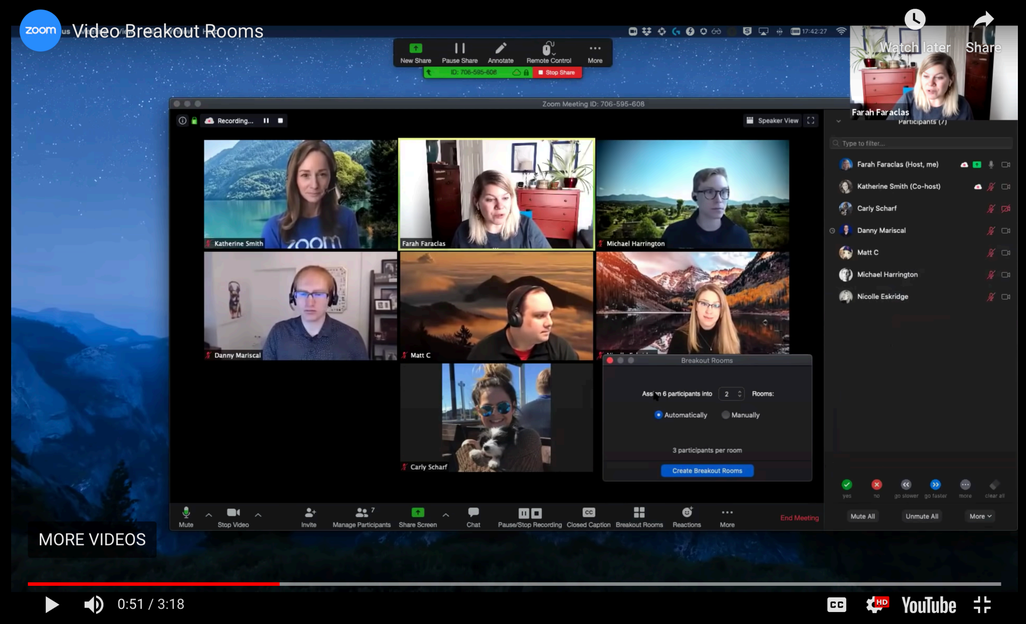
Data Types in the Tidyverse

Exercise: What do you already know?

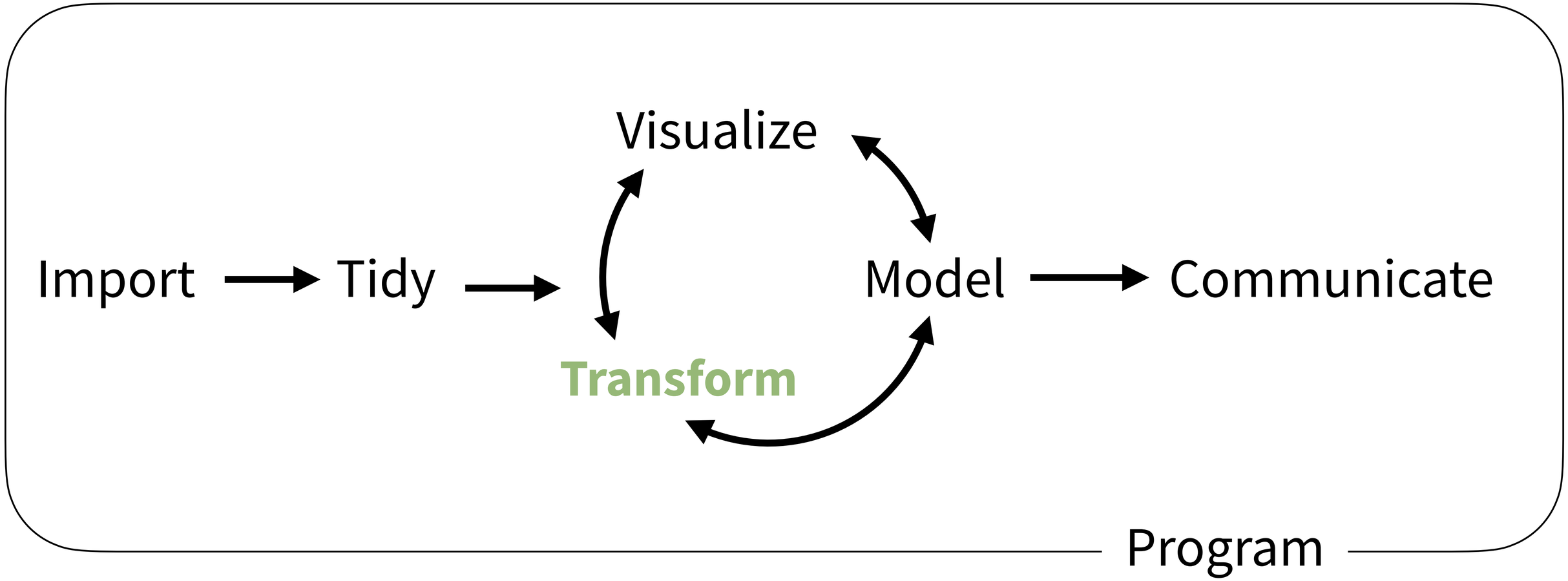
- In the Chat Window type:

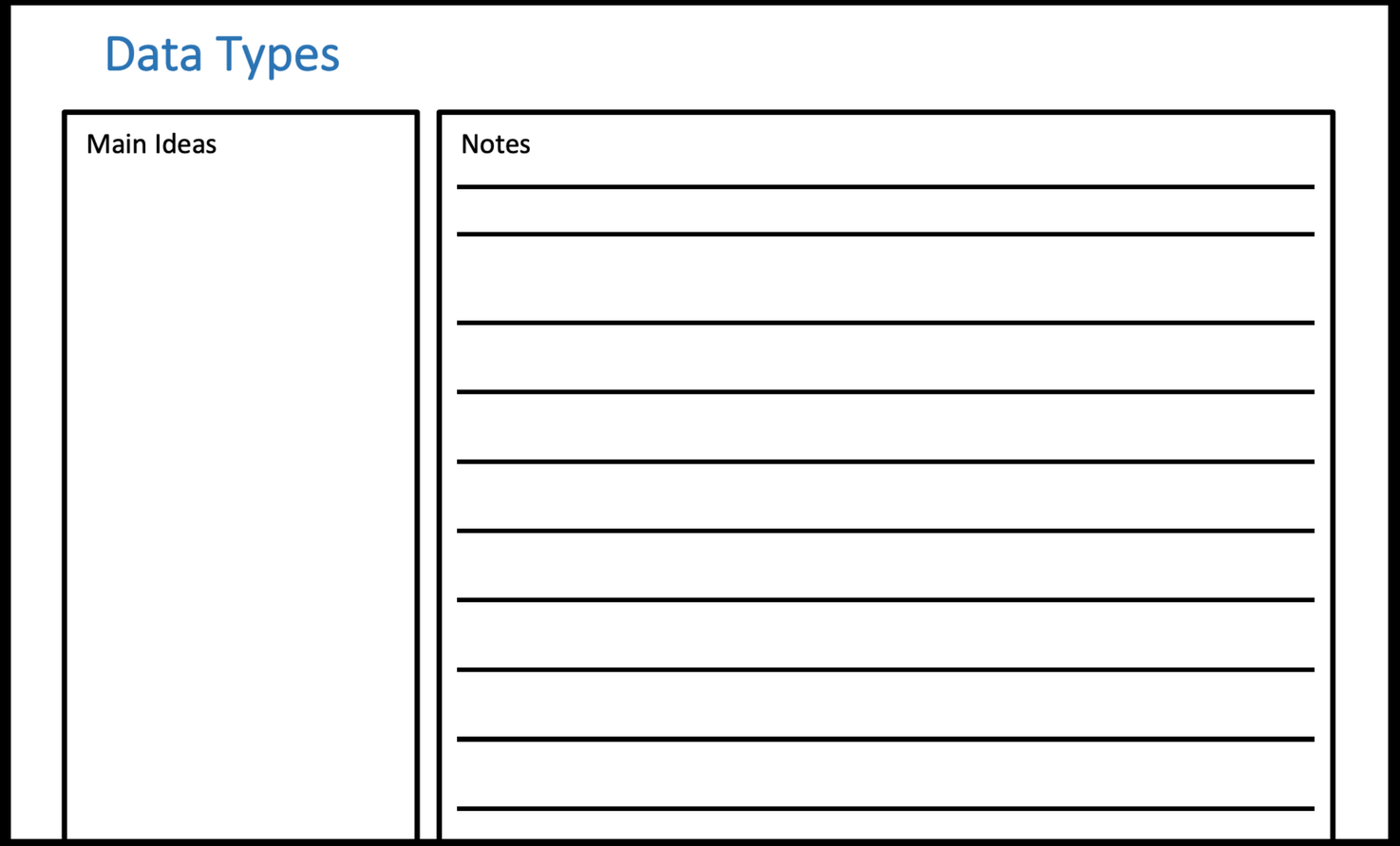
  1. What types of data do you need to work with / manipulate?

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



(Applied) **Data Science**





Notes Form

Please Take out

Data types with

Open **Data-Types.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

Logicals

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"
```

```
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

```
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?

Most useful skills

1. Math with logicals

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
```

Your Turn 1

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

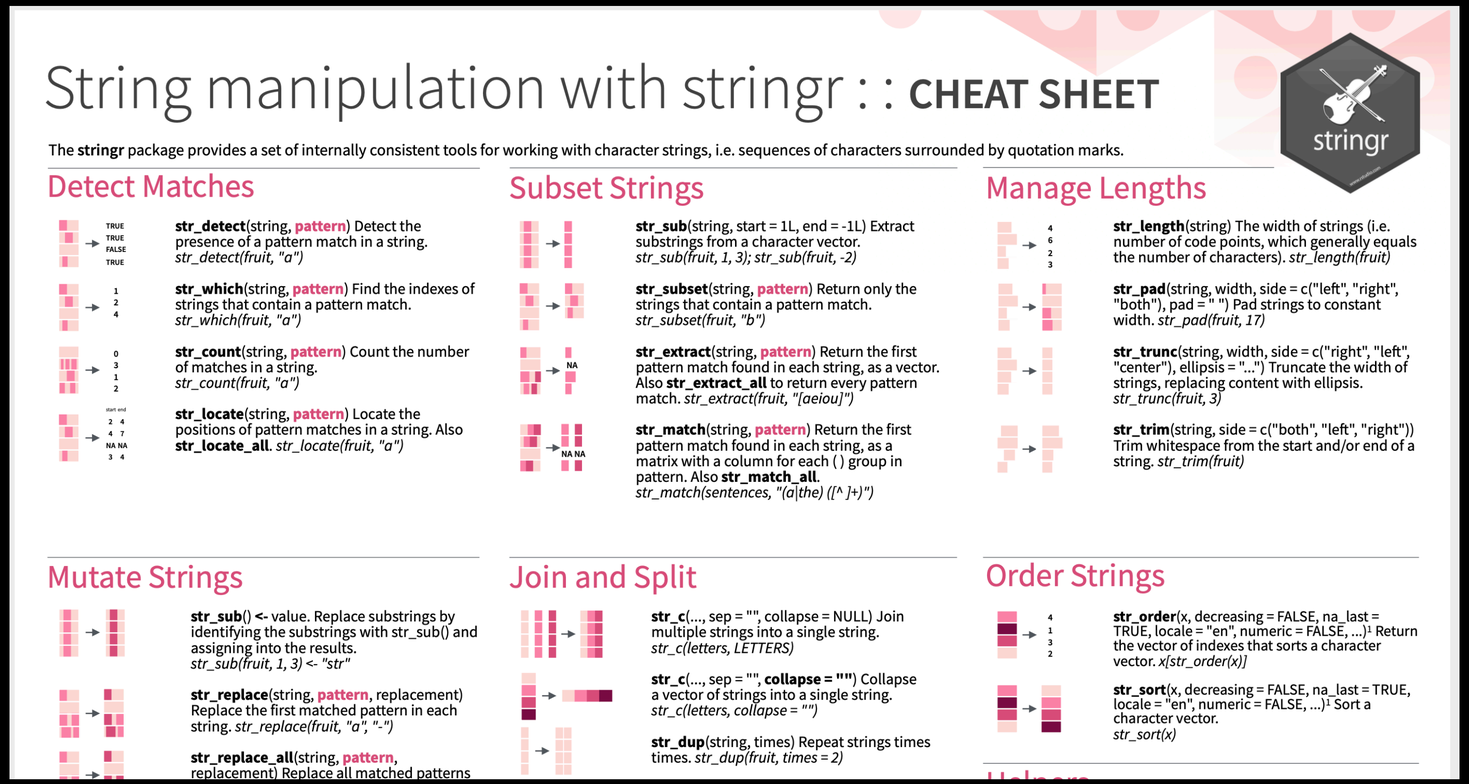
Then, remove all rows that contain an NA in **delayed**.

