of characters in str_length(string).
- str_pad(string, width, side = c("left", "right"))**: Pad a string to a specified width, either to the left or right of the original string.
- str_strip(string, side = c("left", "right"))**: Trim whitespace from the start or end of a string, respectively.
- str_l�ad(string, n = 1, side = c("start", "end"))**: Append or prepend a string to the start or end of another string.

Mutate Strings

- str_replace(string, pattern, replacement)**: Replace substrings by identifying the subunits to be replaced and replacing them with replacement.
- str_replace_all(string, replacement)**: Replace the first occurrence of each string in pattern with replacement.
- str_replace_na(string, pattern)**: Replace missing values in a character vector with a specified value.
- str_collapse(string, pattern)**: Collapse multiple strings into a single string.
- str_flatten(string, pattern)**: Collapse multiple strings into a single string.
- str_flatten_all(string, pattern)**: Flatten all levels of nesting in a character vector.
- str_gsub(string, pattern)**: Replace a series of strings in a series of nested strings with a replacement string.
- str_gsub_all(string, pattern)**: Replace all levels of nesting in a character vector.
- str_glue(string, pattern)**: Create a string or strings in a series of nested strings with a replacement string.
- str_collapse(string, pattern)**: Create a string or strings in a series of nested strings with a replacement string.
- str_collapse_na(string, pattern)**: Create a string or strings in a series of nested strings with a replacement string.
- str_collapse(string, pattern)**: Convert strings to the blank string, str_collapse(string).

Join and Split

- str_c(...)**: Join multiple strings into a single string.
- str_collapse(string, pattern)**: Collapse multiple strings into a single string.
- str_collapse_all(string, pattern)**: Collapse all levels of nesting in a character vector.
- str_flatten(string, pattern)**: Flatten all levels of nesting in a character vector.
- str_flatten_all(string, pattern)**: Flatten all levels of nesting in a character vector.
- str_glue(string, pattern)**: Create a string or strings in a series of nested strings with a replacement string.
- str_glue_all(string, pattern)**: Create a string or strings in a series of nested strings with a replacement string.
- str_gsub(string, pattern)**: Replace a series of strings in a series of nested strings with a replacement string.
- str_gsub_all(string, pattern)**: Replace all levels of nesting in a character vector.
- str_glue(string, pattern)**: Create a string or strings in a series of nested strings with a replacement string.
- str_gsub(string, pattern)**: Replace a series of strings in a series of nested strings with a replacement string.
- str_gsub_all(string, pattern)**: Replace all levels of nesting in a character vector.

Order Strings

- str_nchar(string, decreasing = FALSE)**: Sort a character vector based on the length of its elements.
- str_order(string, decreasing = FALSE)**: Returns the indices of a character vector that sort it in a character vector.
- str_rank(string, decreasing = FALSE)**: Returns the rank of each element in a character vector.

Helpers

- str_collapse(encoding)**: Override the encoding of a string, str_collapse(UTF8).
- str_view(string, pattern)**: Extract a portion of a string based on a regular expression pattern.
- str_view(string, width = 10, end = 1)**: Extract a portion of a string starting at width and ending at end.
- str_view(string, pattern, match = 1)**: View a portion of a string based on a regular expression pattern.
- str_wrap(string, width = 80, end = 1)**: Wrap a string into multiple lines terminated by newlines.

R Studio logo is trademark of RStudio, Inc. © 2021 by RStudio, Inc. All rights reserved. This document is provided under the terms of the MIT license. See https://rstudio.com/terms/ for details. This document is part of the [RStudio Cheat Sheets](#) collection. See https://rstudio.com/cheatsheets/ for a complete list of contents.