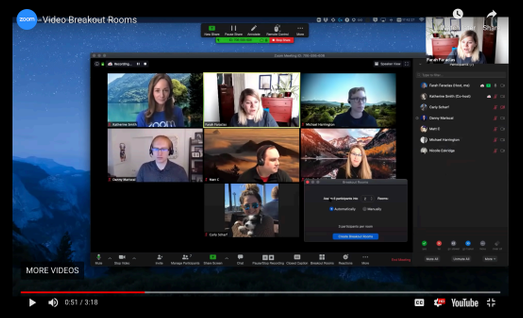
Non Numeric Data in R

Exercise: What do you already know?

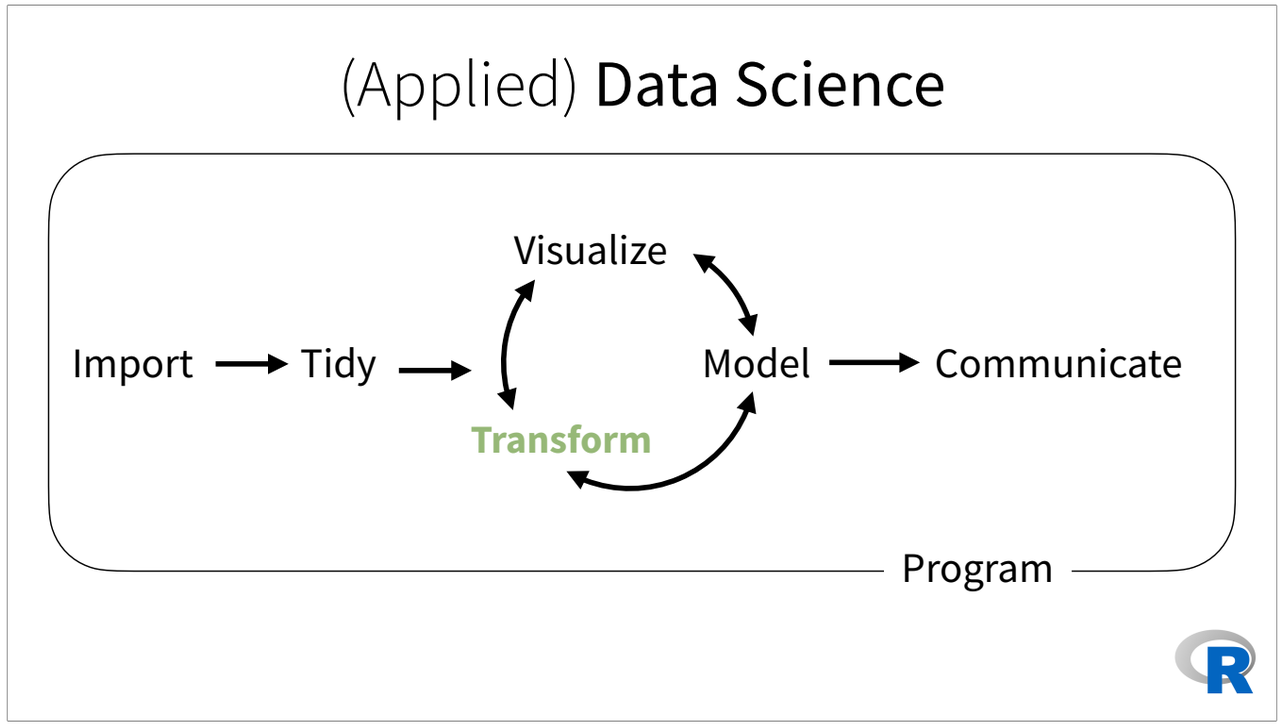
- In the Chat Window type:

  1. What types of non-numeric data do you need to work with / manipulate?

  2. What functions / packages do you use to work with them?



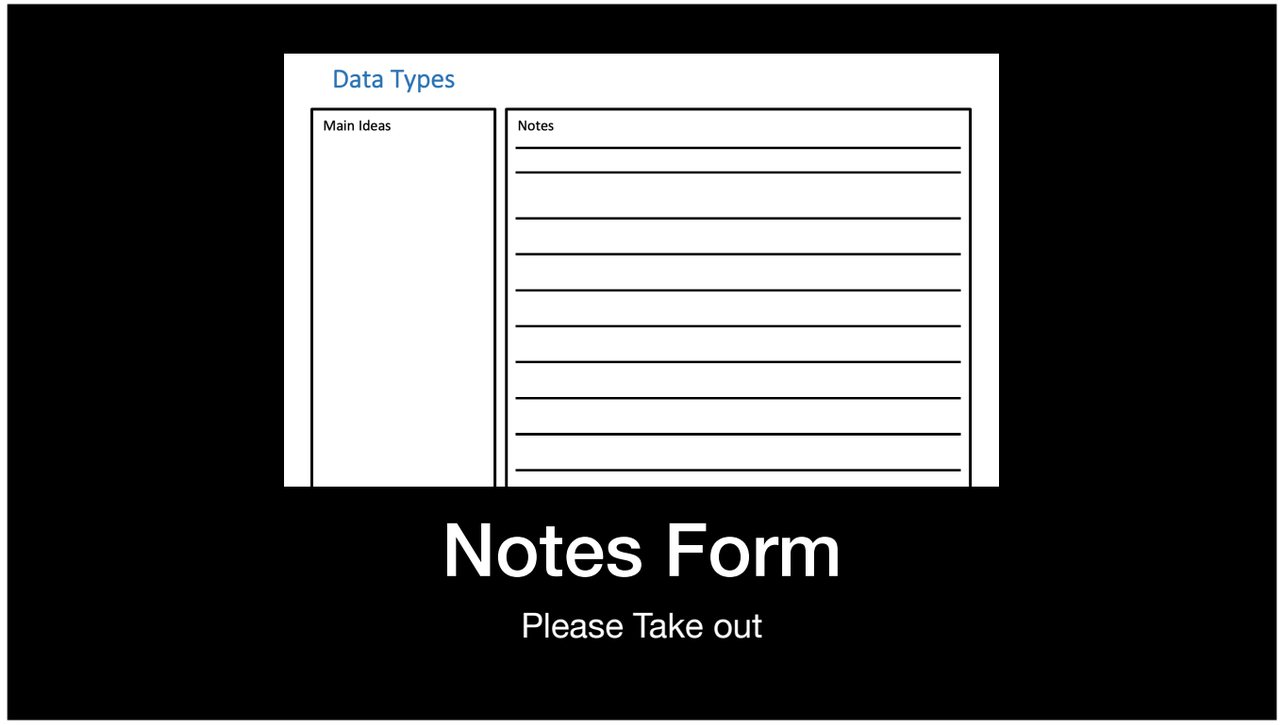
Split into group for this.



At the start of the workshop I mentioned that there are only two packages in the Tidyverse that you really need to learn: dplyr and ggplot2. I also mentioned that dplyr is the main package for data transformation.

The caveat is that up until now we've only worked with numeric data. We've been asking questions like "How many babies were given a specific name in a specific year?" But there's a ton of non-numeric data that is useful to analyze, and today we'll be focusing on that.

This is analogous to how we approached importing data. "We can analyze data that is already in R. If we can learn to import data into R, then we can do so much more." Similarly, if we can learn to analyze non-numeric data, then we can do so much more (again!).