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
```

Strings



String manipulation with stringr : : CHEAT SHEET

The **stringr** 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 the presence of a pattern match in a string.
str_detect(fruit, "a")

str_which(string, **pattern**) Find the indexes of strings that contain a pattern match.
str_which(fruit, "a")

str_count(string, **pattern**) Count the number of matches in a string.
str_count(fruit, "a")

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. *str_locate(fruit, "a")*

Subset Strings

str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector.
str_sub(fruit, 1, 3); str_sub(fruit, -2)

str_subset(string, **pattern**) Return only the strings that contain a pattern match.
str_subset(fruit, "b")

str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*

str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all**.
str_match(sentences, "(a|the) ([^]+)")

Manage Lengths

str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). *str_length(fruit)*

str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis.
str_trunc(fruit, 3)

str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. *str_trim(fruit)*

Mutate Strings

str_sub() **<-** value. Replace substrings by identifying the substrings with str_sub() and assigning into the results.
str_sub(fruit, 1, 3) <- "str"

str_replace(string, **pattern**, replacement) Replace the first matched pattern in each string. *str_replace(fruit, "a", "-")*

str_replace_all(string, **pattern**, replacement) Replace all matched patterns

Join and Split

str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string.
str_c(letters, LETTERS)

str_c(..., sep = "", **collapse = ""**) Collapse a vector of strings into a single string.
str_c(letters, collapse = "")

str_dup(string, times) Repeat strings times times. *str_dup(fruit, times = 2)*

Order Strings

str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Return the vector of indexes that sorts a character vector. *x[str_order(x)]*

str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Sort a character vector.
str_sort(x)

stringr Cheat Sheet

Please Take out

(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)
```

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

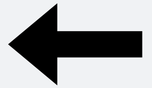
does the equivalent of

```
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")
```



str_sub()

Extract or replace portions of a string with **str_sub()**

```
str_sub(string, start = 1, end = -1)
```

string(s) to manipulate

position of first character to extract within each string

position of last character to extract within each string

Quiz

What will this return?

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

Quiz

What will this return?

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

"Ga"

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)
```

Quiz

What will this return?

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

```
"arrett"
```

Quiz

What will this return?

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

Quiz

What will this return?

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

```
"ett"
```

Quiz

What will this return?

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

Quiz

What will this return?

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

```
"Garreth"
```

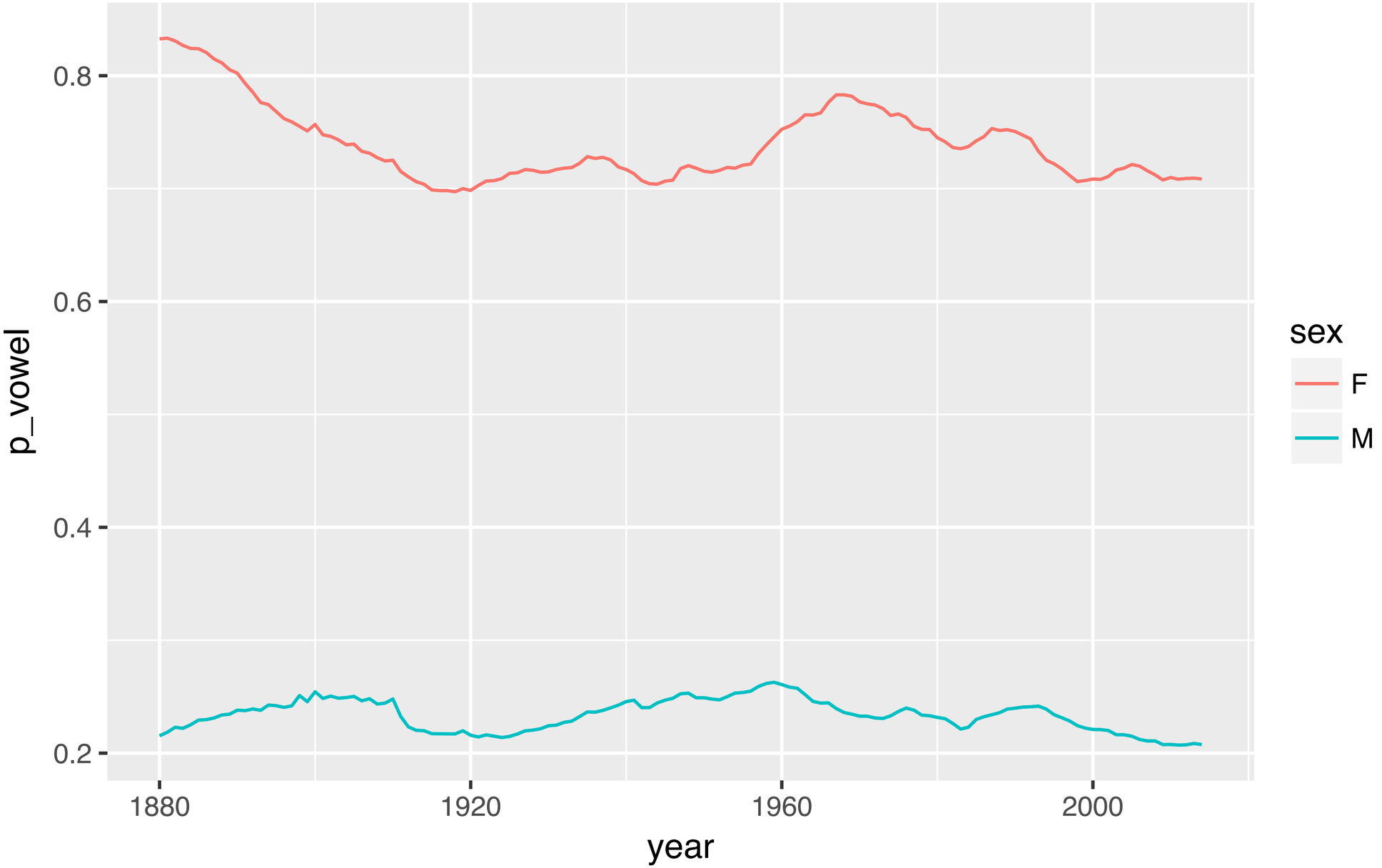
Your Turn 2

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))
```

Proportion of names that end in a vowel



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

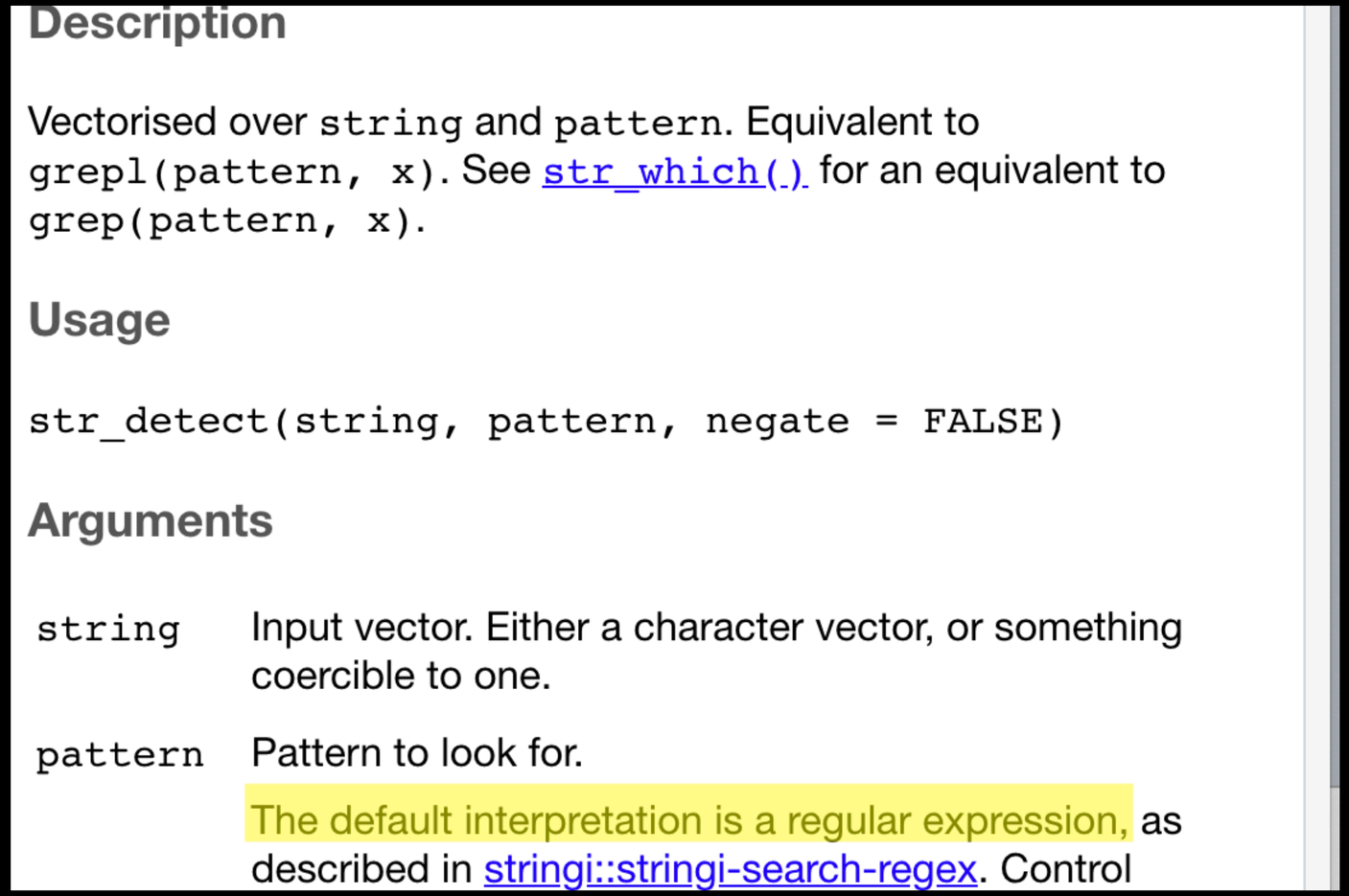
Simple, Consistent Wrappers for Common String Operations

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.



Description

Vectorised over `string` and `pattern`. Equivalent to `grepl(pattern, x)`. See `str_which()` for an equivalent to `grep(pattern, x)`.

