Data types with

Navigate to the main page of the class: **https://astamm.github.io/data-science-with-r/**.

Download **08-Types-Exercises.qmd** from the outline table and open them.

Recall

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

Common data types

1. Logicals
2. Strings
3. Factors
4. Dates and Times

Logicals

Logicals

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

```
TRUE
##  TRUE

typeof(TRUE)
##  "logical"

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

Details

⚠ The class of service you searched may not be available on one or more flights

BNA - ORD
Flight 1 of 2

ORD - YVR
Flight 2 of 2

Nashville, TN to Chicago, IL

Thursday, July 26, 2018

4:10 PM → 6:03 PM

AA 3246 ▪ CR7-Canadair RJ 700

Operated by SkyWest Airlines As American Eagle

Travel info

Travel time: **1h 53m**
Connection time: **2h 33m**

Performanc

On time: **52**
Late: **43%**

Main Cabin

Meals: Beverage service
Booking code: V
Class: Economy

Business

Meals: Beverage service
Booking code: I
Class: First

* This is based on information from the month of May 2018
** The on-time arrival percentage for the selected flight is based on arrival within 14 minutes after

Performance*

On time: **52%****
Late: **43%**

** The on-time arrival percentage for the selected flight is based on arrival within 14 minutes after the scheduled arrival as reported monthly to the U.S. Department of Transportation.

Warm Up

Did you fly here?

Did your flight arrive late?

```
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. Logical tests
2. Math with logicals

Quiz

What does this return?

TRUE + 0

NOT AN ERROR!

Quiz

What does this return?

TRUE + 0

1

Math with logicals

TRUE = 1

FALSE = 0

Quiz

What will this return?

sum(c(FALSE, FALSE, TRUE, FALSE)**)**

THE NUMBER OF TRUES

Quiz

What will this tell us?

sum(flights$arr_delay > 0**)**

THE NUMBER OF DELAYED FLIGHTS

Quiz

What will this tell us?

mean(flights$arr_delay > 0**)**

THE PROPORTION OF DELAYED FLIGHTS

TRUE = 1

FALSE = 0

sum() = **number** that pass

mean() = **proportion** that pass

Your Turn 1

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

Then, filter to rows where **delayed** does not equal NA.

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) |>
  filter(!is.na(delayed)) |>
  summarise(total = sum(delayed), prop = mean(delayed))
## # A tibble: 1 × 2
##   total      prop
##   <int>     <dbl>
## 1 133004 0.4063101
```

Strings

(character) **strings**

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

```
"one"
##  "one"

typeof("one")
##  "character"

typeof("oops. I'm stuck in a string)
+ ")
##  "character"
```

Warm Up

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

babynames

Names of male and female babies born in the US from 1880 to 2017. 1.9M rows.

```
# install.packages("babynames")
library(babynames)
```

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(tidyverse)
```


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"
```

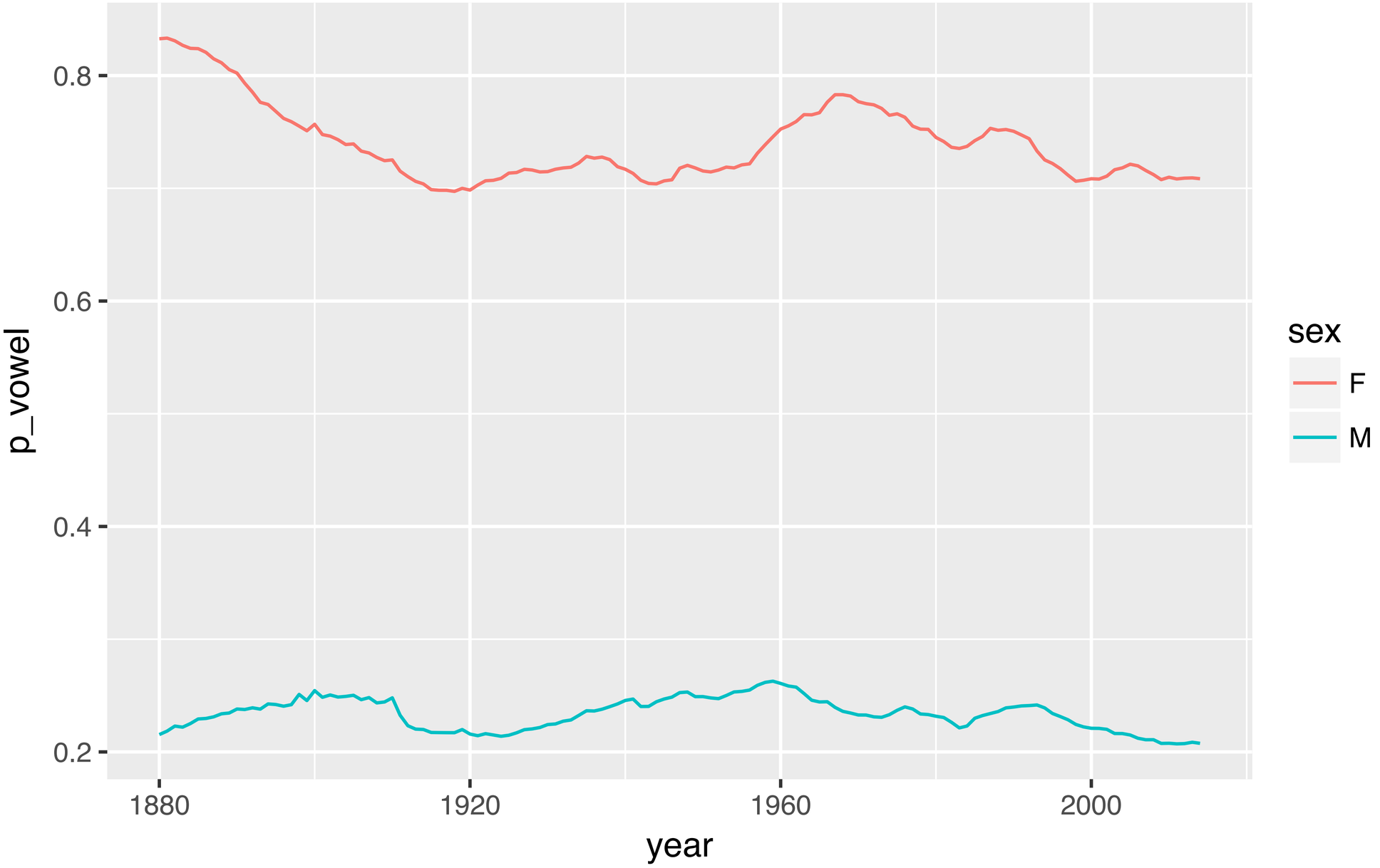
Your Turn 2

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



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 in each string. *str_replace_all(fruit, "a", "-")*

str_to_lower(string, locale = "en")[1] Convert strings to lower case.
str_to_lower(sentences)

str_to_upper(string, locale = "en")[1] Convert strings to upper case.
str_to_upper(sentences)

str_to_title(string, locale = "en")[1] Convert strings to title case. *str_to_title(sentences)*

Join and Split

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

str_c(..., sep = "", **collapse = NULL**) 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)*

str_split_fixed(string, **pattern**, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split** to return a list of substrings.
str_split_fixed(fruit, " ", n=2)

str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. *str_glue("Pi is {pi}")*

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate.
str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")

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)

Helpers

str_conv(string, encoding) Override the encoding of a string. *str_conv(fruit,"ISO-8859-1")*

str_view(string, **pattern**, match = NA) View HTML rendering of first regex match in each string. *str_view(fruit, "[aeiou]")*

str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches.
str_view_all(fruit, "[aeiou]")

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. *str_wrap(sentences, 20)*

[1] See bit.ly/ISO639-1 for a complete list of locales.

Atomic types

R recognizes six elemental data types.

```
typeof(1)          ## double
typeof(1L)         ## integer
typeof(TRUE)       ## logical
typeof("one")      ## character
typeof(raw(1))     ## raw
typeof(1i)         ## complex
```

So what are dates and factors?

Classes

You can use atomic types to build new classes.

```
x <- 1560139200
```

```
x                    ## 1560139200
typeof(x)            ## double
```

Classes

You can use atomic types to build new classes.

```
x <- 1560139200
class(x) <- "POSIXct"
```

```
x                       ## "2019-06-10 EDT"
typeof(x)               ## double
```

Classes

You can use atomic types to build new classes.