Data Types
Main Ideas
Notes
Notes Form
Please Take out

Open **Non-Numeric-Data.Rmd**

Quiz

What types of data are in this data set?

	time_hour	name	air_time	distance	day	delayed
1	2013-01-01 05:00:00	United Air Lines Inc.	13620s (~3.78 hours)	1400	Tuesday	TRUE
2	2013-01-01 05:00:00	United Air Lines Inc.	13620s (~3.78 hours)	1416	Tuesday	TRUE
3	2013-01-01 05:00:00	American Airlines Inc.	9600s (~2.67 hours)	1089	Tuesday	TRUE
4	2013-01-01 05:00:00	JetBlue Airways	10980s (~3.05 hours)	1576	Tuesday	FALSE
5	2013-01-01 06:00:00	Delta Air Lines Inc.	6960s (~1.93 hours)	762	Tuesday	FALSE
6	2013-01-01 05:00:00	United Air Lines Inc.	9000s (~2.5 hours)	719	Tuesday	TRUE
7	2013-01-01 06:00:00	JetBlue Airways	9480s (~2.63 hours)	1065	Tuesday	TRUE
8	2013-01-01 06:00:00	ExpressJet Airlines Inc.	3180s (~53 minutes)	229	Tuesday	FALSE
9	2013-01-01 06:00:00	JetBlue Airways	8400s (~2.33 hours)	944	Tuesday	FALSE
10	2013-01-01 06:00:00	American Airlines Inc.	8280s (~2.3 hours)	733	Tuesday	TRUE
11	2013-01-01 06:00:00	JetBlue Airways	8940s (~2.48 hours)	1028	Tuesday	FALSE

Please put your answer in the chat

Non Numeric Data in R

- Logical / Boolean
- Character / String
- Factor / Categorical
- Dates and Times / POSIXct

These are the 4 most important non-numeric data types that you're likely to encounter. We'll be covering each of them today. Each of these datatypes has multiple names, so I'm including a few of those names here.

Logicals

R's data type for boolean values (i.e. TRUE and FALSE).

```
typeof(TRUE)
##  "logical"

typeof(FALSE)
##  "logical"

typeof(c(TRUE, TRUE, FALSE))
##  "logical"
```

I often use "class" where Garrett uses "typeof"

```r
flights %>%
  mutate(delayed = arr_delay > 0) %>%
  select(arr_delay, delayed)
```

arr_delay <dbl>	delayed <lgl>
11	TRUE
20	TRUE
33	TRUE
-18	FALSE
-25	FALSE
12	TRUE
19	TRUE
-14	FALSE
-8	FALSE
8	TRUE

This is an example of creating a logical value from an expression on another column. This is very common task.

```
flights %>%
  mutate(delayed = arr_delay > 0) %>%
  select(arr_delay, delayed)
```

arr_delay <dbl>	delayed <lgl>
11	TRUE
20	TRUE
33	TRUE
-18	FALSE
-25	FALSE
12	TRUE
19	TRUE
-14	FALSE
-8	FALSE
8	TRUE

Can we compute the proportion of NYC flights that arrived late?

Take a moment to think about how you would answer this question.

Most useful skills

1. Math with logicals

With logical data, there is really only one operation you need to know: how to do math with them.

Math

When you do math with logicals, **TRUE becomes 1** and **FALSE becomes 0**.

Math

When you do math with logicals, **TRUE becomes 1** and **FALSE becomes 0**.

- The **sum** of a logical vector is the **count of TRUEs**

```
sum(c(TRUE, FALSE, TRUE, TRUE))
##  3
```

Math

When you do math with logicals, **TRUE becomes 1** and **FALSE becomes 0**.

- The **sum** of a logical vector is the **count of TRUEs**

```
sum(c(TRUE, FALSE, TRUE, TRUE))
##  3
```

- The **mean** of a logical vector is the **proportion of TRUEs**

```
mean(c(1, 2, 3, 4) < 4)
##  0.75
```

Perhaps demo

c(1, 2, 3, 4)

c(1, 2, 3, 4) < 4

mean(c(1, 2, 3, 4))

Your Turn 1

Use flights to create **delayed**, a variable that displays whether a flight was delayed (arr_delay > 0).

Then, remove all rows that contain an NA in **delayed**.

Finally, create a summary table that shows:

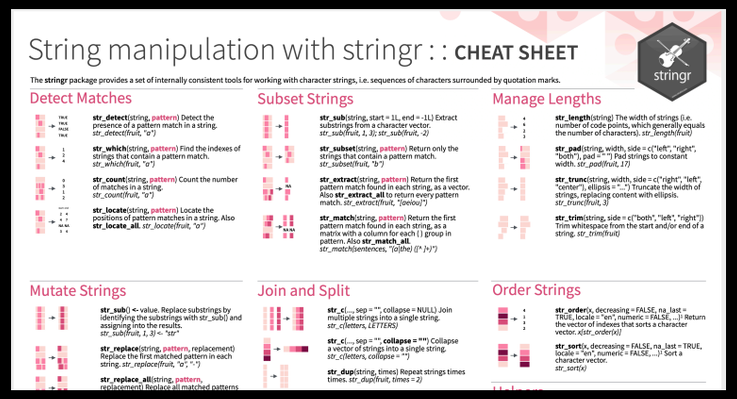
1. How many flights were delayed
2. What proportion of flights were delayed

```
flights %>%
  mutate(delayed = arr_delay > 0) %>%
  drop_na(delayed) %>%
  summarise(total = sum(delayed), prop = mean(delayed))
## # A tibble: 1 × 2
##   total      prop
##   <int>     <dbl>
## 1 133004 0.4063101
```

Non Numeric Data in R

- Logical / Boolean
- Character / String
- Factor / Categorical
- Dates and Times

There were only two operations we wanted to do with Logical values, and we were able to do them with base R. Now let's turn our attention to Strings.



stringr Cheat Sheet

Please Take out

Base R isn't great at working with strings. So we'll be using a separate package for it called stringr

(character) **strings**

Anything surrounded by quotes(") or single quotes(').

```
> "one"
> "1"
> "one's"
> '"Hello World"'
> "foo
+
+
+ oops. I'm stuck in a string."
```

Warm Up

Type into the chat:

Are boys names or girls names more likely to end in a vowel?

babynames

year <dbl>	sex <chr>	name <chr>	n <int>	prop <dbl>
1880	F	Mary	7065	7.238433e-02
1880	F	Anna	2604	2.667923e-02
1880	F	Emma		
1880	F	Elizabeth		
1880	F	Minnie		
1880	F	Margaret		
1880	F	Ida		
1880	F	Alice	1414	1.448711e-02
1880	F	Bertha	1320	1.352404e-02
1880	F	Sarah	1288	1.319618e-02

1–10 of 1,858,689 rows Previous 1 2 3 4 5 6 ... 100 Next

How can we build the proportion of boys and girls whose name ends in a vowel?