Usage

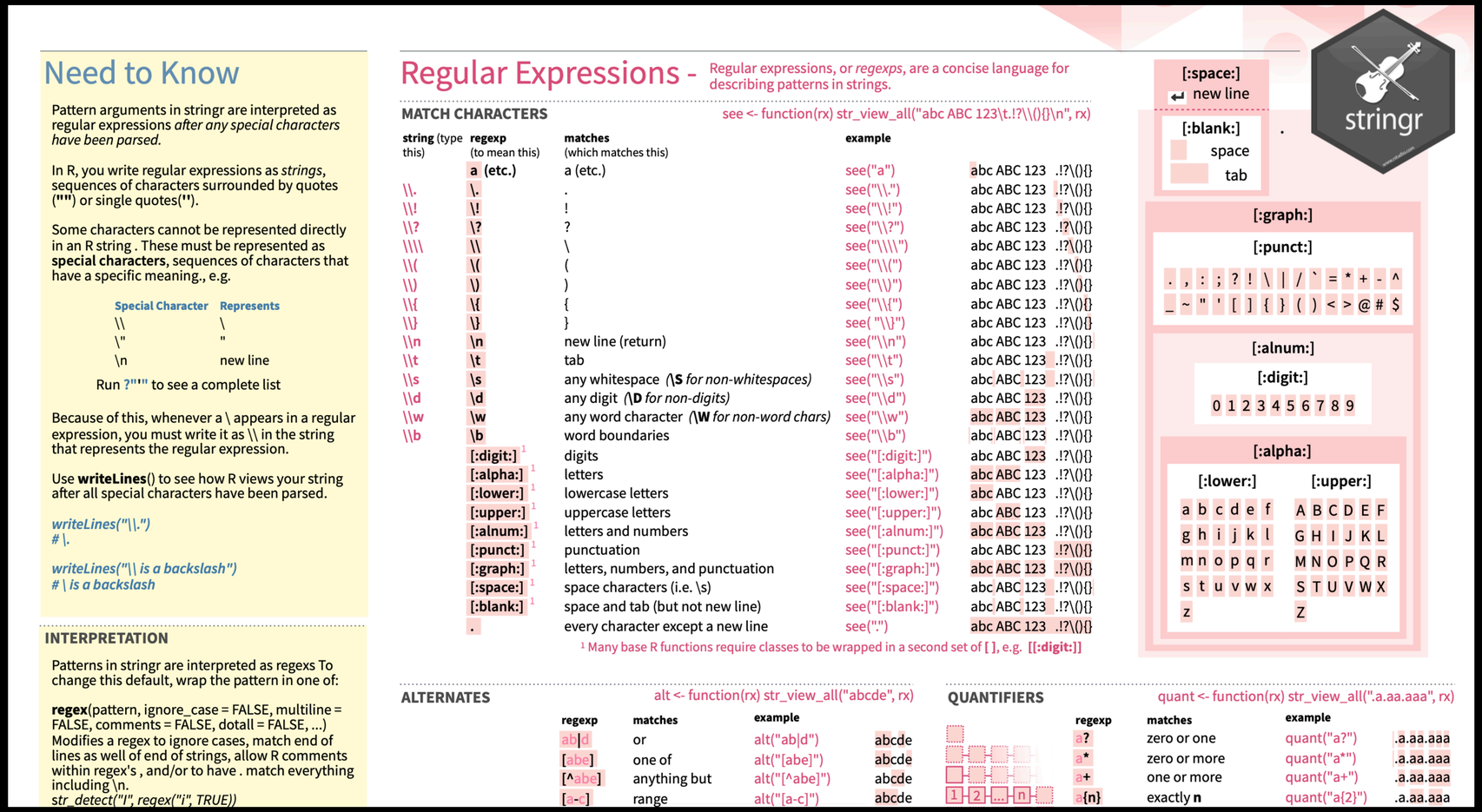
```
str_detect(string, pattern, negate = FALSE)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringi::stringi-search-regex. Control

Regular Expressions

Advanced, flexible string detection



Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes (**""**) or single quotes(**''**).

Some characters cannot be represented directly in an R string . These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\	\
\"	"
\n	new line

Run **?"'"** to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use **writeLines**() to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.

writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...)
Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
str_detect("I", regex("i", TRUE))

Regular Expressions - Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	\.	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	\!	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	\?	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	\\	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	\(	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	\)	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	\{	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	\}	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	\n	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	\t	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	\s	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	\d	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	\w	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	\b	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:] [1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:] [1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:] [1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:] [1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:] [1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:] [1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:] [1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:] [1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of **[ ]**, e.g. **[[:digit:]]**

[:space:] new line
[:blank:] space tab
.

[:graph:]
[:punct:]
. , : ; ? ! \ | / ` = * + - ^ _ ~ " ' [ ] { } ( ) < > @ # $

[:alnum:]
[:digit:]
0 1 2 3 4 5 6 7 8 9

[:alpha:]
[:lower:]
a b c d e f g h i j k l m n o p q r s t u v w x z
[:upper:]
A B C D E F G H I J K L M N O P Q R S T U V W X Z

ALTERNATES

alt <- function(rx) str_view_all("abcde", rx)

regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

QUANTIFIERS

quant <- function(rx) str_view_all(".a.aa.aaa", rx)

regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa

Regular Expressions

Advanced, flexible string detection

Page 2 of Cheat Sheet


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

\w*[aeiouy]\b

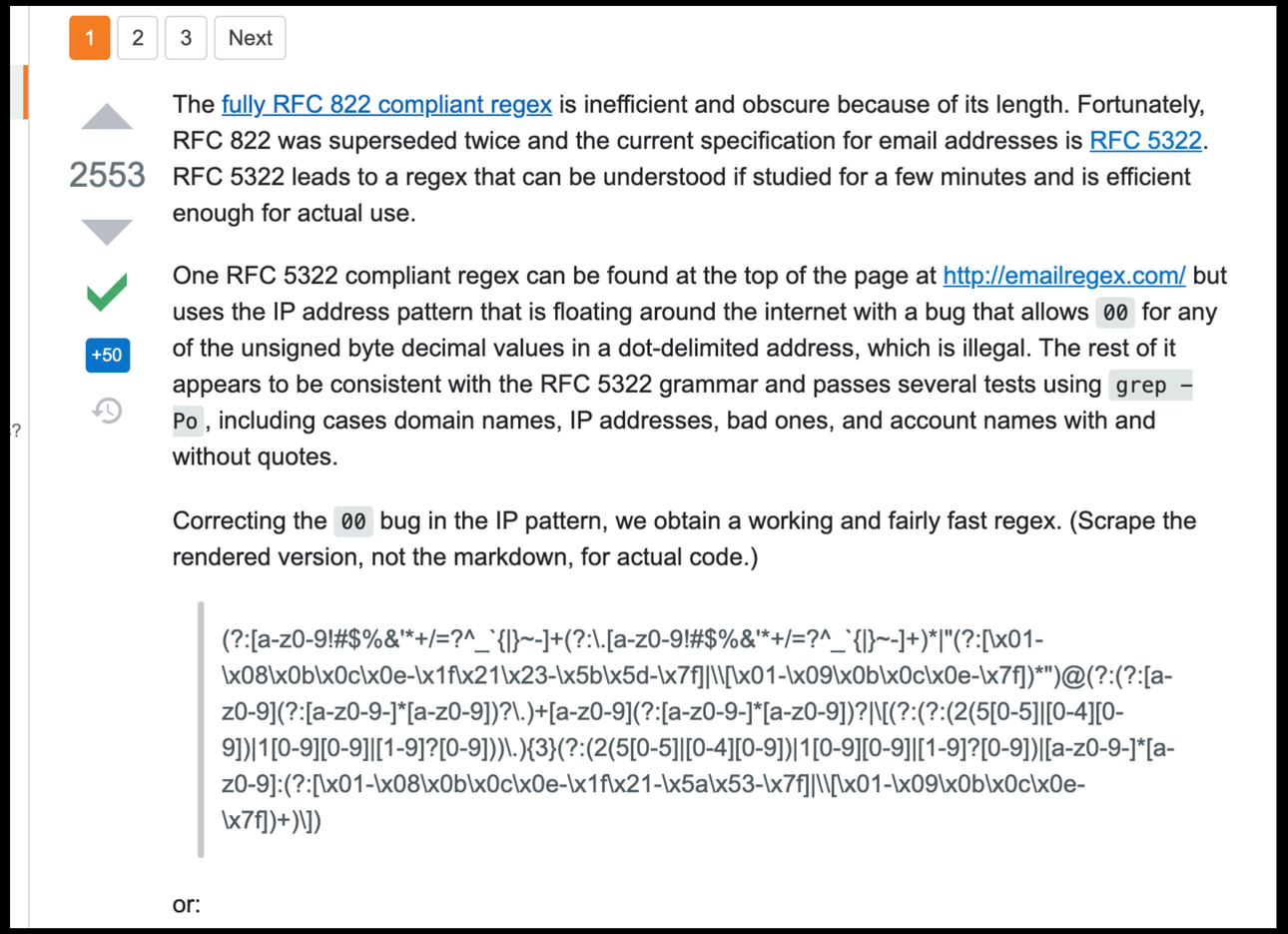
Symbol	Meaning
\w	"word character"
*	"any number of"
[aeiouy]	"any character in between the []"
\b	"word boundary"