```
x <- 1560139200
class(x) <- "POSIXct"
```

```
x                       ## "2019-06-10 EDT"
typeof(x)               ## double
class(x)                ## POSIXct
```

Classes

```
eyes <- c(1L, 3L, 3L)
```

```
eyes                    ## 1 3 3

typeof(eyes)            ## integer
```

Classes

```
eyes <- c(1L, 3L, 3L)
levels(eyes) <- c("blue", "brown", "green")
```

```
eyes                    ## 1 3 3

typeof(eyes)            ## integer
```

Classes

```
eyes <- c(1L, 3L, 3L)
levels(eyes) <- c("blue", "brown", "green")
class(eyes) <- "factor"
```

```
eyes                    ## blue  green green
                        ##    Levels: blue brown green
typeof(eyes)            ## integer
```

Classes

```
eyes <- c(1L, 3L, 3L)
levels(eyes) <- c("blue", "brown", "green")
class(eyes) <- "factor"
```

```
eyes                    ## blue  green green
                        ##    Levels: blue brown green
typeof(eyes)            ## integer
class(eyes)             ## factor
```

Factors

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")
)
```

Stored as an integer vector with a levels attribute

```
eyes
## [1] blue  green green
## Levels: blue brown green

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

forcats

Simple functions for working with factors.

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

Warm Up

Which religions watch the least TV?

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

gss_cat

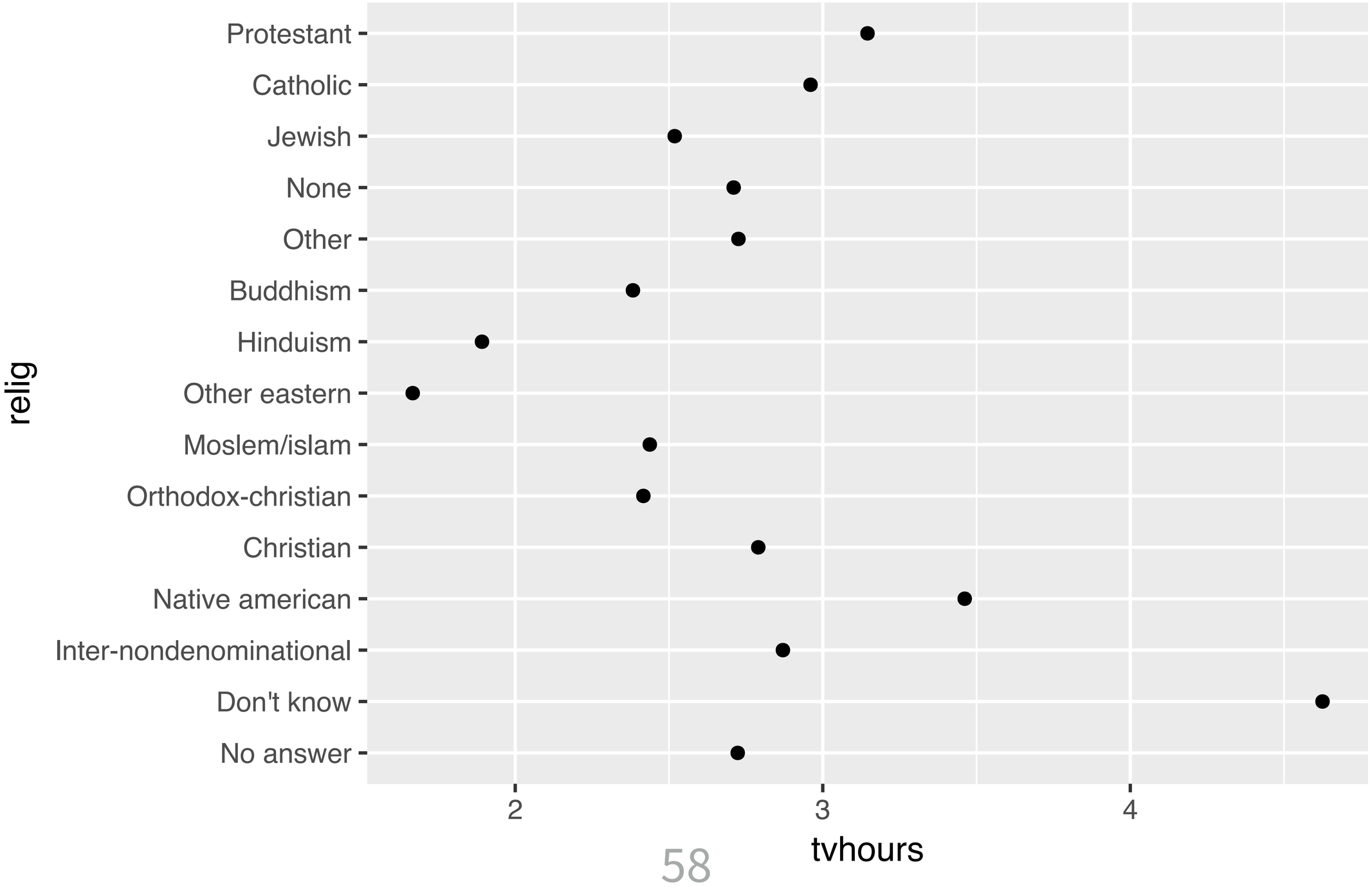
```
library(forcats)
gss_cat
```

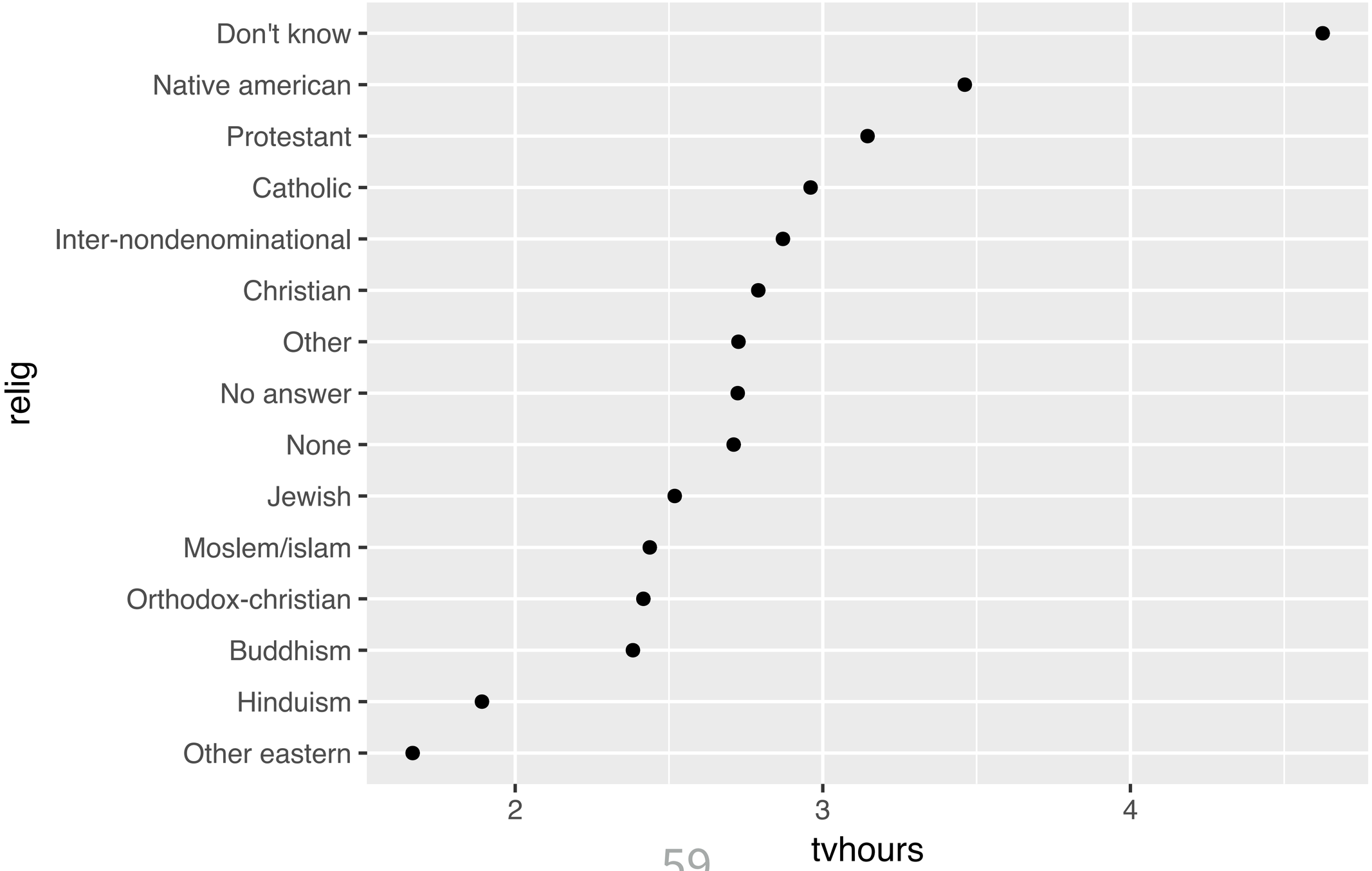
A sample of data from the General Social Survey, a long-running US survey conducted by NORC at the University of Chicago.

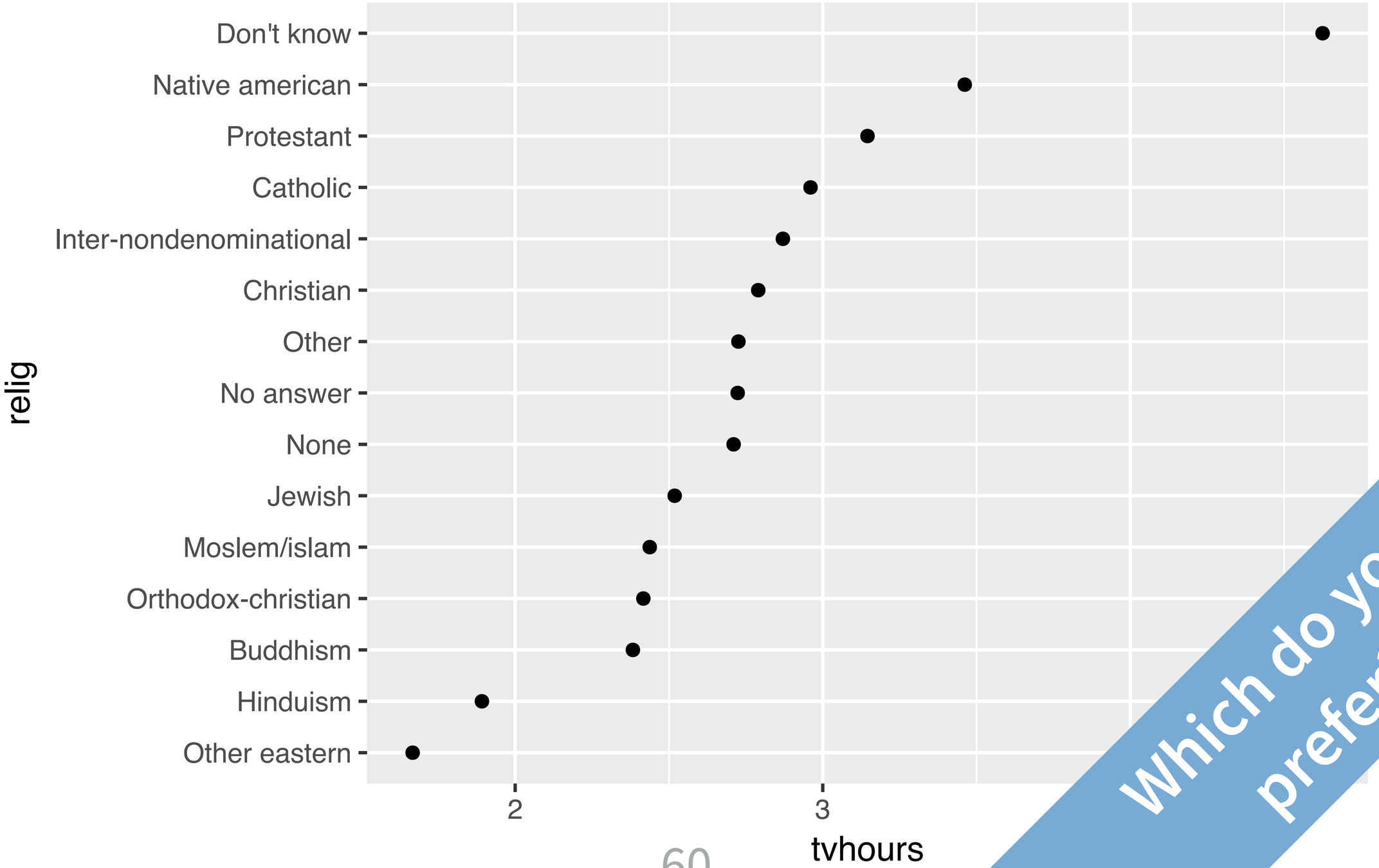
tvhours <int>	marital <fctr>	age <int>	race <fctr>	partyid <fctr>	relig <fctr>
12	Never married	26	White	Ind,near rep	Protestant
NA	Divorced	48	White	Not str republican	Protestant
2	Widowed	67	White	Independent	Protestant
4	Never married	39	White	Ind,near rep	Orthodox-christian
1	Divorced	25	White	Not str democrat	None
NA	Married	25	White	Strong democrat	Protestant
3	Never married	36	White	Not str republican	Christian
NA	Divorced	44	White	Ind,near dem	Protestant
0	Married	44	White	Not str democrat	Protestant

Which religions watch the least TV?