Most useful skills

1. How to extract/ replace substrings
2. How to find matches for patterns
3. Regular expressions

stringr

Simple, consistent functions for working with strings.

```
# install.packages("tidyverse")
library(stringr)
```

You installed this package indirectly when you installed the Tidyverse

```
install.packages("tidyverse")
```

does the equivalent of

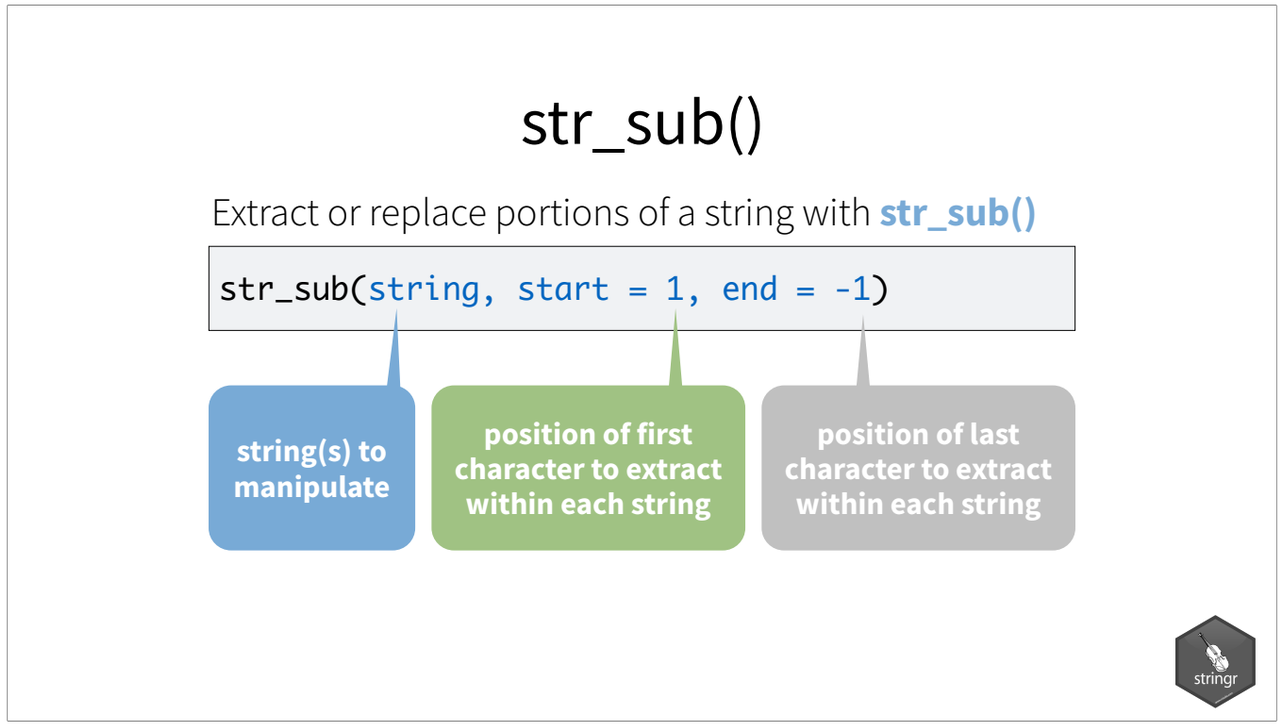
```
install.packages("ggplot2")
install.packages("dplyr")
install.packages("tidyr")
install.packages("readr")
install.packages("purrr")
install.packages("tibble")
install.packages("stringr")
install.packages("hms")
install.packages("lubridate")
install.packages("stringr")
install.packages("forcats")
install.packages("DBI")
install.packages("haven")
install.packages("httr")
install.packages("jsonlite")
install.packages("readxl")
install.packages("rvest")
install.packages("xml2")
install.packages("modelr")
install.packages("broom")
```

```
library("tidyverse")
```

does the equivalent of

```
library("ggplot2")
library("dplyr")
library("tidyr")
library("readr")
library("purrr")
library("tibble")
libraray("stringr")
```

You also load it automatically once you type "library(tidyverse)"



str_sub is the most popular function in the entire stringr package. Like all Tidyverse functions, the first parameter is the data you want to manipulate. But then you have 2 optional parameters, which specify the starting and stopping indices.

Let's now do a few short exercises to test your intuition of how str_sub works.

Quiz

What will this return?

```
str_sub("Garrett", 1, 2)
```

Quiz

What will this return?

```
str_sub("Garrett", 1, 2)
```

"Ga"

Remember: R is unusual in that indexes start at 1.

Quiz

What will this return?

```
str_sub("Garrett", 1, 1)
```

Quiz

What will this return?

```
str_sub("Garrett", 1, 1)
```

"G"

Quiz

What will this return?

```
str_sub("Garrett", 2)
```

This is a trick question. What happens when the 2nd position argument is missing?

Quiz

What will this return?

```
str_sub("Garrett", 2)
```

```
"arrett"
```

Quiz

What will this return?

```
str_sub("Garrett", -3)
```

Another trick question. What happens when the first position argument is negative?

Quiz

What will this return?

```
str_sub("Garrett", -3)
```

```
"ett"
```

Quiz

What will this return?

```
g <- "Garrett"
str_sub(g, -3) <- "eth"
g
```

Another trick: how does assignment work?

Quiz

What will this return?

```
g <- "Garrett"
str_sub(g, -3) <- "eth"
g
```

```
"Garreth"
```

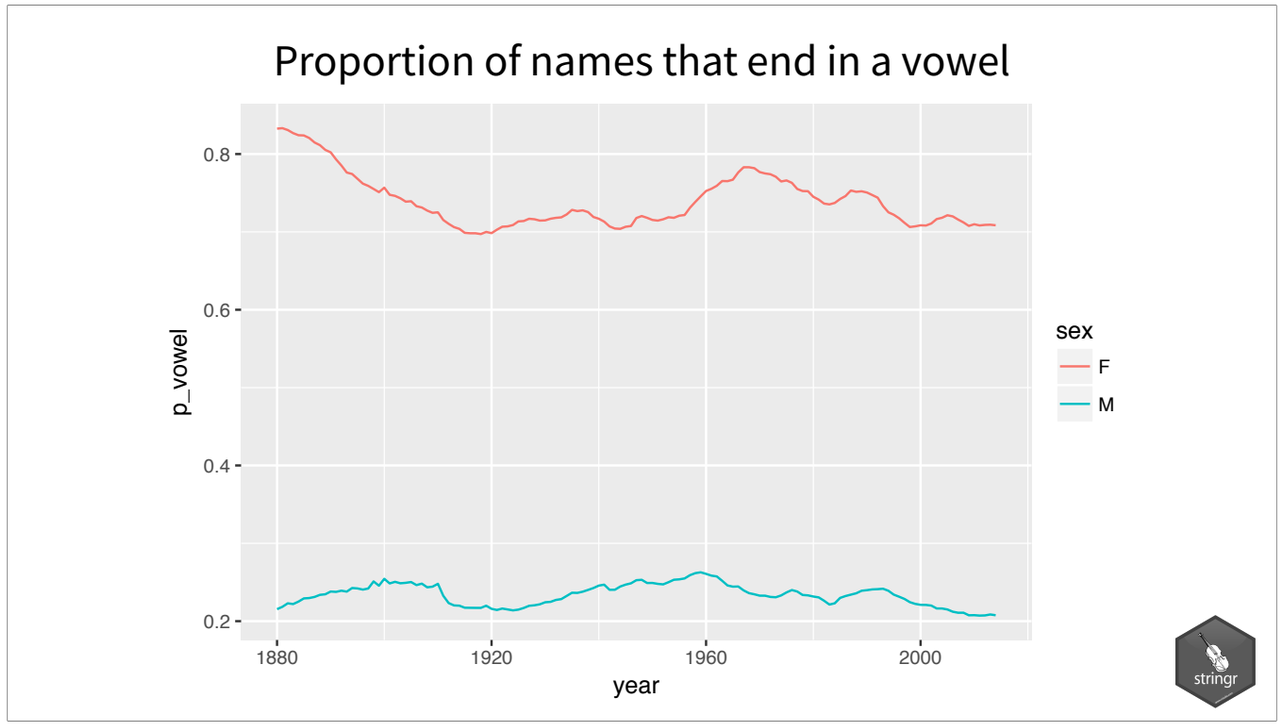
Your Turn 2

In your group, fill in the blanks to:

1. Isolate the last letter of every name
2. and create a logical variable that displays whether the last letter is one of "a", "e", "i", "o", "u", or "y".
3. Use a **weighted mean** to calculate the proportion of children whose name ends in a vowel (by year and sex).
4. and then display the results as a line plot.

```
babynames %>%
  mutate(last = str_sub(name, -1),
    vowel = last %in% c("a", "e", "i", "o", "u", "y")) %>%
  group_by(year, sex) %>%
  summarise(p_vowel = weighted.mean(vowel, n)) %>%
  ggplot() +
    geom_line(mapping = aes(year, p_vowel, color = sex))
```

"color = sex" is a new aesthetic. It means that ggplot should create multiple lines: one for each unique value in the column. Here there are values for sex (M and F), so they are each get their own line, and each line gets its own color


Proportion of names that end in a vowel
p_vowel
0.8
0.6
0.4
0.2
1880
1920
1960
2000
year
sex
F
M
stringr

```
help(package = stringr)
```

Simple, Consistent Wrappers for Common String Operations

Documentation for package ‘stringr’ version 1.2.0

- DESCRIPTION file.
- User guides, package vignettes and other documentation.