```
> str_detect("Ari", "\\w*[aeiouy]\\b")
[1] TRUE
> str_detect("Bob", "\\w*[aeiouy]\\b")
[1] FALSE
```

\ becomes \\

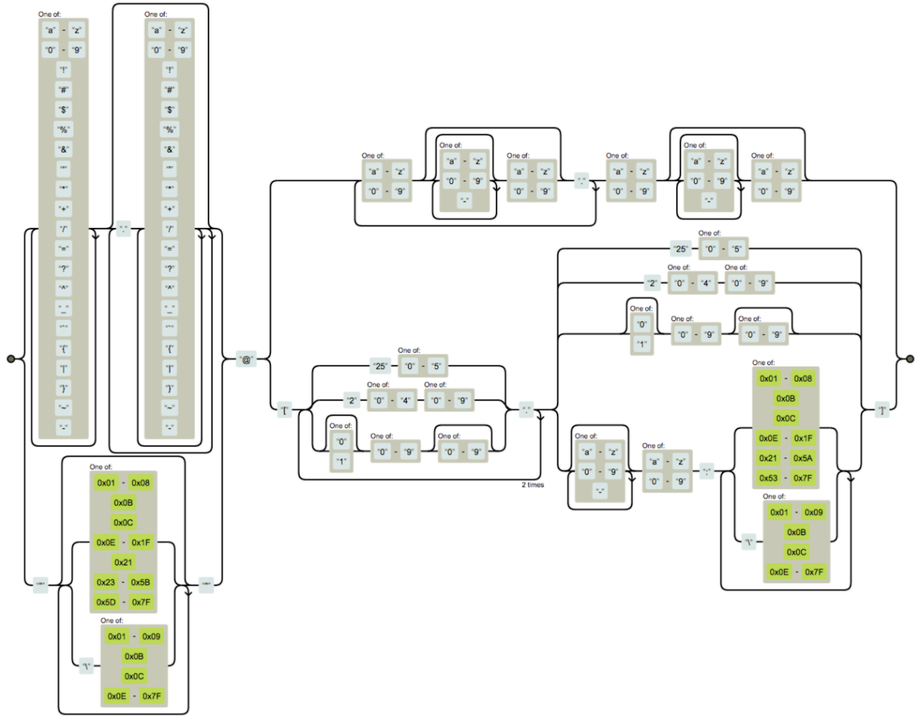
hello@example.com

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



Not Human Readable

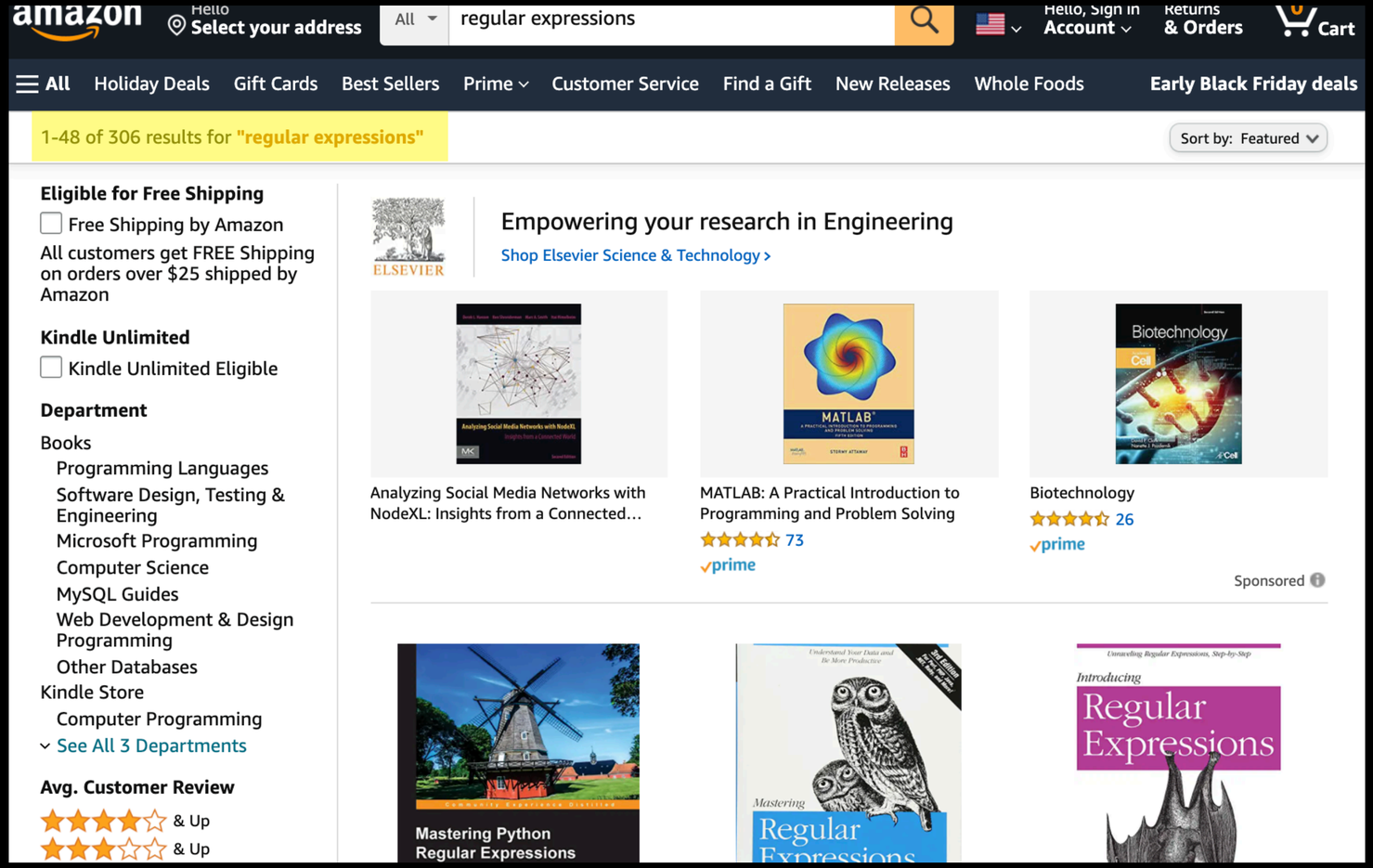
Here is diagram of finite state machine for above regexp which is more clear than regexp itself



The more sophisticated patterns in Perl and PCRE (regex library used e.g. in PHP) can correctly parse RFC 5322 without a hitch. Python and C# can do that too, but they use a different syntax from those first two. However, if you are forced to use one of the many less powerful pattern-matching languages, then it's best to use a real parser.

"RegEx Visualizer"

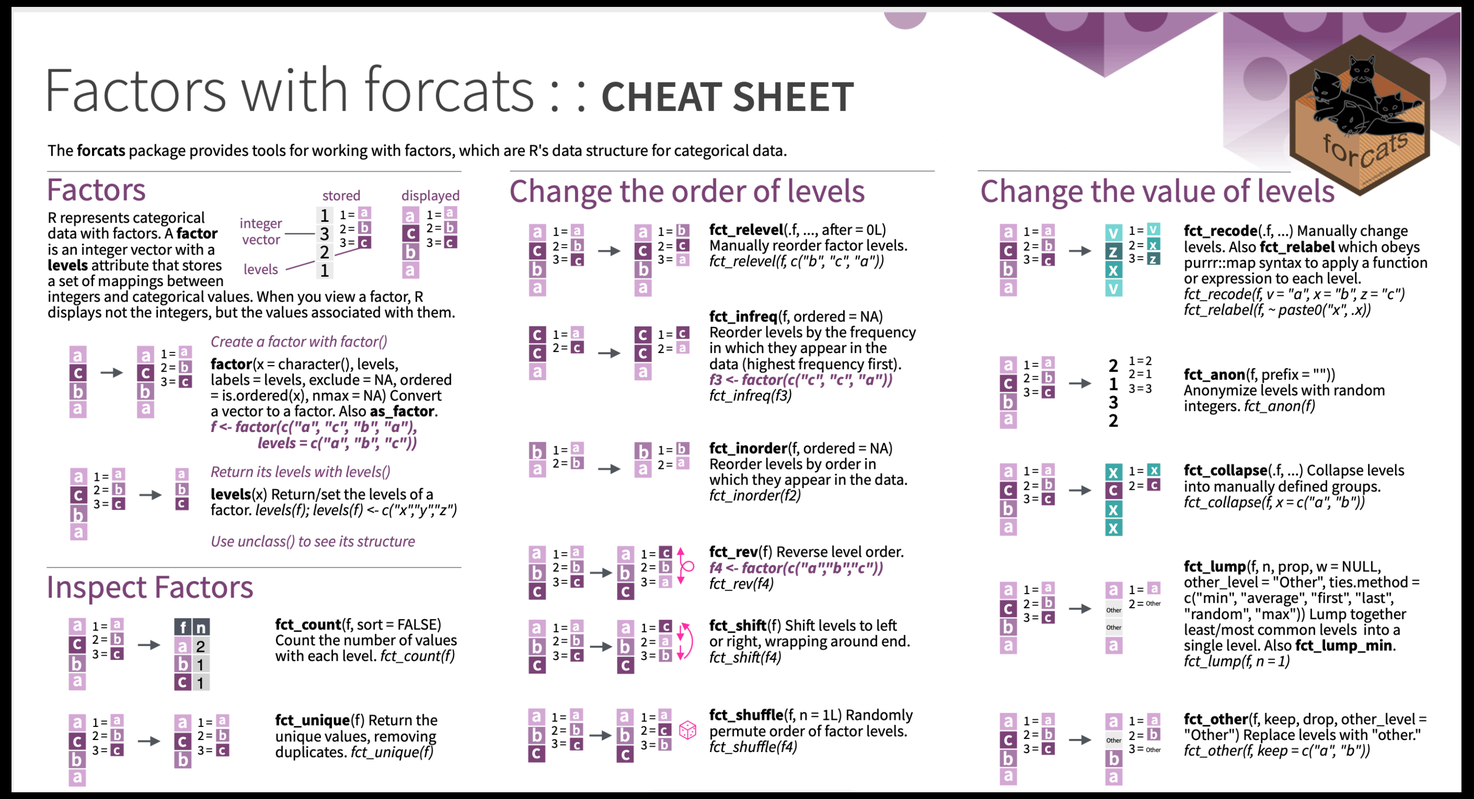
"Railroad Diagrams"



Tons of books on Reg Ex

Online tutorials and cheat sheets as well

Factors



forcats Cheat Sheet

Please take out

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
```

factors

Stored as an integer vector with a levels attribute

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

forcats

Simple functions for working with factors.

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

Warm Up

Decide in your group:

Which religions watch the least TV?

Do married people watch more or less TV than single people?

gss_cat

```
library(forcats)
gss_cat
```

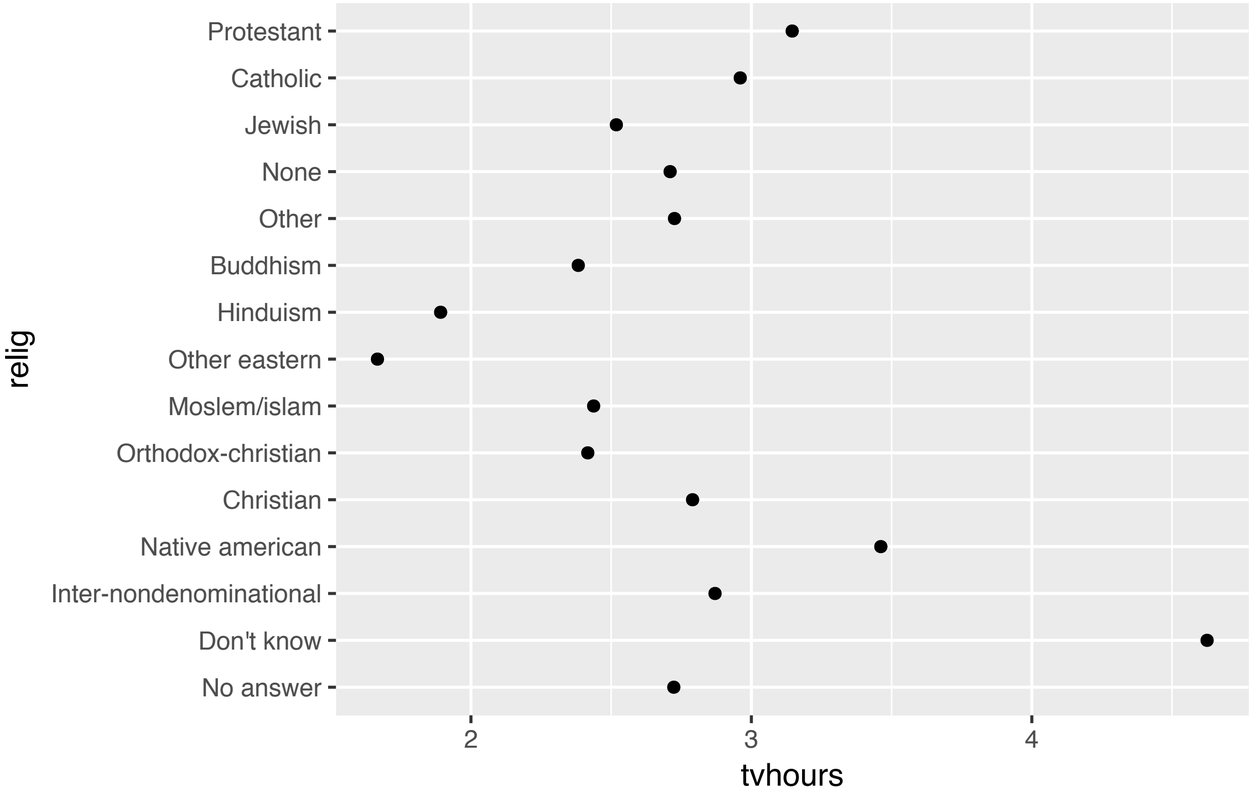
A sample of data from the General Social Survey, a long-running US survey conducted by NORC at the University of Chicago.