```
gss_cat |>
  filter(!is.na(tvhours)) |>
  group_by(relig) |>
  summarise(tvhours = mean(tvhours)) |>
  ggplot(aes(tvhours, relig)) +
  geom_point()
```

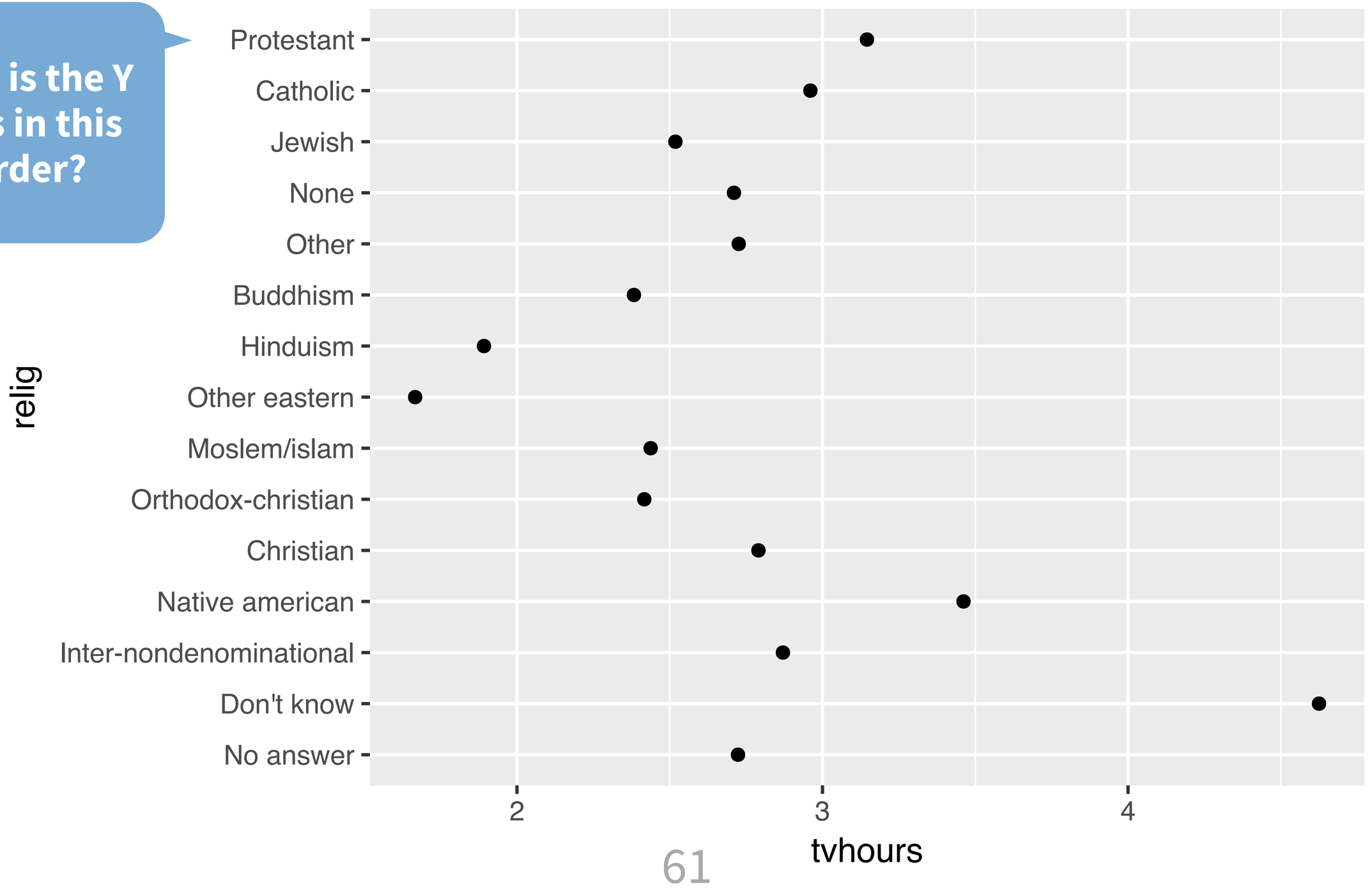

Protestant
Catholic
Jewish
None
Other
Buddhism
Hinduism
Other eastern
Moslem/islam
Orthodox-christian
Christian
Native american
Inter-nondenominational
Don't know
No answer
relig
2
3
4
tvhours


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



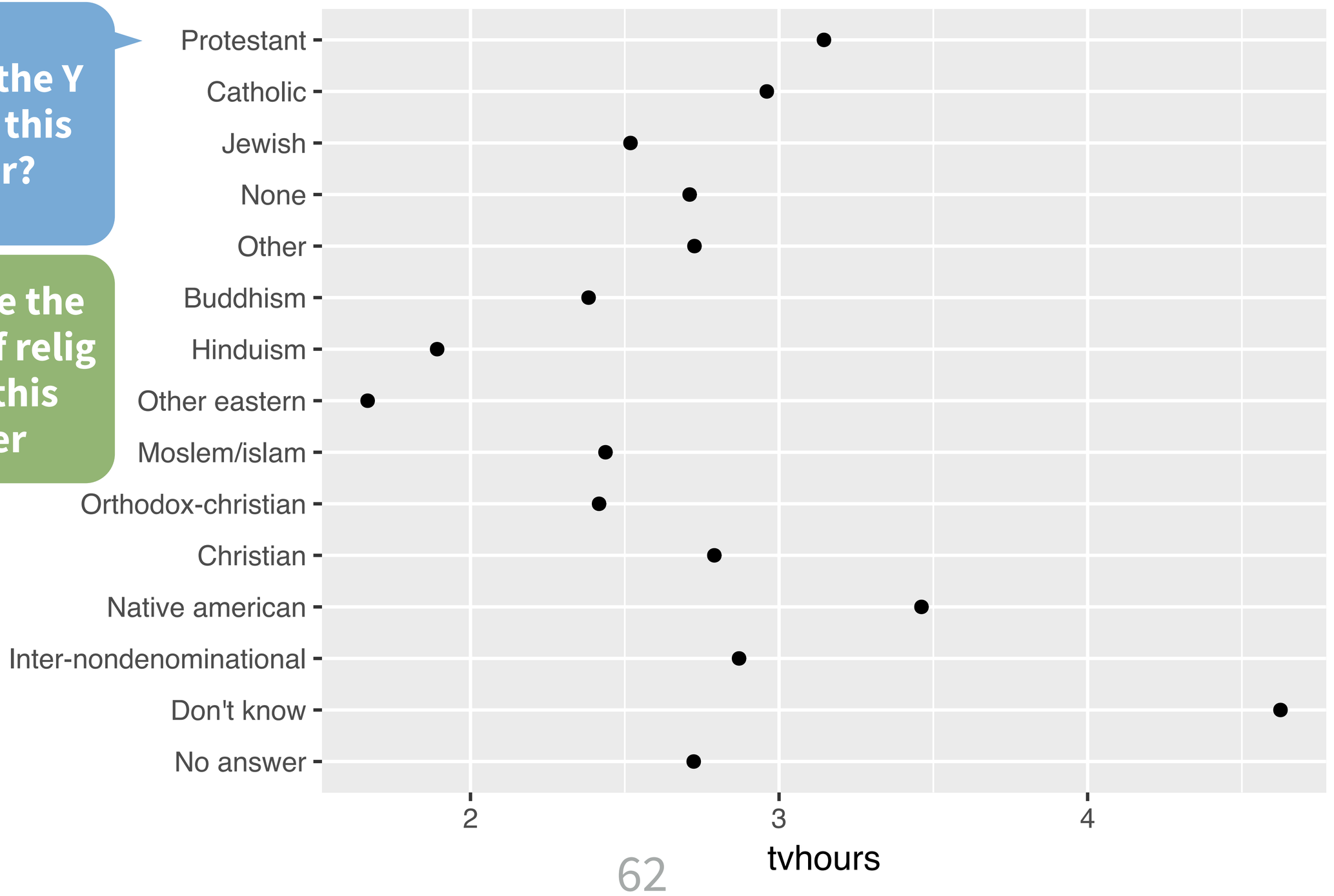
Which do you prefer?

Why is the Y axis in this order?



Why is the Y axis in this order?

Because the levels of relig have 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"
```