Help Pages

boundary	Control matching behaviour with modifier functions.
case	Convert case of a string.
coll	Control matching behaviour with modifier functions.
fixed	Control matching behaviour with modifier functions.
fruit	Sample character vectors for practicing string manipulations.
invert_match	Switch location of matches to location of non-matches.
modifiers	Control matching behaviour with modifier functions.
regex	Control matching behaviour with modifier functions.
sentences	Sample character vectors for practicing string manipulations.

The stringr package contains a ton of functions. We won't get a chance to cover them all. Instead, I'd like to make sure that you're able to use the help for the package effectively. Bring up the help now, either by clicking on the package name in the RStudio window, or by typing "help(package=stringr)"

Your Turn

Run **help(package = stringr)** to open the help directory for stringr.

Read through the function descriptions and find the function that determines whether two strings match.

The first person in your group to find it wins!

Garret says to type "help(package=stringr)". I recommend just finding stringr in the package pane, and clicking on it.

str_detect()

Test whether a pattern appears within a string.

```
str_detect(string, pattern)
```

vector of strings to find patterns in

a string that represents a regular expression

This is the function that I hoped you would find. This function might work slightly differently than you'd expect, so let's go through a few examples.

Quiz

What will this return?

```
strings <- c("Apple", "Orange")
str_detect(strings, "pp")
```

Quiz

What will this return?

```
strings <- c("Apple", "Orange")
str_detect(strings, "pp")
```

```
TRUE FALSE
```

Quiz

What will this return?

```
strings <- c("Apple", "Pineapple")
str_detect(strings, "apple")
```

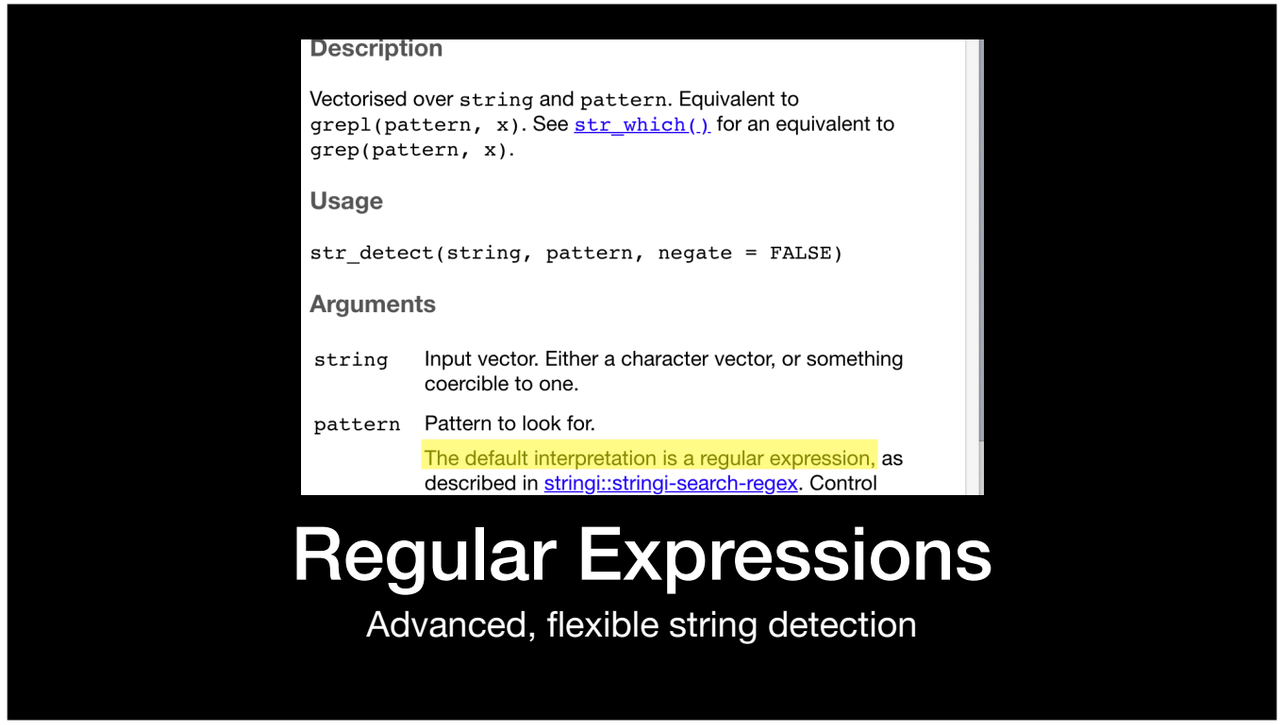
Quiz

What will this return?

```
strings <- c("Apple", "Pineapple")
str_detect(strings, "apple")
```

```
FALSE  TRUE
```

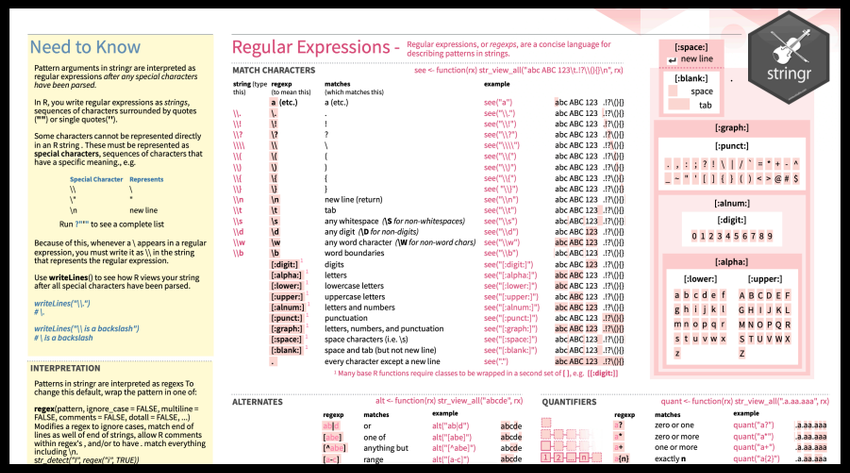
Case sensitive!



Regular Expressions

Advanced, flexible string detection

When it comes to working with strings, something that you at least need to be aware of is Regular Expressions. Type into the chat window whether or not you've heard of Regular Expressions before. You can see here that the pattern defaults to being interpreted as a regular expression



Regular Expressions

Advanced, flexible string detection

Page 2 of Cheat Sheet

Regular Expressions are a mini language that is designed to detect patterns in text. As you can imagine, using language to discuss language is complicated. You can google "Regular Expressions Cheat Sheet" and see tons of people have created tons of resources to help you learn them.