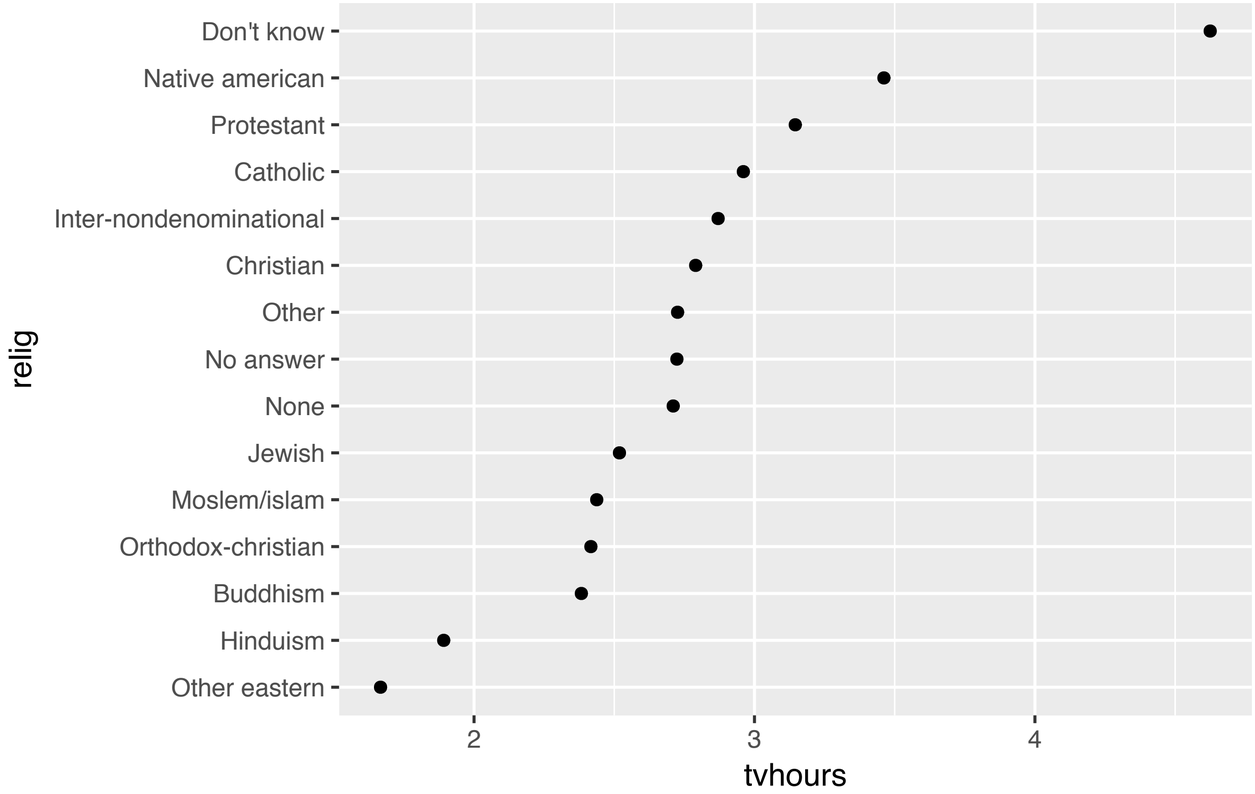
tvhours <int>	marital <fctr>	age <int>	race <fctr>	partyid <fctr>	relig <fctr>
12	Never married	26	White	Ind,near rep	Protestant
NA	Divorced	48	White	Not str republican	Protestant
2	Widowed	67	White	Independent	Protestant
4	Never married	39	White	Ind,near rep	Orthodox-christian
1	Divorced	25	White	Not str democrat	None
NA	Married	25	White	Strong democrat	Protestant
3	Never married	36	White	Not str republican	Christian

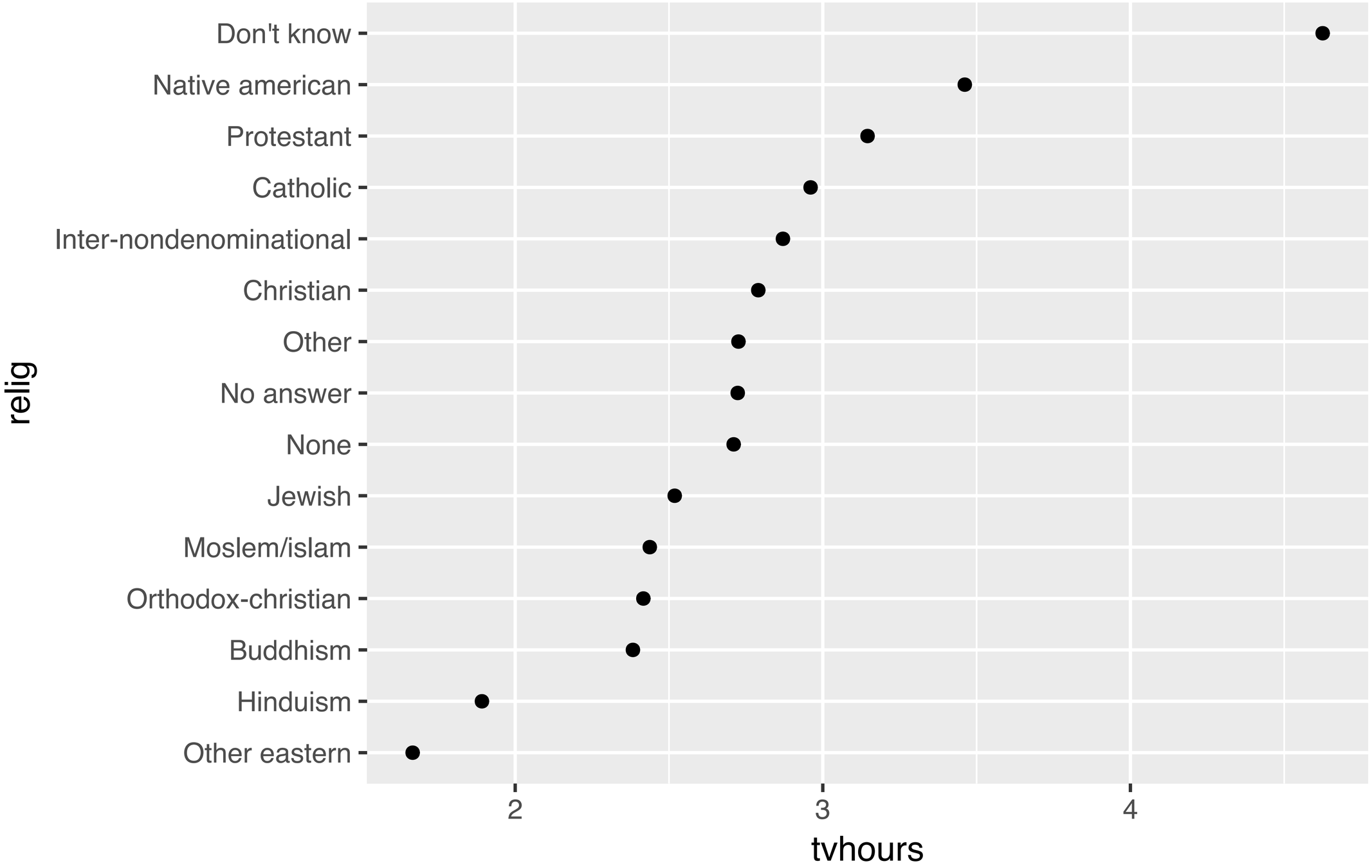
Which religions watch the least TV?

```
gss_cat %>%
  drop_na(tvhours) %>%
  group_by(relig) %>%
  summarise(tvhours = mean(tvhours)) %>%
  ggplot(aes(tvhours, relig)) +
    geom_point()
```

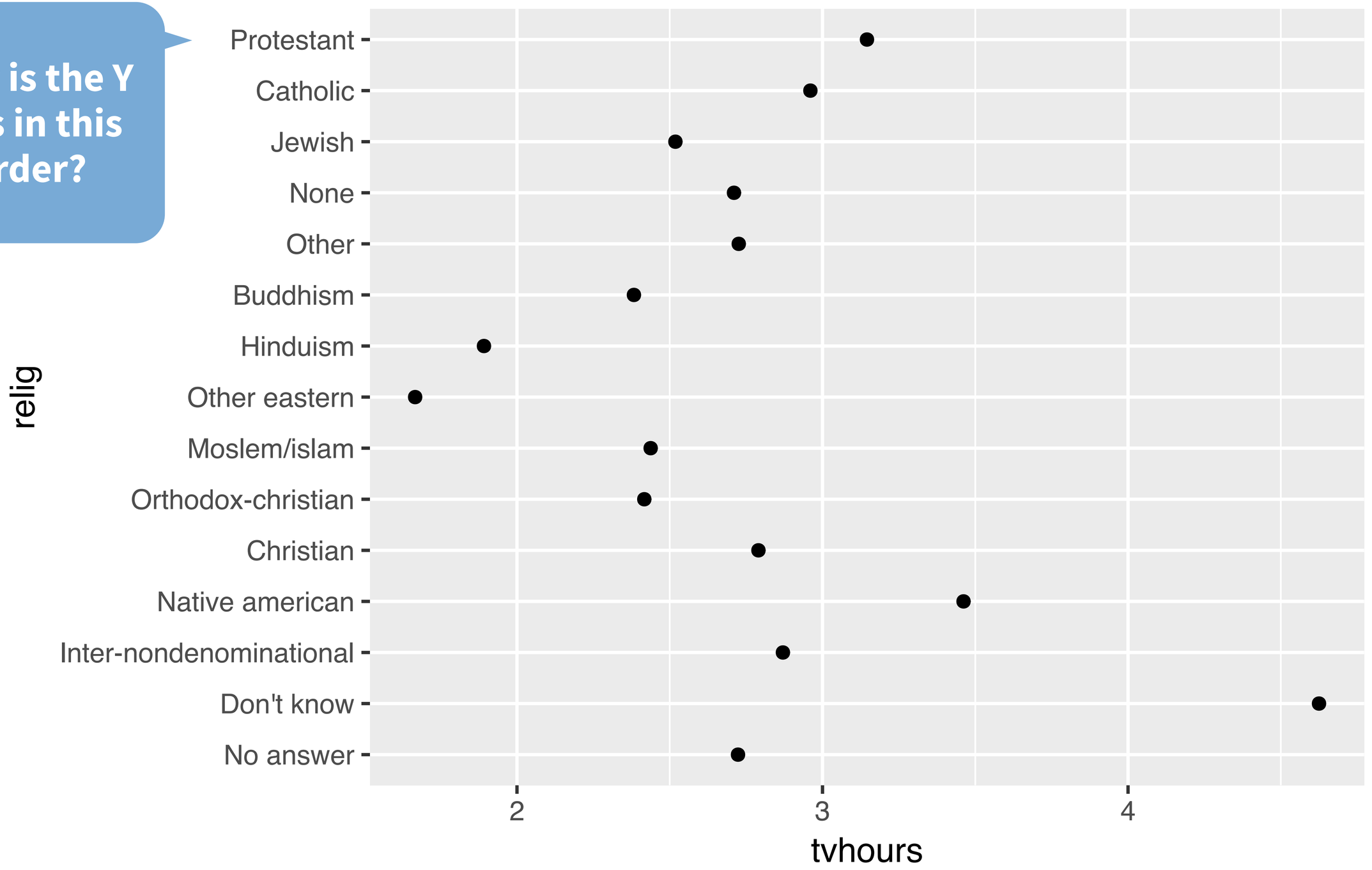
Which do you prefer?






Don't know
Native american
Protestant
Catholic
Inter-nondenominational
Christian
Other
No answer
None
Jewish
Moslem/islam
Orthodox-christian
Buddhism
Hinduism
Other eastern
relig
2
3
4
tvhours

Why is the Y axis in this order?