Most useful skills

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

Reordering levels

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)
```

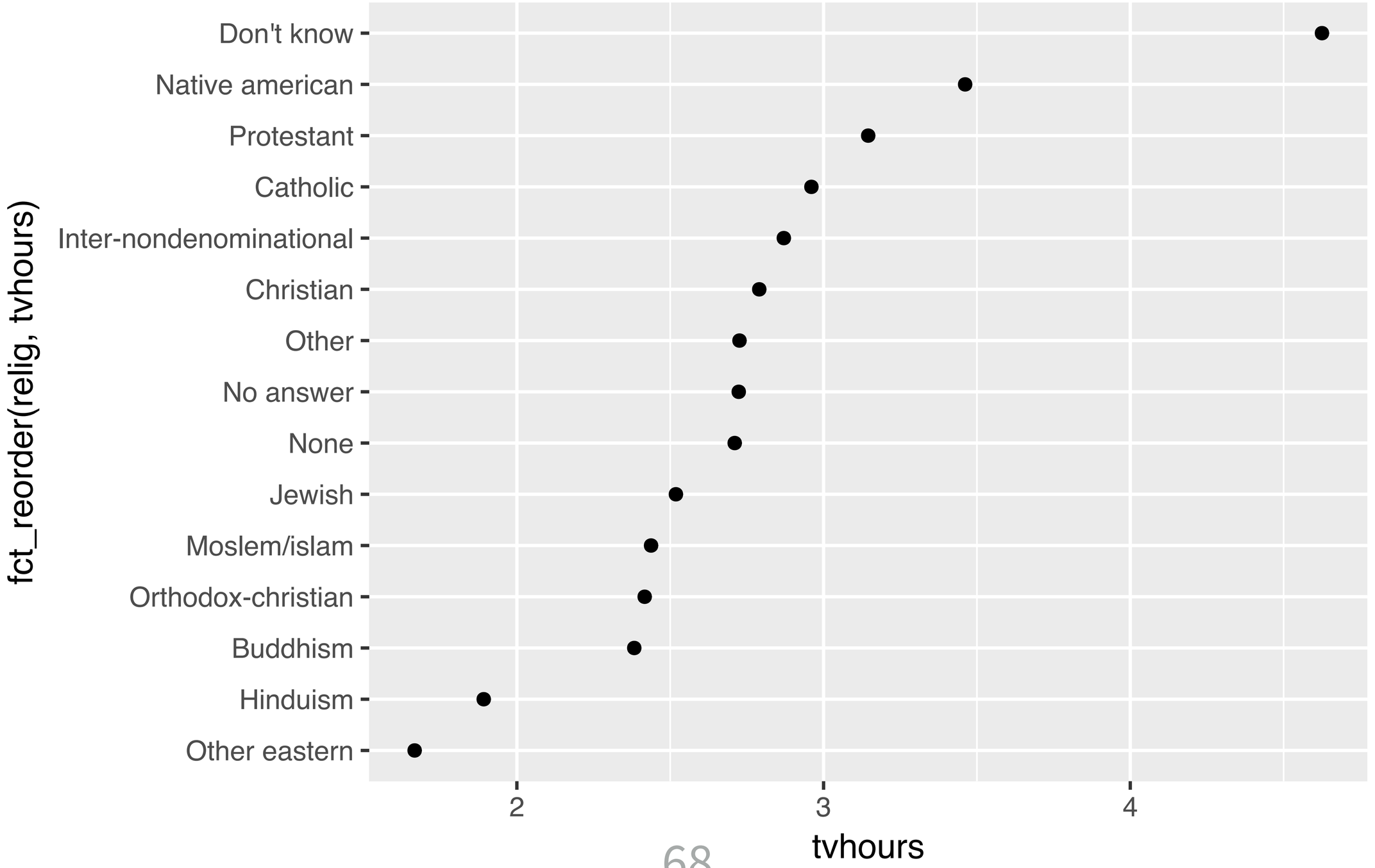
factor to reorder

variable to reorder by (in conjunction with fun)

function to reorder by (in conjunction with x)

put in descending order?

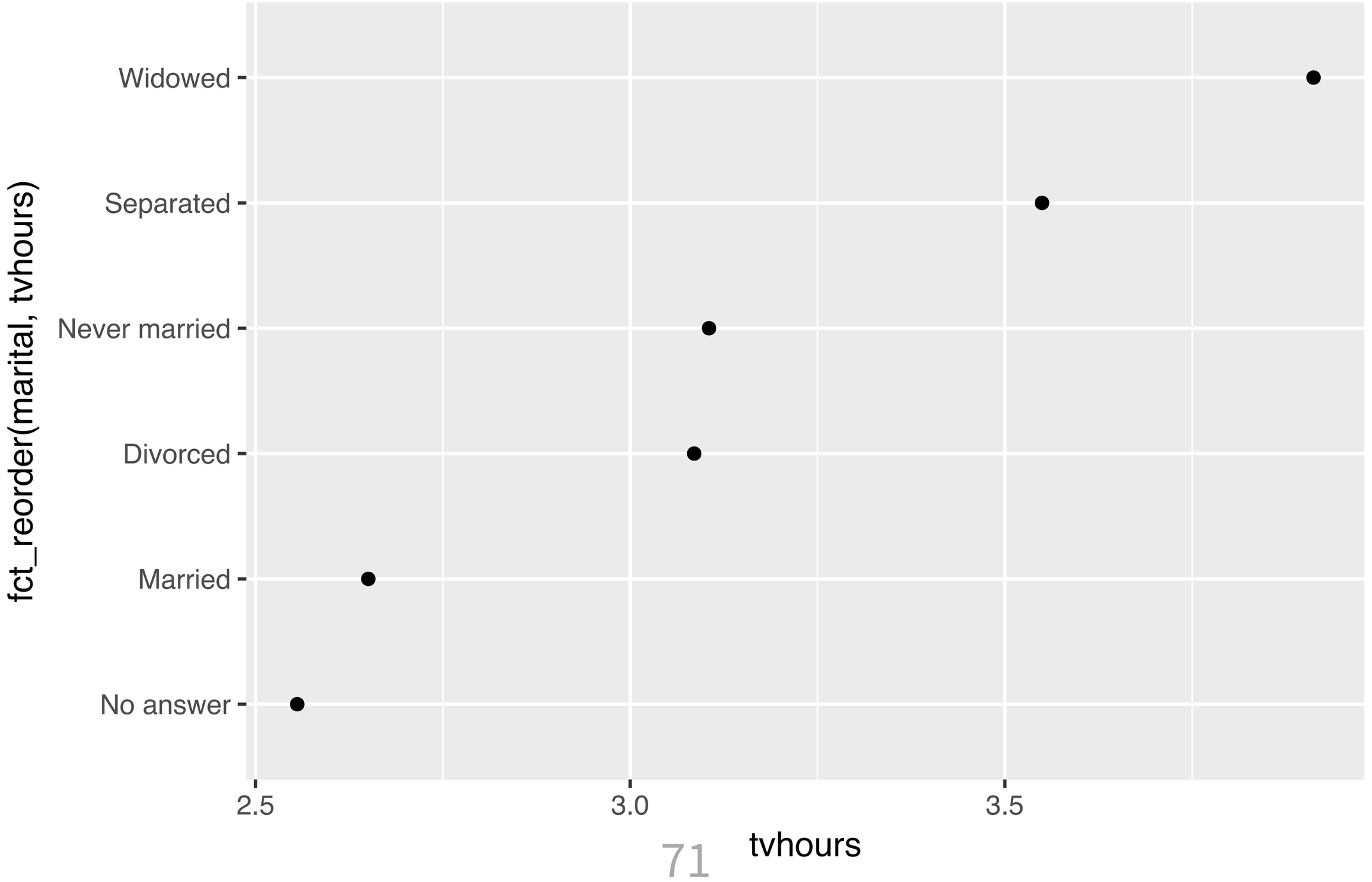
```
gss_cat |>
  filter(!is.na(tvhours)) |>
  group_by(relig) |>
  summarise(tvhours = mean(tvhours)) |>
  ggplot(aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point()
```


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

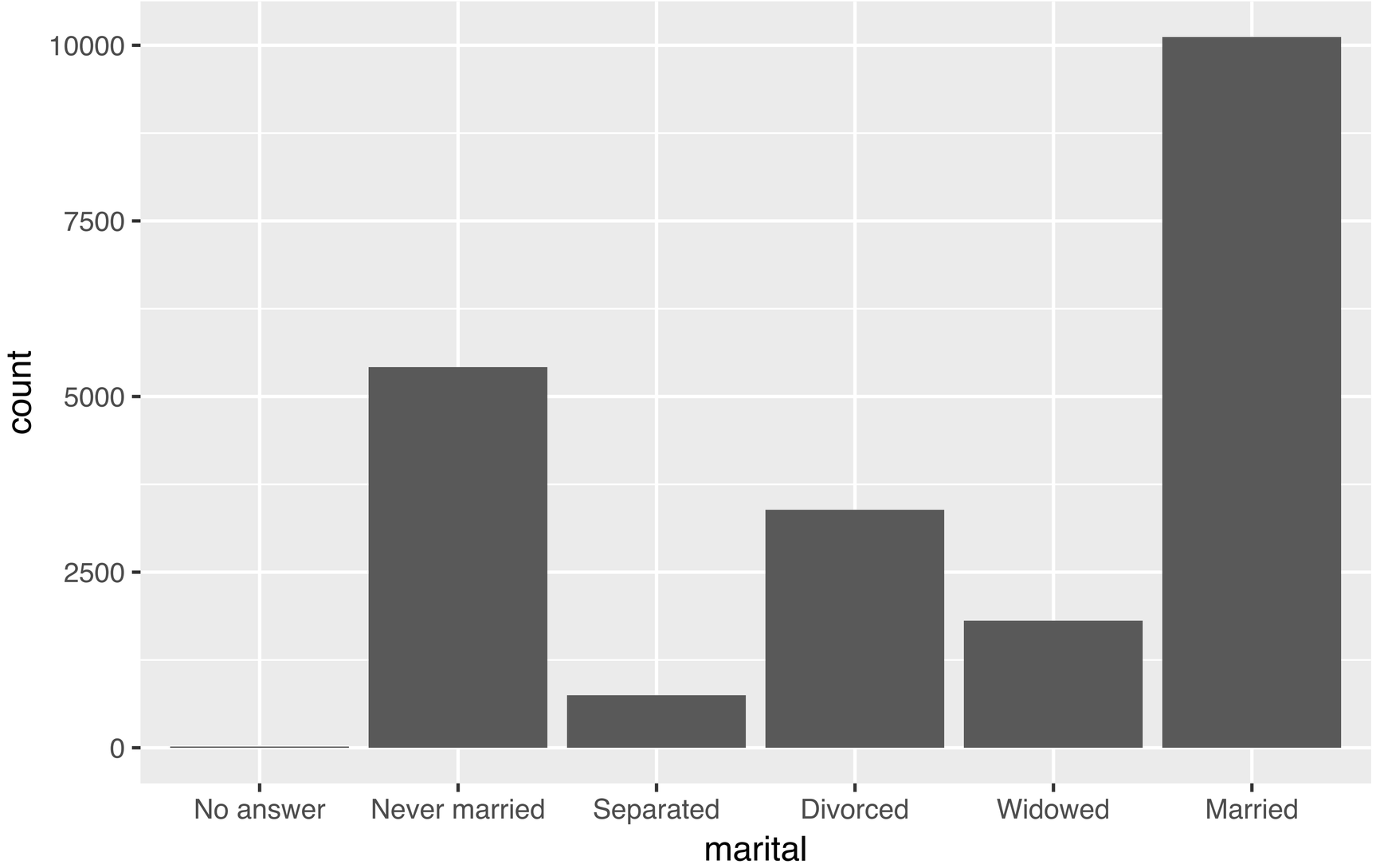
Your Turn 3

Repeat the previous demonstration, some of whose code is in your notebook, to make a sensible graph of average TV consumption by marital status.