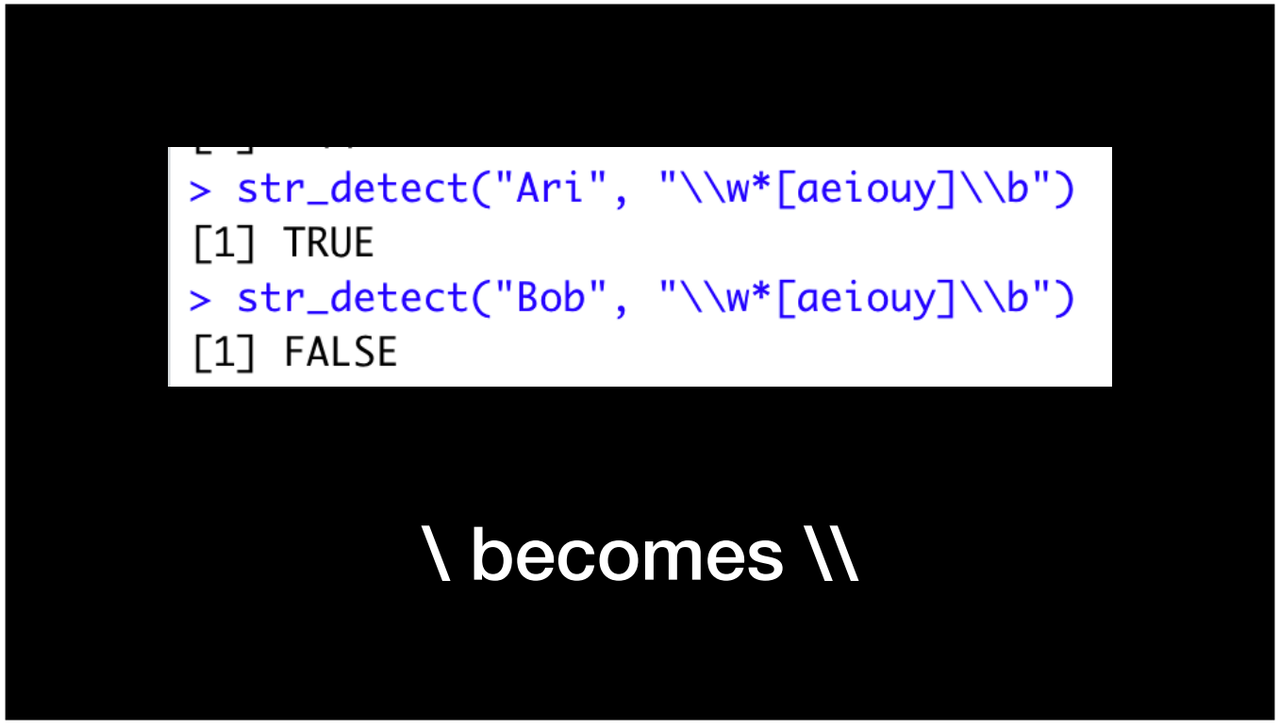
The most famous joke about regular expressions ever. From the comic XKCD.

\w*[aeiouy]\b

Symbol	Meaning
\w	"word character"
*	"any number of"
[aeiouy]	"any character in between the []"
\b	"word boundary"

This is an example of a regular expression. It might be a bit scary - most regular expressions are.

QUIZ: What does this RegEx do?



If you give a regular expression to str_detect, all the back slashes need to be escaped. That is, each \ becomes \\. As you can see here, this Regular Expression "works". "Ari" ends in a vowel but "Bob" does not.

Your Turn

How many names end in a vowel anyways?

1. Winnow babynames to distinct names with:

   babynames %>% distinct(name)

2. Use str_detect() to detect names that end in a vowel (pattern = "\\w*[aeiouy]\\b")

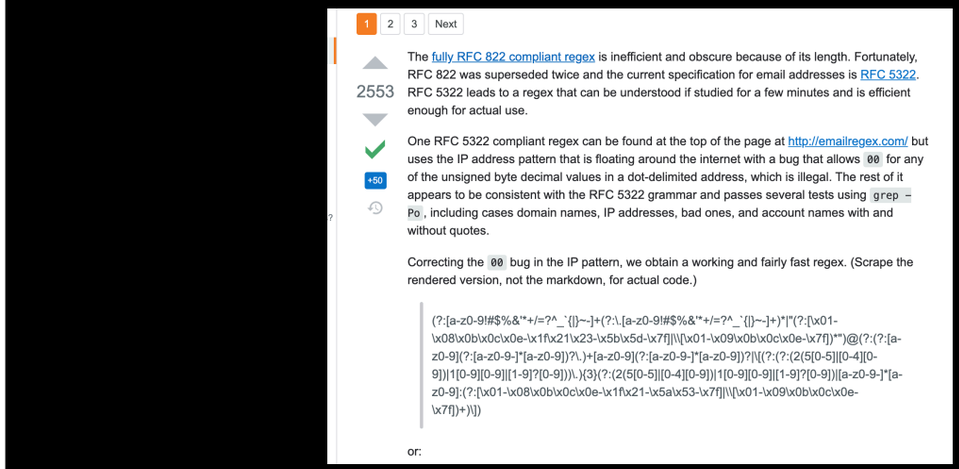
3. Calculate the number of names that end in a vowel

```
babynames %>%
  distinct(name) %>%
  mutate(vowel = str_detect(name, "\\w*[aeiouy]\\b")) %>%
  summarise(n = n(), n_vowel = sum(vowel))
# A tibble: 1 × 2
      n n_vowel
  <int>   <int>
1 97310   56945
```

hello@example.com

Create a RegEx for an email address...
How hard can it be?

Take a minute to try and solve this.



1 2 3 Next

2553

The fully RFC 822 compliant regex is inefficient and obscure because of its length. Fortunately, RFC 822 was superseded twice and the current specification for email addresses is RFC 5322. RFC 5322 leads to a regex that can be understood if studied for a few minutes and is efficient enough for actual use.