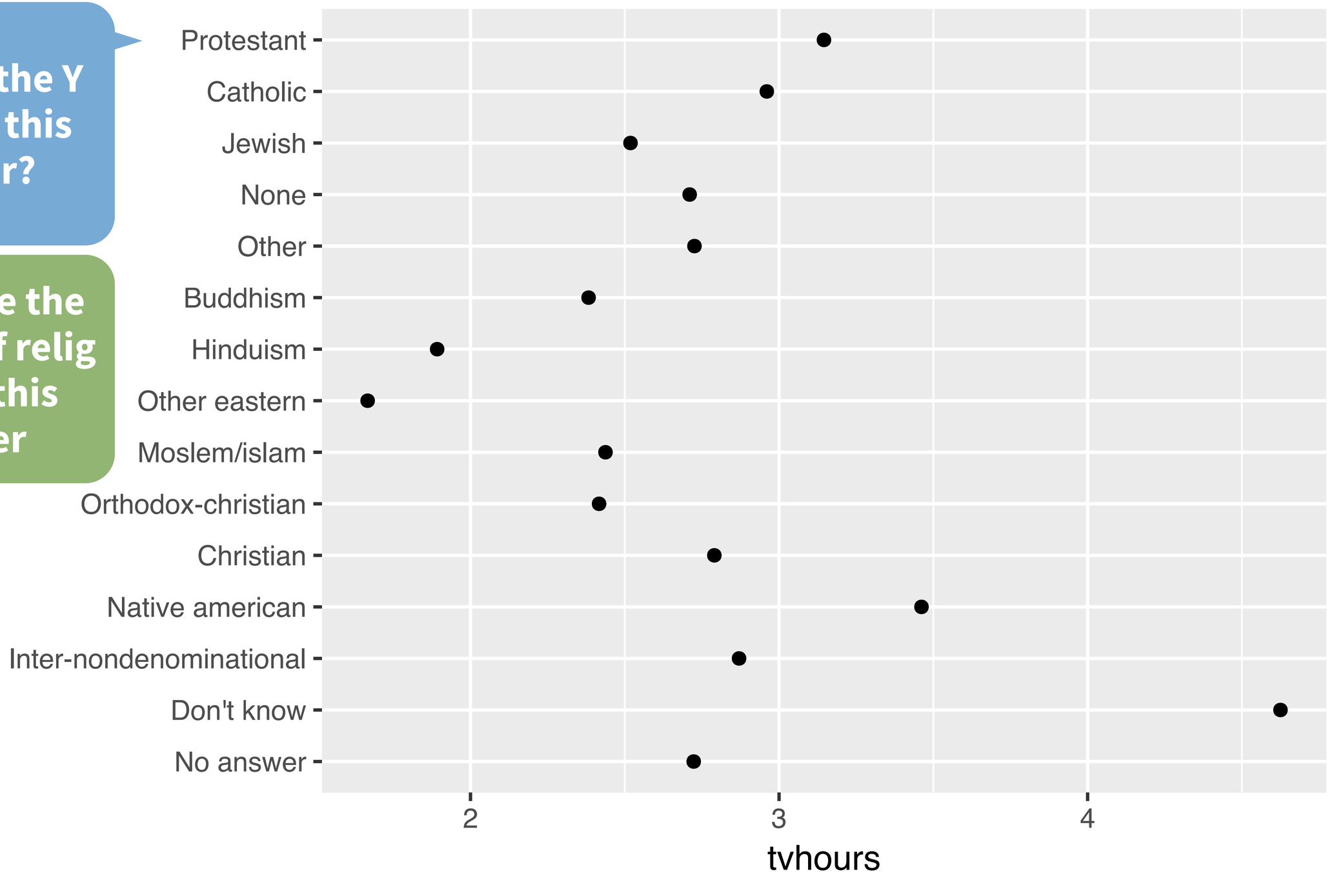
levels()

Use **levels()** to access a factor's levels

```
levels(gss_cat$relig)
##  [1] "No answer"                "Don't know"
## [3] "Inter-nondenominational" "Native american"
## [5] "Christian"                "Orthodox-christian"
## [7] "Moslem/islam"             "Other eastern"
## [9] "Hinduism"                 "Buddhism"
## [11] "Other"                    "None"
## [13] "Jewish"                   "Catholic"
## [15] "Protestant"               "Not applicable"
```

Why is the Y axis in this order?

Because the levels of relig have this order

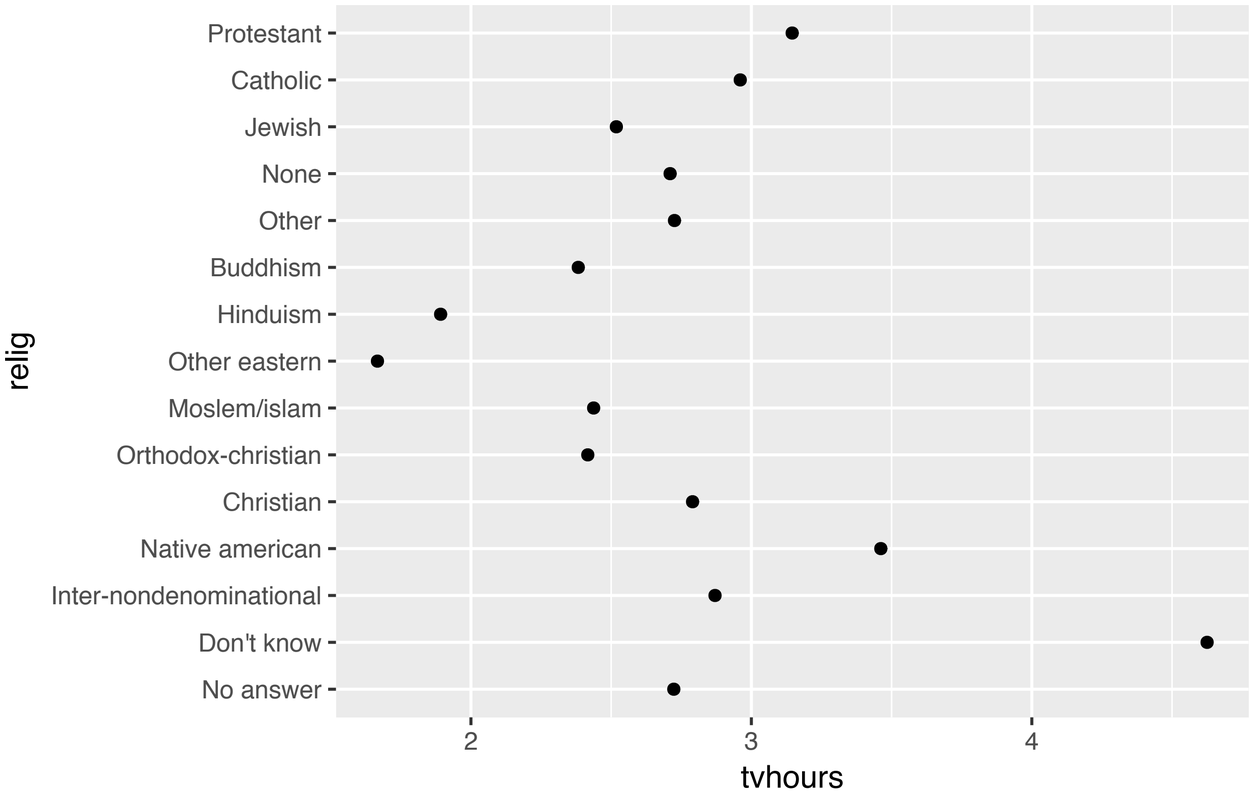


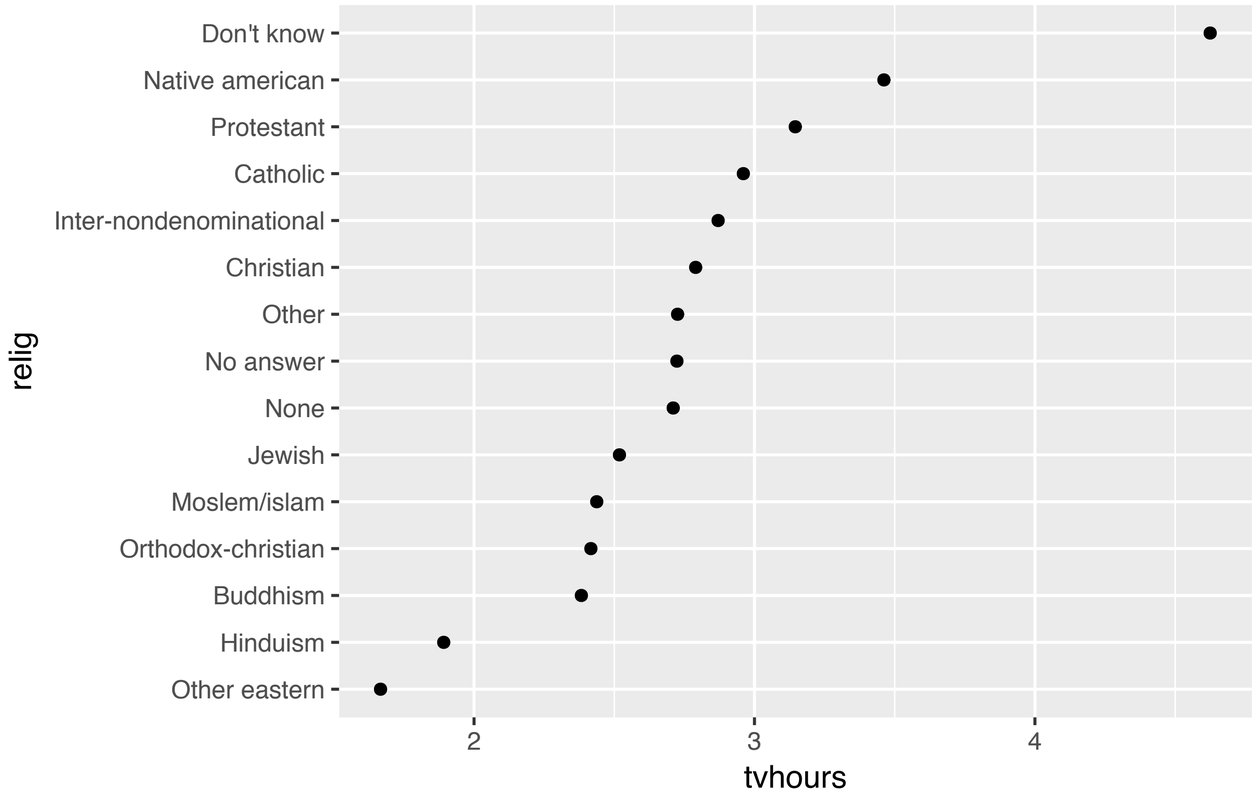
Most useful skills

1. Reorder the levels
2. Recode the levels
3. Collapse levels

Reordering levels

Reorder **relig** by **tvhours**





fct_reorder()

Reorders the levels of a factor based on the result of fun(x) applied to each group of cases (grouped by level).

```
fct_reorder(f, x, fun = median, …, .desc = FALSE)
```