```
gss_cat |>
  filter(!is.na(tvhours)) |>
  group_by(marital) |>
  summarise(tvhours = mean(tvhours)) |>
  ggplot(aes(tvhours, fct_reorder(marital, tvhours))) +
  geom_point()
```

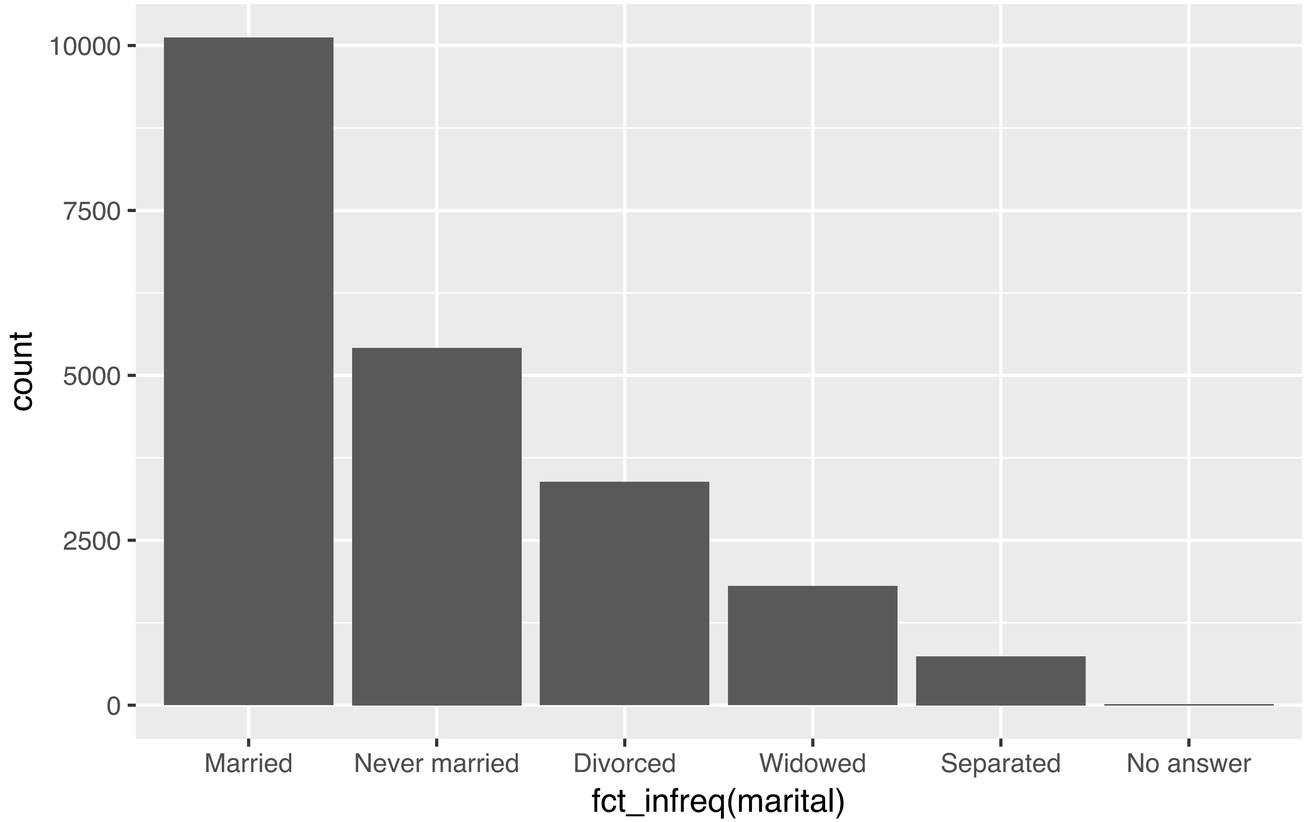

Widowed
Separated
Never married
Divorced
Married
No answer
fct_reorder(marital, tvhours)
2.5
3.0
3.5
tvhours

Similar reordering functions



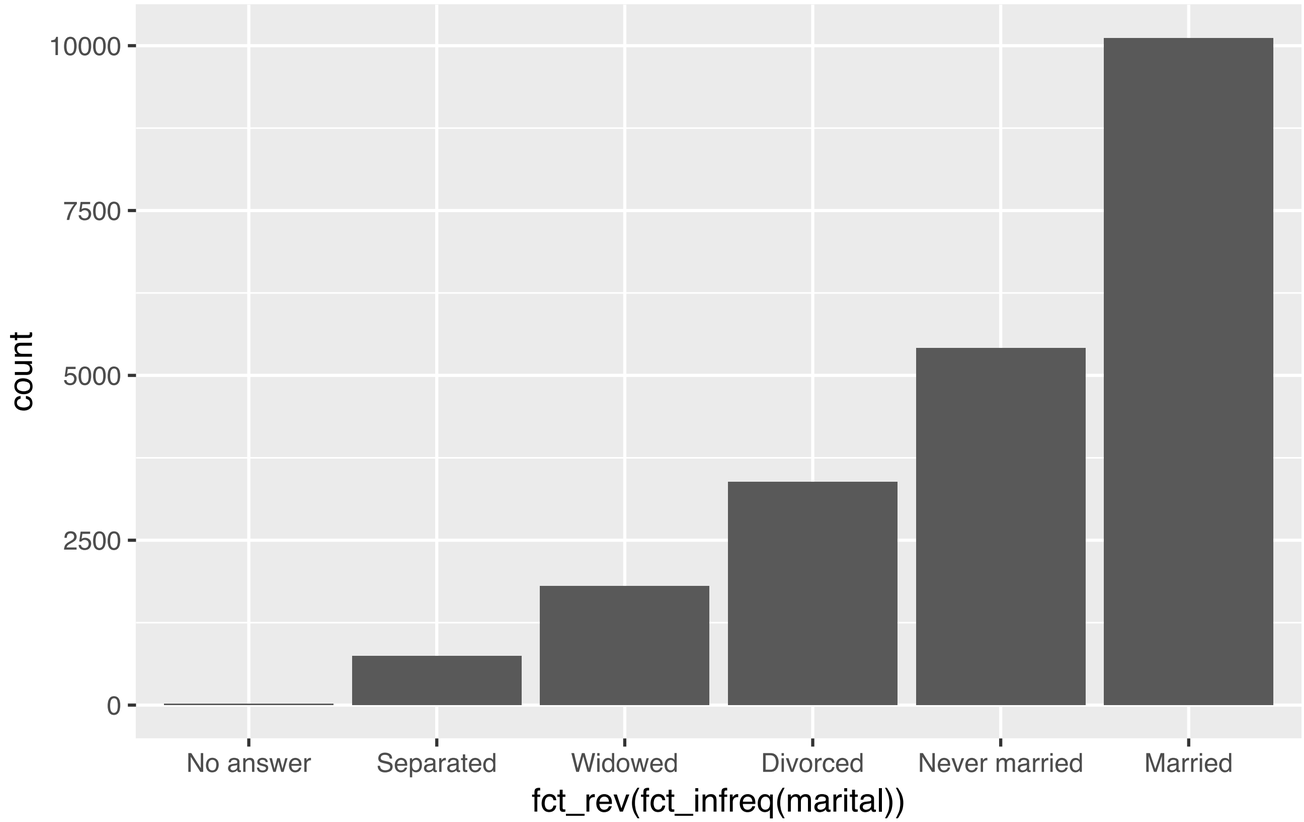