+50

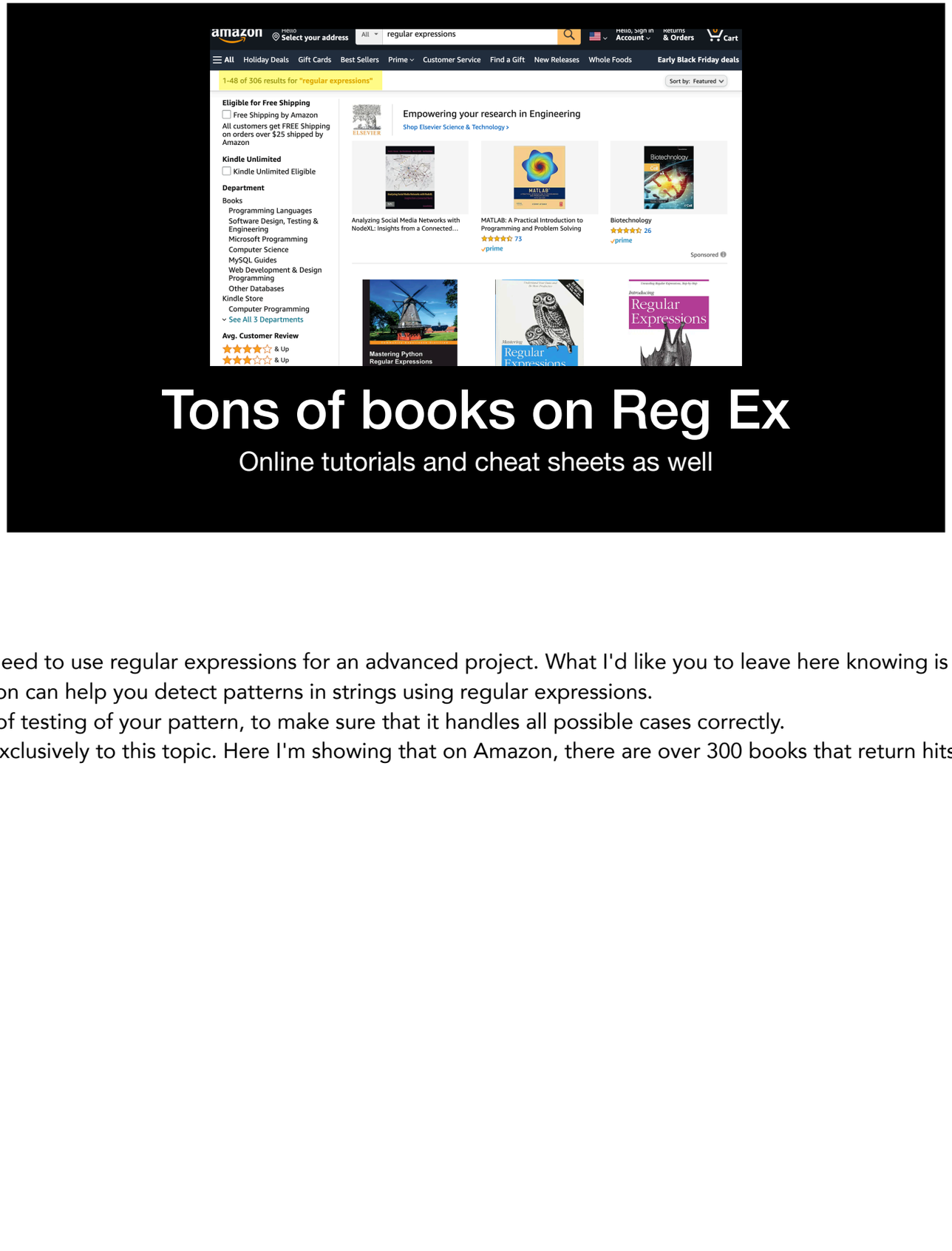
One RFC 5322 compliant regex can be found at the top of the page at http://emailregex.com/ but uses the IP address pattern that is floating around the internet with a bug that allows `00` for any of the unsigned byte decimal values in a dot-delimited address, which is illegal. The rest of it appears to be consistent with the RFC 5322 grammar and passes several tests using `grep -Po`, including cases domain names, IP addresses, bad ones, and account names with and without quotes.

Correcting the `00` bug in the IP pattern, we obtain a working and fairly fast regex. (Scrape the rendered version, not the markdown, for actual code.)

> (?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])

or:

Not Human Readable



Tons of books on Reg Ex

Online tutorials and cheat sheets as well

There might very well be a time when you need to use regular expressions for an advanced project. What I'd like you to leave here knowing is a few things.

1. The stringr package's str_detect function can help you detect patterns in strings using regular expressions.
2. You should expect to need to do a lot of testing of your pattern, to make sure that it handles all possible cases correctly.
3. There are a ton of resources devoted exclusively to this topic. Here I'm showing that on Amazon, there are over 300 books that return hits for Regular Expressions

Non Numeric Data in R

- Logical / Boolean
- Character / String
- Factor / Categorical
- Dates and Times

Now lets talk about factors (categorical data), and how the Tidyverse can make it easier for you to work with them.

factors

R's representation of categorical data. Consists of:

1. A set of **values**
2. An ordered set of **valid levels**

```
eyes <- factor(x = c("blue", "green", "green"),
               levels = c("blue", "brown", "green"))
eyes
## [1] blue green green
## Levels: blue brown green
```

Even though "brown" is not in the data, it's still a possible value

factors

Stored as an integer vector with a levels attribute

```
unclass(eyes)
## 1 3 3
## attr(,"levels")
## "blue"  "brown" "green"
```

Factors are stored as integers. And you can see here that "blue green green" is really, internally, "1 3 3".

This impacts how factors behave when you sort them.

Quiz

Sorting Character Data

What will this return?

```
possible_values = c("low", "normal", "high")
sort(possible_values)
```

Imagine recording (and sorting) medical data in character form. Say blood pressure levels, which can be either low, normal or high.

Quiz

Sorting Character Data

What will this return?

```
possible_values = c("low", "normal", "high")
sort(possible_values)
```

```
[1] "high"  "low"   "normal"
```

Character data is sorted in alphabetical order (low to high). Sometimes this is what you want. And sometimes (like in this case) it isn't.

Quiz

Sorting Factor Data

What will this return?

```
possible_values = c("low", "normal", "high")
recorded_values = c("low", "low", "normal", "high")
f = factor(x = recorded_values,
           levels = possible_values)

sort(f)
```

How will the data sort when the data is a factor?

Quiz

Sorting Factor Data

```
possible_values = c("low", "normal", "high")
recorded_values = c("low", "low", "normal", "high")
f = factor(recorded_values,
           levels = possible_values)

sort(f)

[1] low low normal high
Levels: low normal high
```

factor data is sorted by its levels

Quiz

Should "day" be a factor or character?

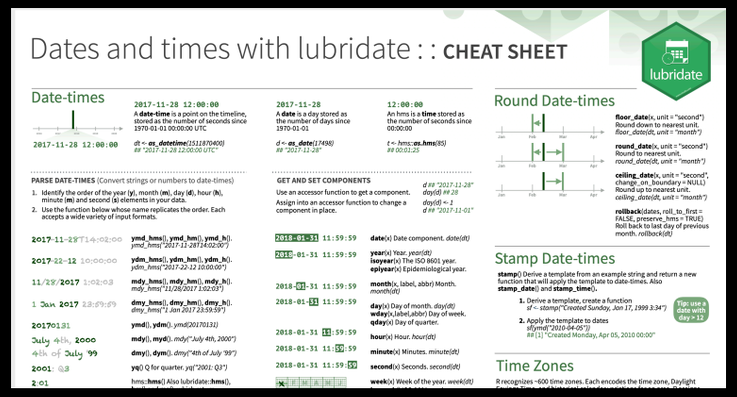
	time_hour	name	air_time	distance	day	delayed
1	2013-01-01 05:00:00	United Air Lines Inc.	13620s (~3.78 hours)	1400	Tuesday	TRUE
2	2013-01-01 05:00:00	United Air Lines Inc.	13620s (~3.78 hours)	1416	Tuesday	TRUE
3	2013-01-01 05:00:00	American Airlines Inc.	9600s (~2.67 hours)	1089	Tuesday	TRUE
4	2013-01-01 05:00:00	JetBlue Airways	10980s (~3.05 hours)	1576	Tuesday	FALSE
5	2013-01-01 06:00:00	Delta Air Lines Inc.	6960s (~1.93 hours)	762	Tuesday	FALSE
6	2013-01-01 05:00:00	United Air Lines Inc.	9000s (~2.5 hours)	719	Tuesday	TRUE
7	2013-01-01 06:00:00	JetBlue Airways	9480s (~2.63 hours)	1065	Tuesday	TRUE
8	2013-01-01 06:00:00	ExpressJet Airlines Inc.	3180s (~53 minutes)	229	Tuesday	FALSE
9	2013-01-01 06:00:00	JetBlue Airways	8400s (~2.33 hours)	944	Tuesday	FALSE
10	2013-01-01 06:00:00	American Airlines Inc.	8280s (~2.3 hours)	733	Tuesday	TRUE
11	2013-01-01 06:00:00	JetBlue Airways	8940s (~2.48 hours)	1028	Tuesday	FALSE

Remember this slide? Should "day" be a factor or a character? What are the pros and cons of this decision?