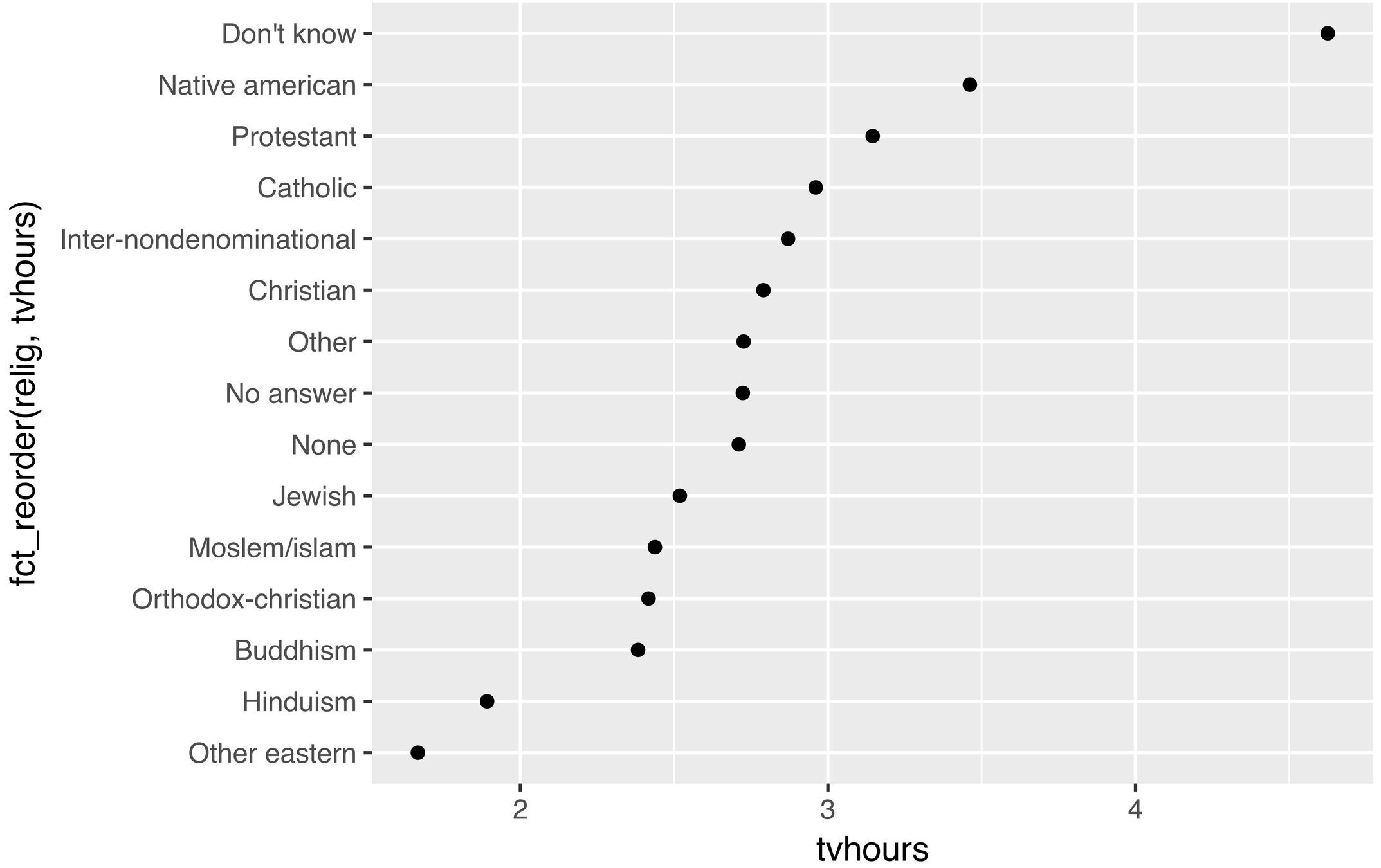
factor to reorder

variable to reorder by (in conjunction with fun)

function to reorder by (in conjunction with x)

put in descending order?

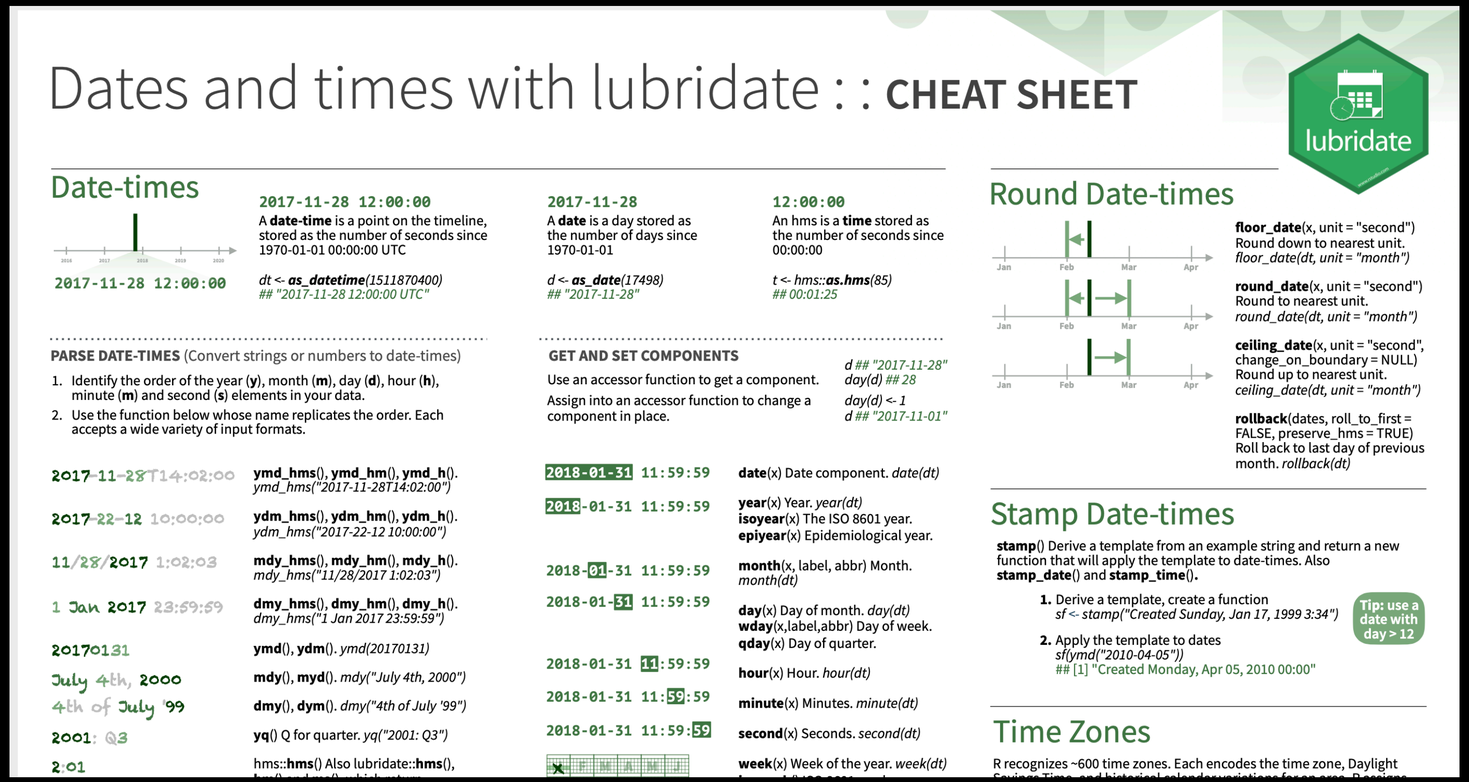
```
gss_cat %>%
  drop_na(tvhours) %>%
  group_by(relig) %>%
  summarise(tvhours = mean(tvhours)) %>%
  ggplot(aes(tvhours, fct_reorder(relig, tvhours))) +
    geom_point()
```


Don't know
Native american
Protestant
Catholic
Inter-nondenominational
Christian
Other
No answer
None
Jewish
Moslem/islam
Orthodox-christian
Buddhism
Hinduism
Other eastern
fct_reorder(relig, tvhours)
2
3
4
tvhours

Most useful skills

1. Reorder the levels
2. Recode the levels
3. Collapse levels

Date times



Dates and times with lubridate : : CHEAT SHEET

lubridate

Date-times

2017-11-28 12:00:00

2017-11-28 12:00:00
A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

*dt <- **as_datetime**(1511870400)*
"2017-11-28 12:00:00 UTC"

2017-11-28
A **date** is a day stored as the number of days since 1970-01-01

*d <- **as_date**(17498)*
"2017-11-28"

12:00:00
An hms is a **time** stored as the number of seconds since 00:00:00

*t <- hms::**as.hms**(85)*
00:01:25

PARSE DATE-TIMES (Convert strings or numbers to date-times)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
2. Use the function below whose name replicates the order. Each accepts a wide variety of input formats.

2017-11-28T14:02:00	**ymd_hms**(), **ymd_hm**(), **ymd_h**(). *ymd_hms("2017-11-28T14:02:00")*
2017-22-12 10:00:00	**ydm_hms**(), **ydm_hm**(), **ydm_h**(). *ydm_hms("2017-22-12 10:00:00")*
11/28/2017 1:02:03	**mdy_hms**(), **mdy_hm**(), **mdy_h**(). *mdy_hms("11/28/2017 1:02:03")*
1 Jan 2017 23:59:59	**dmy_hms**(), **dmy_hm**(), **dmy_h**(). *dmy_hms("1 Jan 2017 23:59:59")*
20170131	**ymd**(), **ydm**(). *ymd(20170131)*
July 4th, 2000	**mdy**(), **myd**(). *mdy("July 4th, 2000")*
4th of July '99	**dmy**(), **dym**(). *dmy("4th of July '99")*
2001: Q3	**yq**() Q for quarter. *yq("2001: Q3")*
2:01	hms::**hms**() Also lubridate::**hms**(),

GET AND SET COMPONENTS

Use an accessor function to get a component.

d ## "2017-11-28"
day(d) ## 28

Assign into an accessor function to change a component in place.

day(d) <- 1
d ## "2017-11-01"

2018-01-31 11:59:59	**date**(x) Date component. *date(dt)*
2018-01-31 11:59:59	**year**(x) Year. *year(dt)* **isoyear**(x) The ISO 8601 year. **epiyear**(x) Epidemiological year.
2018-01-31 11:59:59	**month**(x, label, abbr) Month. *month(dt)*
2018-01-31 11:59:59	**day**(x) Day of month. *day(dt)* **wday**(x,label,abbr) Day of week. **qday**(x) Day of quarter.
2018-01-31 11:59:59	**hour**(x) Hour. *hour(dt)*
2018-01-31 11:59:59	**minute**(x) Minutes. *minute(dt)*
2018-01-31 11:59:59	**second**(x) Seconds. *second(dt)*
	week(x) Week of the year. *week(dt)*

Round Date-times

floor_date(x, unit = "second") Round down to nearest unit. *floor_date(dt, unit = "month")*

round_date(x, unit = "second") Round to nearest unit. *round_date(dt, unit = "month")*

ceiling_date(x, unit = "second", change_on_boundary = NULL) Round up to nearest unit. *ceiling_date(dt, unit = "month")*

rollback(dates, roll_to_first = FALSE, preserve_hms = TRUE) Roll back to last day of previous month. *rollback(dt)*

Stamp Date-times

stamp() Derive a template from an example string and return a new function that will apply the template to date-times. Also **stamp_date**() and **stamp_time**().

1. Derive a template, create a function
   sf <- stamp("Created Sunday, Jan 17, 1999 3:34")
2. Apply the template to dates
   sf(ymd("2010-04-05"))
   ## [1] "Created Monday, Apr 05, 2010 00:00"

Tip: use a date with day > 12

Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight

lubridate Cheat Sheet

Please take out

Quiz

Does every year have 365 days?

Quiz

Does every day have 24 hours?