```
gss_cat |>
  ggplot(aes(marital)) + geom_bar()
```

fct_infreq




```
gss_cat |>
  ggplot(aes(fct_infreq(marital))) + geom_bar()
```

fct_rev




```
gss_cat |>
  ggplot(aes(fct_rev(fct_infreq(marital)))) + geom_bar()
```

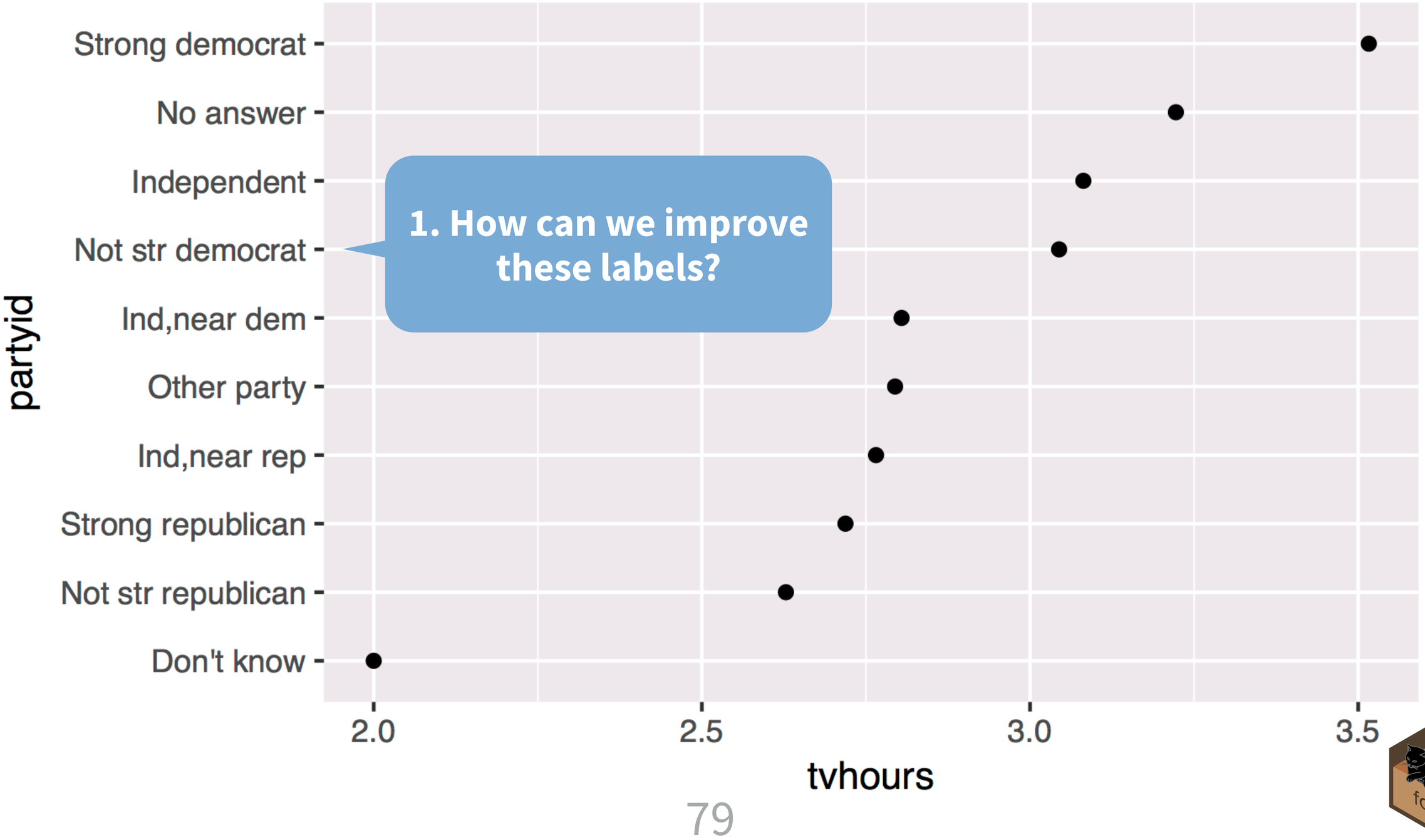
Changing level values

Your Turn 4

Do you think liberals or conservatives watch more TV?

Compute average tv hours by party ID and then plot the results.

```
gss_cat |>
    filter(!is.na(tvhours)) |>
    group_by(partyid) |>
    summarise(tvhours = mean(tvhours)) |>
    ggplot(aes(tvhours, fct_reorder(partyid, tvhours))) +
    geom_point() +
    labs(y = "partyid")
```


Strong democrat
No answer
Independent
Not str democrat
Ind,near dem
Other party
Ind,near rep
Strong republican
Not str republican
Don't know
partyid
2.0
2.5
3.0
3.5
tvhours
1. How can we improve these labels?

fct_recode()

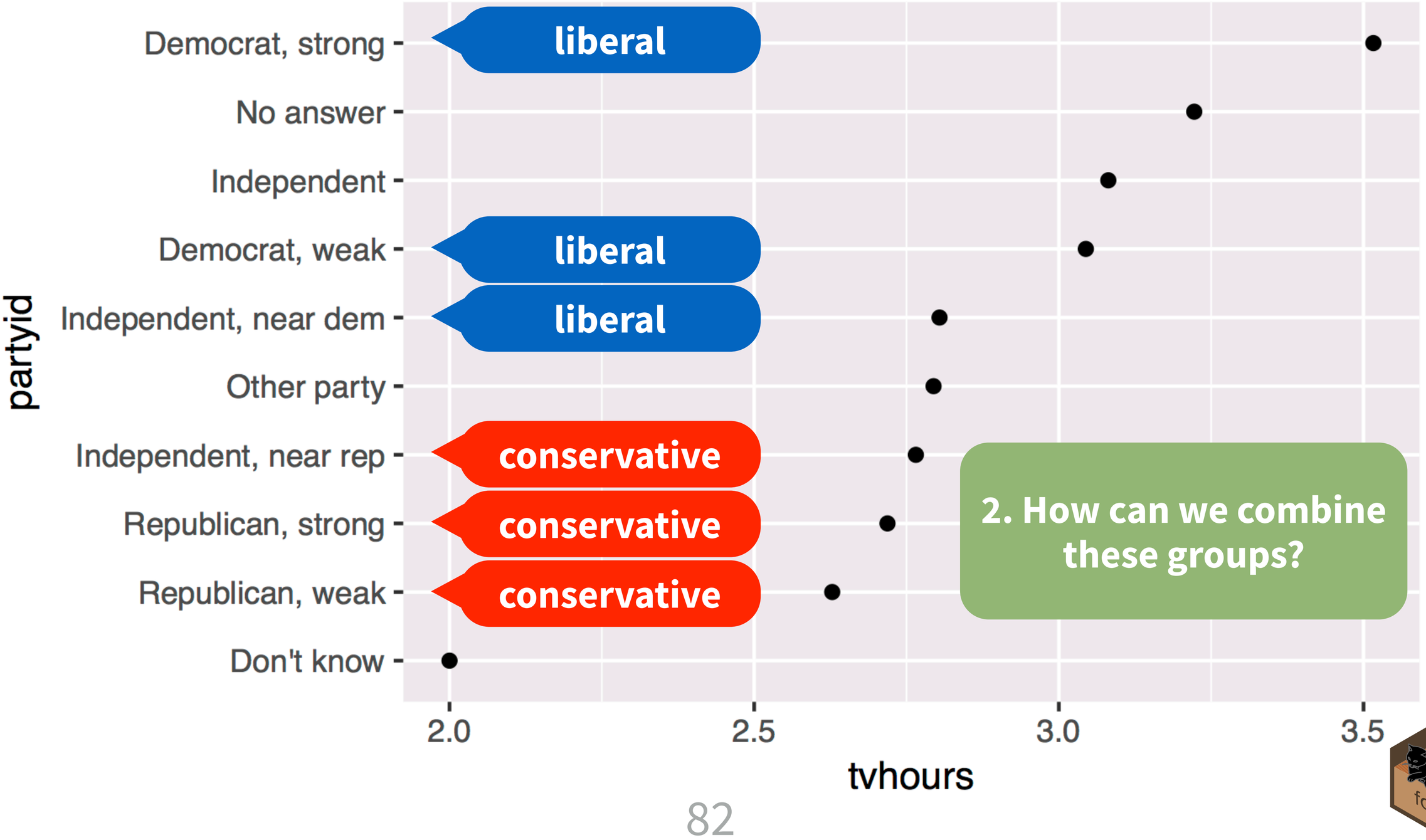
Changes values of levels