Non Numeric Data in R

- Logical / Boolean
- Character / String
- Factor / Categorical
- Dates and Times

We're now coming to one of the most complex data types known to mankind: Dates and Times. If you don't believe me that they are unbelievably complex, you will soon.



lubridate Cheat Sheet

Please take out

Quiz

Does every year have 365 days?

The complexity with date-times is that you have to cover every possible case ahead of time. Consider these examples that show you how many edge cases there are.
Leap Years

Quiz

Does every day have 24 hours?

Daylight Savings Time. Imagine writing a billing system where someone bills by the hour ... and then the clock either skips ahead or moves back and hour!

Quiz

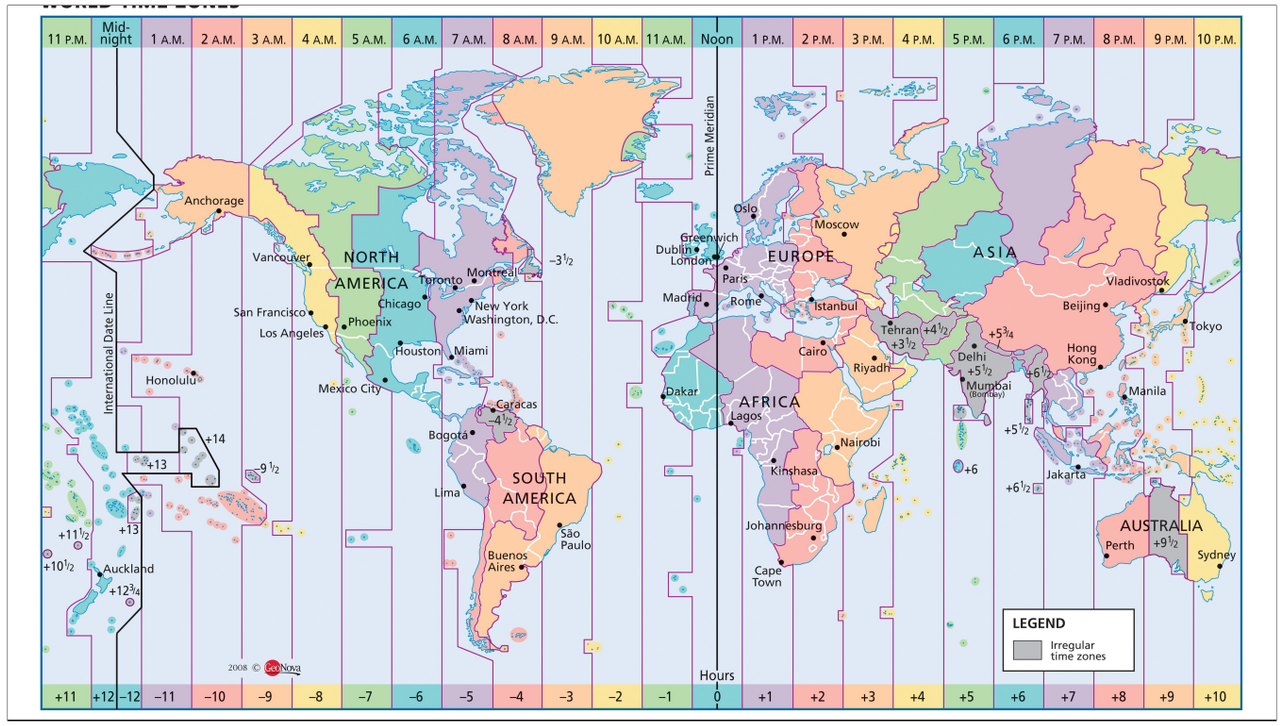
Does every minute have 60 seconds?

Google "Leap Second"

Quiz

What does a month measure?

30 days has September. How can you measure something in "months" when a "month" isn't consistent?



This is how Time Zones are laid out. It shows just how big a difference there is between the actual math and rules of dates and times and how we experience it on a day-to-day basis. Note how the international date line moves to accomodate the border between Alaska and Russia.

Dates and times involve:

1. **instants** - specific moments in time (e.g., Jan 1, 2017)
2. **time spans** - spans of time (e.g., a day)

For instants, think of timestamps in a database - *This email went out at this time.*
For time spans, think of a service that bills by the hour - *The lawyer spent 2 hours in court.*

Most useful skills

1. Creating dates/times (i.e. *parsing*)
2. Access and change parts of a date

I actually recently helped someone with these exact problems during an Office Hours session. The had a bunch of dates, and wanted to strip the day components out of all of them, so you just had the equivalent of "January 2020" instead of "January 1, 2020", "January 2, 2020", etc. You can't actually remove the day portion of a date in R, but we were able to shift all the days to be "1", which accomplished the same thing.

Warm Up

Decide in your group:

- What is the best day of the week to fly?

We're going to be going back to the nycflights database for this problem. Take a minute to think about the answer to this question.

Creating dates and times

Before diving into the tidyverse packages for dealing with dates and times, let's look at the classes that Base R provides.

(some of) R's instant classes

```
2017-01-01 12:34:56
```

Let's start by reviewing two classes R has for working with *points in time ("instants").*

POSIXct

`2017-01-01 12:34:56`

Stored as the number of seconds since **1970-01-01 00:00:00**

```
Sys.time()
## "2017-01-01 12:34:56 EST"

unclass(Sys.time())
## 1483292096
```

Unix Standard Epoch

Dates

2017-01-01 12:34:56

Stored as the number of days since 1970-01-01

```
Sys.Date()
## "2017-01-01"

unclass(Sys.Date())
## 17167
```

lubridate

Functions for working with dates and time spans

```
# install.packages("tidyverse")
library(lubridate)
```

This package used to not be loaded automatically when the tidyverse loads. But now I believe that it is.

ymd() family

To parse strings as dates, use a y, m, d, h, m, s combo

```
ymd("2017/01/11")
mdy("January 11, 2017")
ymd_hms("2017-01-11  01:30:55")
```

y = year, m = month, d = day
Note that the second one is mdy, not ydm.
The third has hms

Parsing functions

function	parses to
ymd_hms(), ymd_hm(), ymd_h() ydm_hms(), ydm_hm(), ydm_h() dmy_hms(), dmy_hm(), dmy_h() mdy_hms(), mdy_hm(), mdy_h()	POSIXct
ymd(), ydm(), mdy() myd(), dmy(), dym(), yq()	Date (POSIXct if tz specified)
hms(), hm(), ms()	Period

All possible variations!

ymd() family

To parse strings as dates, use a y, m, d, h, m, s combo

```
ymd_hms("2017-01-01T12:34:56")
## "2017-01-01 12:34:56 UTC"
ymd_hm("2017/01/01 12:34")
## "2017-01-01 12:34:56 UTC"
dmy("2 January 17")
## "2017-01-02"
ms("34 56")
## "34M 56S"
```

lubridate functions are separator agnostic

Accessing and changing components

That's everything I have to say about parsing dates. Now let's look at accessing and changing components of a datetime object

Accessing components

Extract components by name with a **singular** name

```
date <- ymd("2017-01-11")
year(date)
## 2017
```

Setting components

Use the same function to set components

```
date
## "2017-01-11"
year(date) <- 1999
date
## "1999-01-11"
```

This is something that's very unique in R.

Accessing date time components

function	extracts	extra arguments
year()	year	
month()	month	label = FALSE, abbr = TRUE
week()	week	
day()	day of month	
wday()	day of week	label = FALSE, abbr = TRUE
qday()	day of quarter	
yday()	day of year	
hour()	hour	
minute()	minute	
second()	second	

All functions for getting/setting components of a datetime

Accessing components

```
wday(ymd("2017-01-11"))
## 4
wday(ymd("2017-01-11"), label = TRUE)
## [1] Wed
## 7 Levels: Sun < Mon < Tues < Wed < Thurs < ... < Sat
wday(ymd("2017-01-11"), label = TRUE, abbr = FALSE)
## [1] Wednesday
## 7 Levels: Sunday < Monday < Tuesday < ... < Saturday
```

Your Turn 6

Fill in the blanks to:

Extract the day of the week of each flight (as a full name) from **time_hour**.