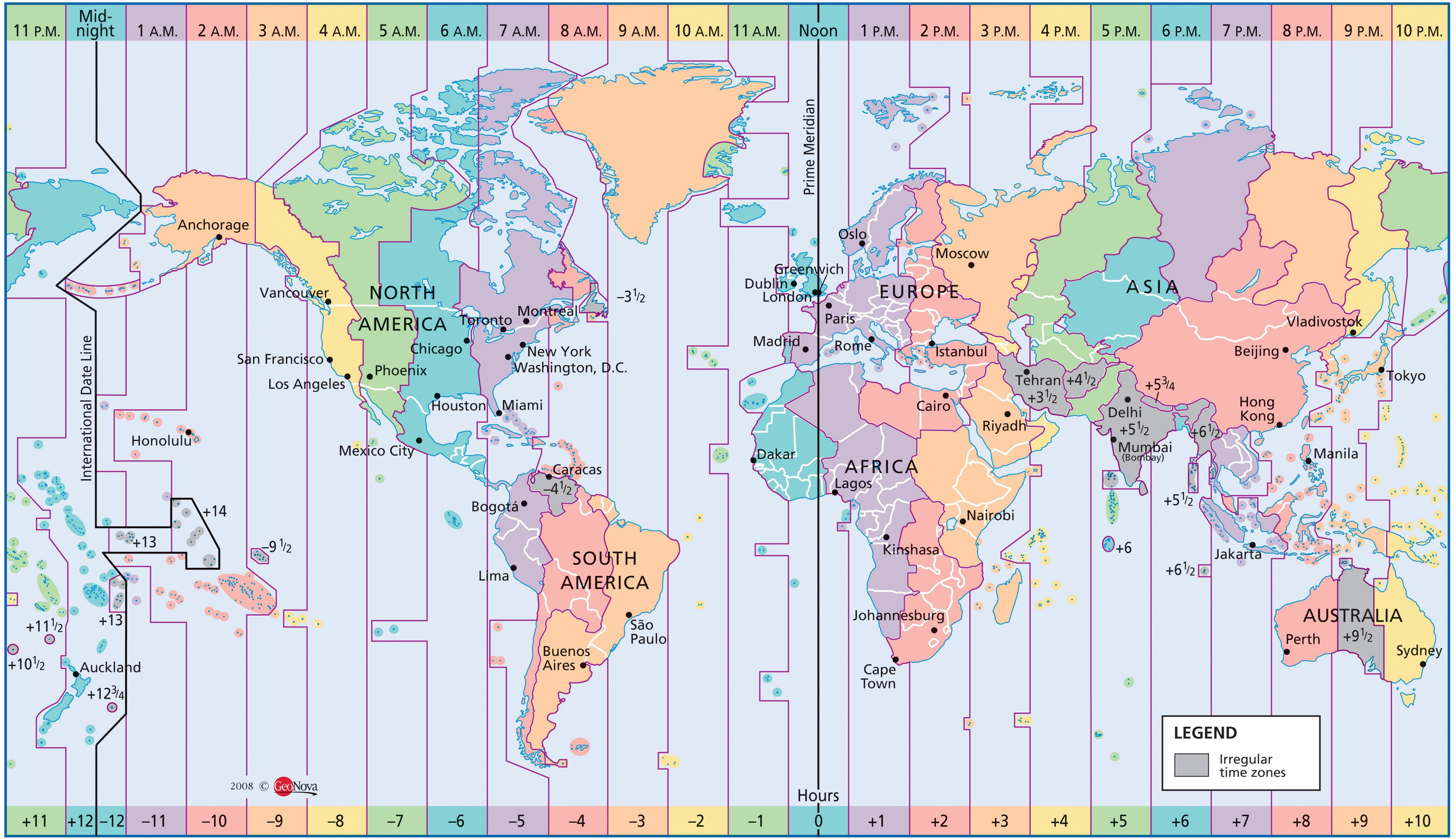
Quiz

Does every minute have 60 seconds?

Quiz

What does a month measure?

WORLD TIME ZONES



Most useful skills

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

Warm Up

Decide in your group:

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

Creating dates and times

hms

A class for representing just clock times.

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

lubridate

Functions for working with dates and time spans

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

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")
```

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

Accessing and changing components

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"
```

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	

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

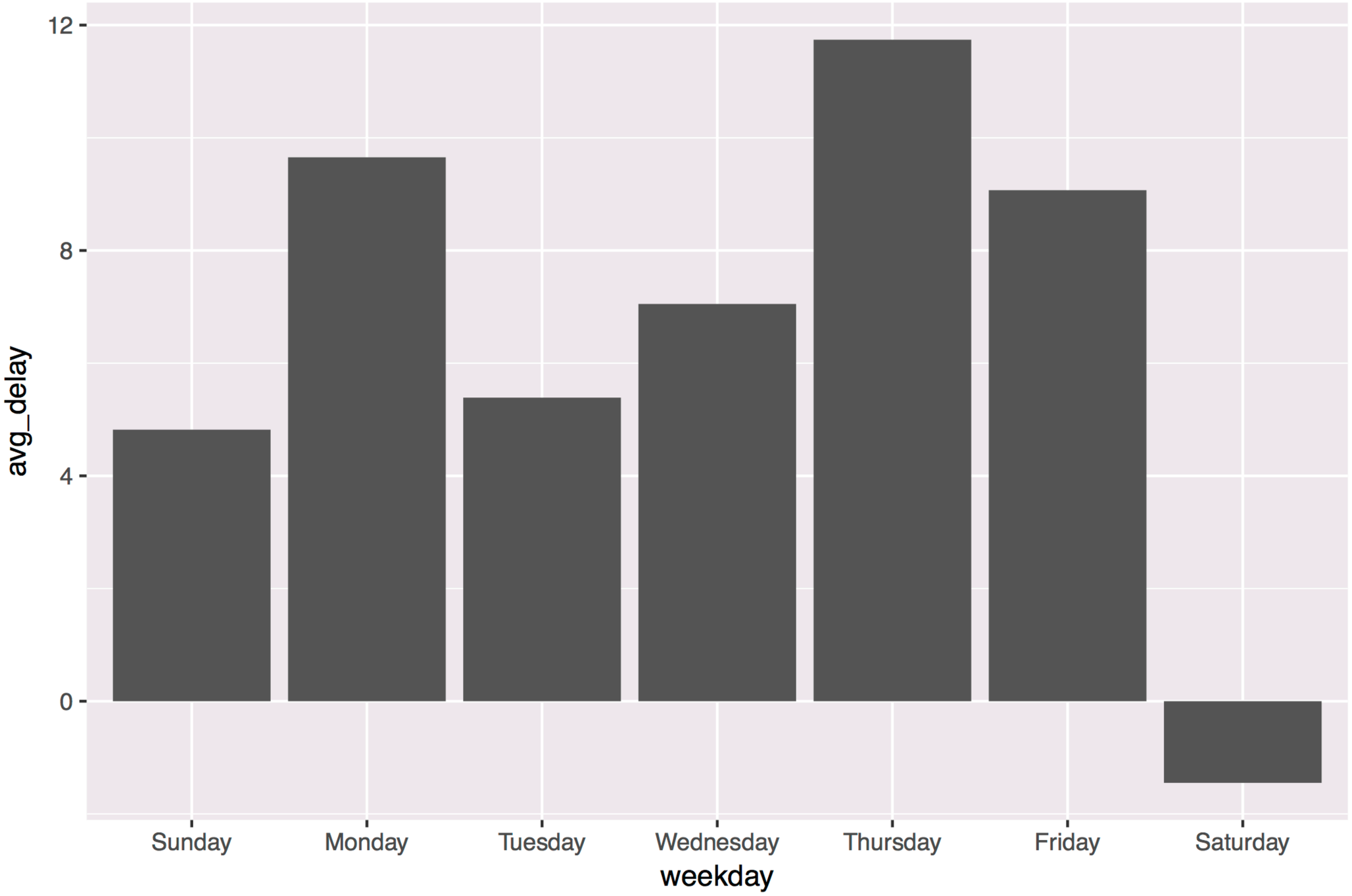
Fill in the blanks to:

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

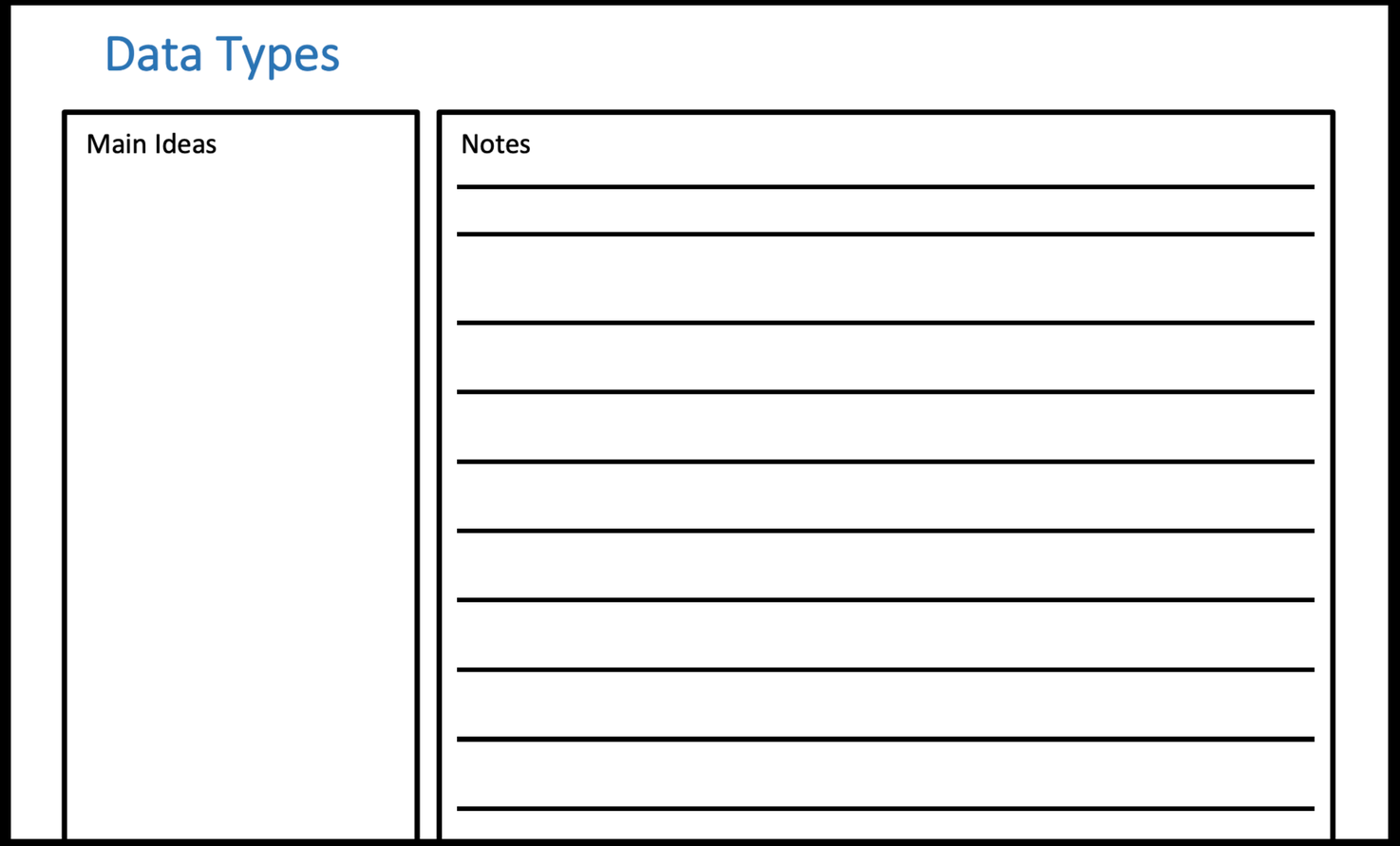
Calculate the average **arr_delay** by day of the week.

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

```
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

Data types with



Notes Form

Please Take out