```
fct_recode(f, Independent = "Ind,near dem")
```

factor with levels

new level = old level pairs (as a named character vector)

```
gss_cat |>
   filter(!is.na(tvhours)) |>
   mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat")) %>%
   group_by(partyid) |>
   summarise(tvhours = mean(tvhours)) |>
   ggplot(aes(tvhours, fct_reorder(partyid, tvhours))) +
   geom_point() + labs(y = "partyid")
```


Democrat, strong
No answer
Independent
Democrat, weak
Independent, near dem
Other party
Independent, near rep
Republican, strong
Republican, weak
Don't know
partyid
liberal
liberal
liberal
conservative
conservative
conservative
2. How can we combine these groups?
2.0
2.5
3.0
3.5
tvhours

Collapsing levels

fct_collapse()

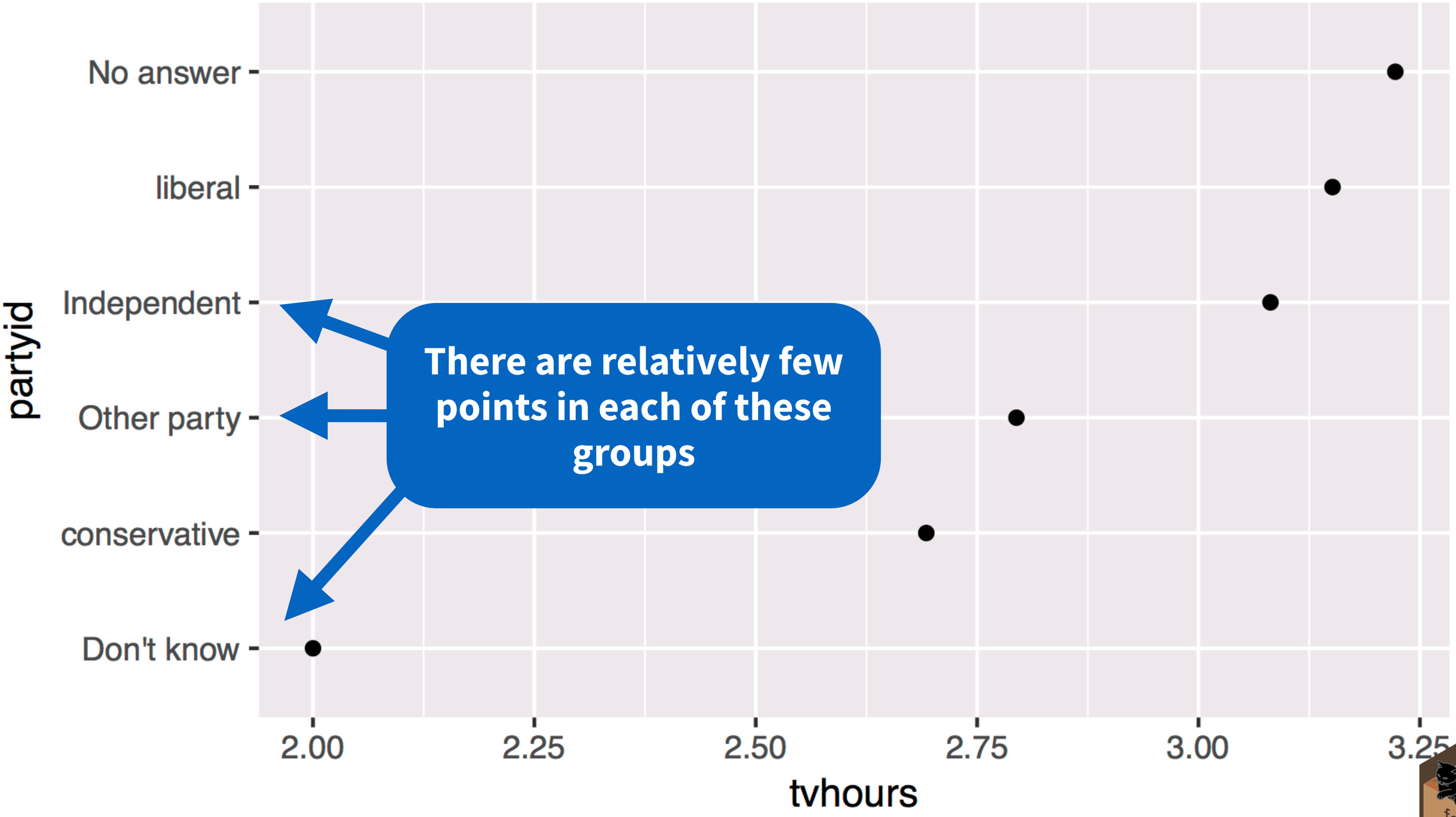
Collapses multiple levels into single levels

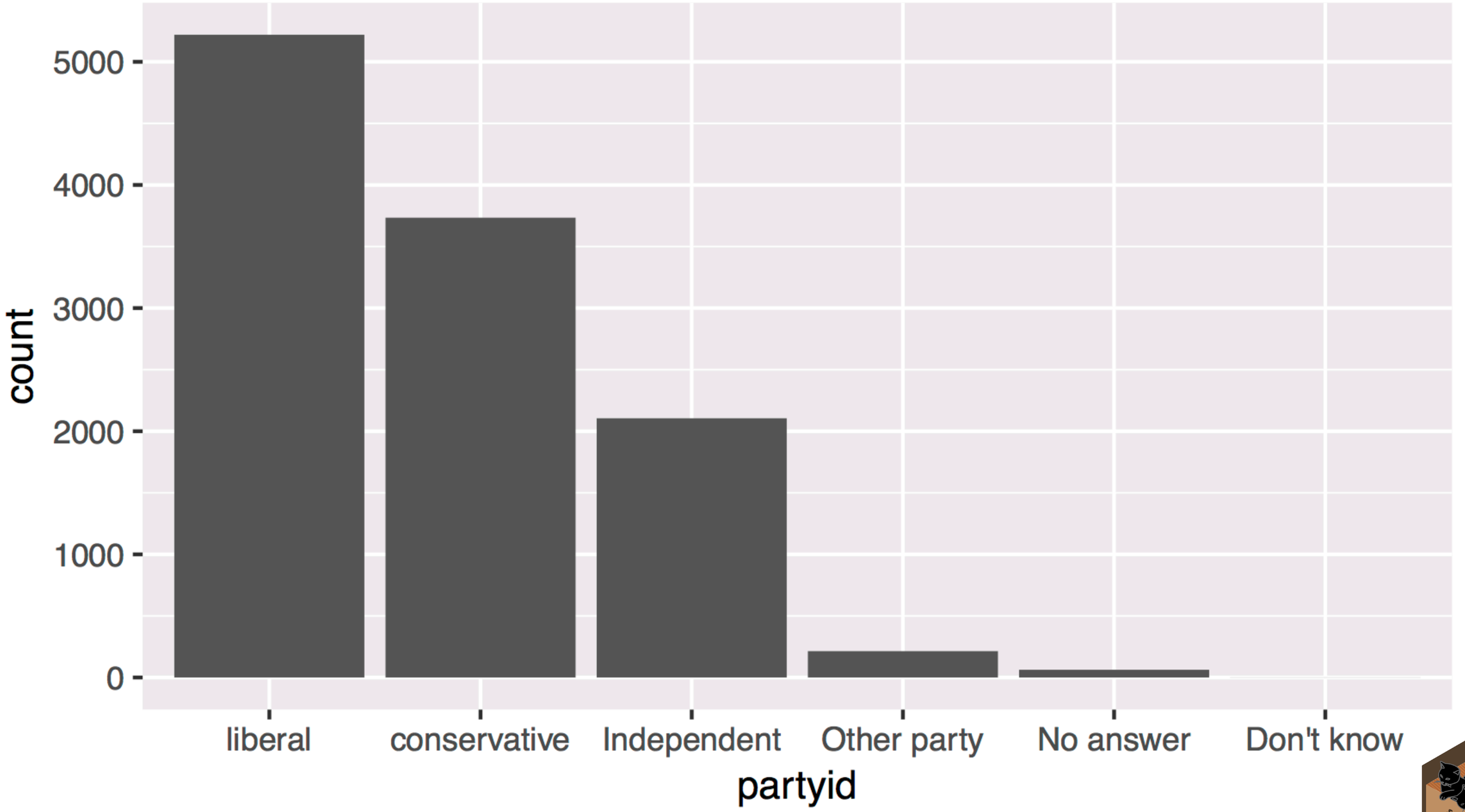
```
fct_collapse(f, Liberal = c("Democrat, strong",
                            "Democrat, weak"))
```

factor with levels

named arguments set to a character vector (levels in the vector will be collapsed to the name of the argument)