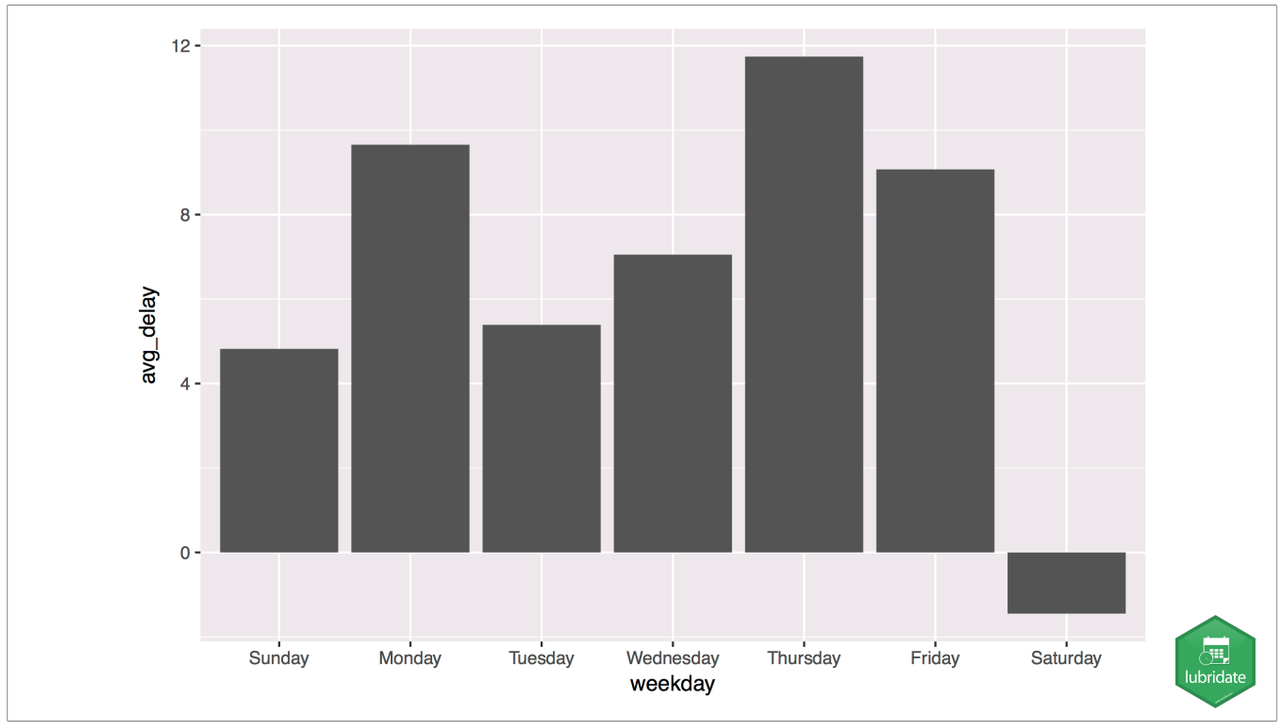
Calculate the average **arr_delay** by day of the week.

Plot the results as a column chart (bar chart) with **geom_col()**.

Use geom_col() (not geom_bar()).

-geom_col assumes you are telling it the height of the bar
-geom_bar assumes you are giving it a lot of rows, and asking it to do the counting for you

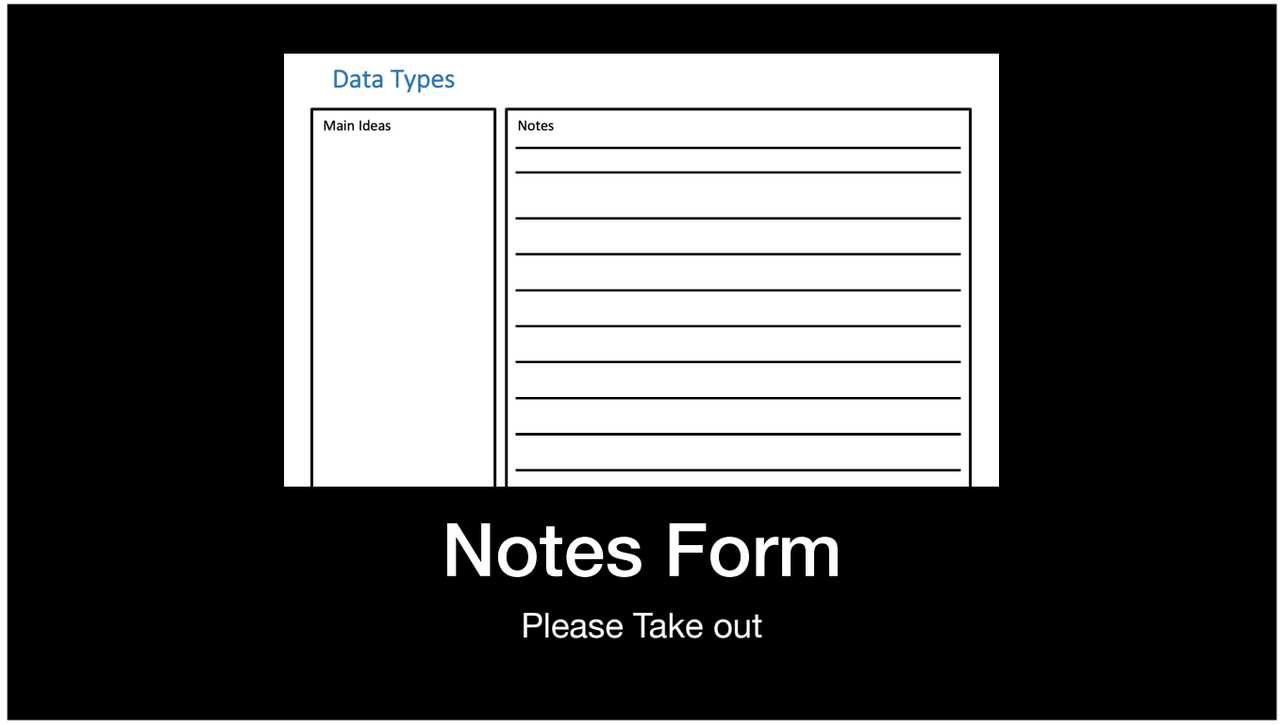
```
flights %>%
  mutate(weekday = wday(time_hour, label = TRUE, abbr = FALSE)) %>%
  group_by(weekday) %>%
  drop_na(arr_delay) %>%
  summarise(avg_delay = mean(arr_delay)) %>%
  ggplot() +
    geom_col(mapping = aes(x = weekday, y = avg_delay))
```


12
8
4
0
avg_delay
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
weekday
lubridate

Non Numeric Data in R

- Logical / Boolean
- Character / String (use the stringr package)
- Factor / Categorical
- Dates and Times (use the lubridate package)

That concludes everything I wanted to say about how working with non-numeric data. As a reminder, logical/boolean data comes up all the time, and you can use base R for that. I expect that you will regularly have to work with string (character) and date data. And stringr and lubridate really do make those tasks easier. I expect that manipulating factors is something you'll need to do rarely.



I'd now like to give you a minute to write down any thoughts you have about what you just learned, including any questions or things you'd like to follow up on