```
gss_cat |>
   filter(!is.na(tvhours)) |>
   mutate(partyid = fct_collapse(partyid,
      conservative = c("Strong republican",
                       "Not str republican",
                       "Ind,near rep"),
      liberal = c("Strong democrat",
                  "Not str democrat",
                  "Ind,near dem"))) |>
   group_by(partyid) |>
   summarise(tvhours = mean(tvhours)) |>
   ggplot(aes(tvhours, fct_reorder(partyid, tvhours))) +
   geom_point() + labs(y = "partyid")
```


No answer
liberal
Independent
Other party
conservative
Don't know
partyid
2.00
2.25
2.50
2.75
3.00
3.25
tvhours
There are relatively few points in each of these groups


5000
4000
3000
2000
1000
0
count
liberal
conservative
Independent
Other party
No answer
Don't know
partyid

fct_lump()

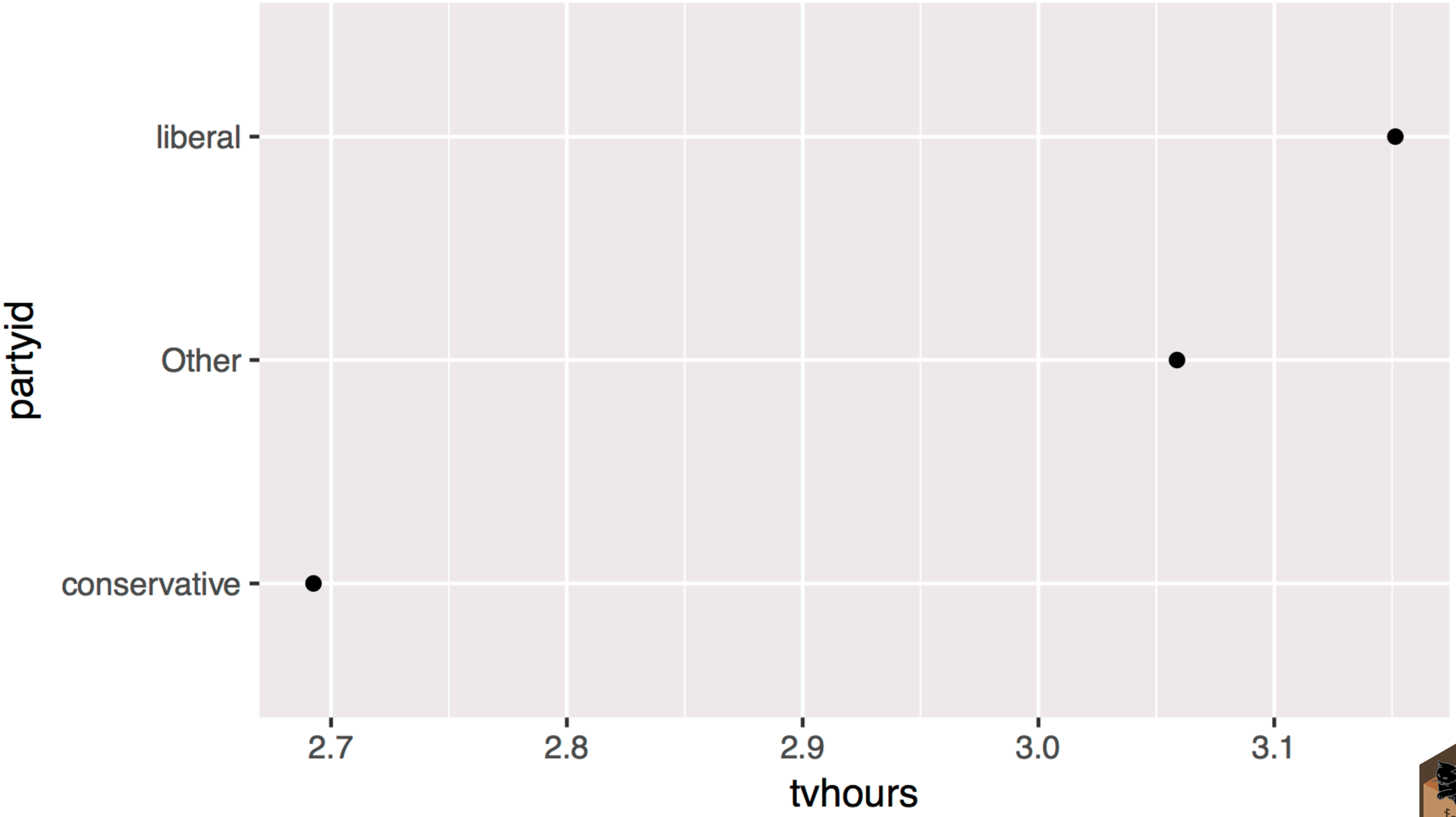
Collapses levels with fewest values into a single level. Collapses as many levels as possible such that the new level is still the smallest.

```
fct_lump(f, other_level = "Other", …)
```

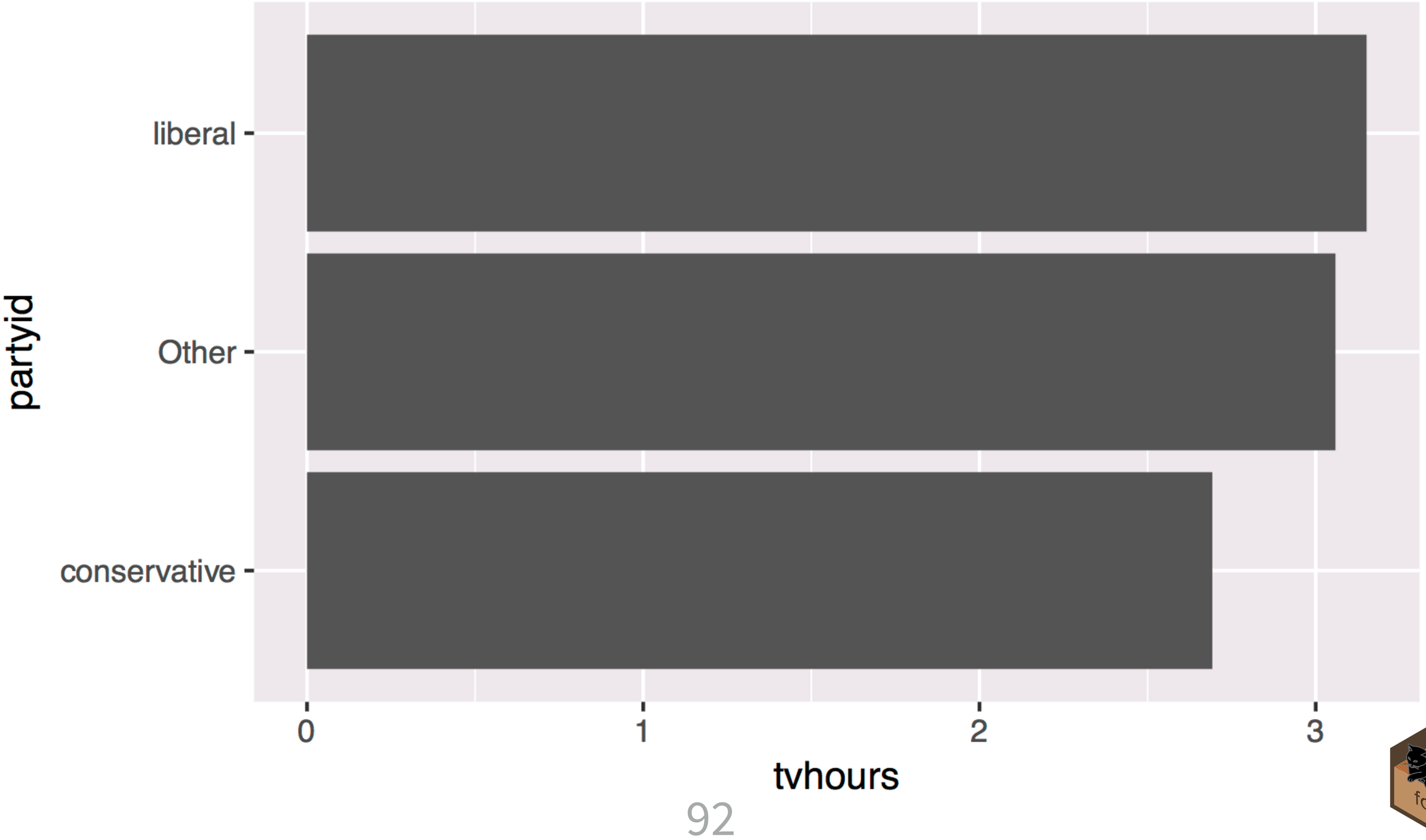
factor with levels

name of new level

```
gss_cat |>
  filter(!is.na(tvhours)) |>
  mutate(partyid = partyid |>
    fct_collapse(
      conservative = c("Strong republican",
                       "Not str republican", "Ind,near rep"),
      liberal = c("Strong democrat", "Not str democrat",
                  "Ind,near dem")) |>
    fct_lump()
  ) |>
  group_by(partyid) |>
  summarise(tvhours = mean(tvhours)) |>
  ggplot(aes(tvhours, fct_reorder(partyid, tvhours))) +
  geom_point() + labs(y = "partyid")
```


liberal
Other
conservative
partyid
2.7
2.8
2.9
3.0
3.1
tvhours

```
gss_cat |>
  filter(!is.na(tvhours)) |>
  mutate(partyid = partyid |>
    fct_collapse(
      conservative = c("Strong republican",
                       "Not str republican", "Ind,near rep"),
      liberal = c("Strong democrat", "Not str democrat",
                  "Ind,near dem")) |>
    fct_lump()
  ) |>
  group_by(partyid) |>
  summarise(tvhours = mean(tvhours)) |>
  ggplot(aes(fct_reorder(partyid, tvhours), tvhours)) +
  geom_col() + labs(x = "partyid") + coord_flip()
```


liberal
Other
conservative
partyid
0
1
2
3
tvhours

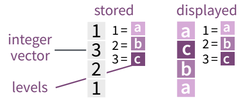
factors

Factors with forcats : : CHEAT SHEET

The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors

R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the values associated with them.



Create a factor with factor()

factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA) Convert a vector to a factor. Also **as_factor**.
f <- factor(c("a", "c", "b", "a"), levels = c("a", "b", "c"))

Return its levels with levels()

levels(x) Return/set the levels of a factor. *levels(f); levels(f) <- c("x","y","z")*

Use unclass() to see its structure

Inspect Factors

fct_count(f, sort = FALSE) Count the number of values with each level. *fct_count(f)*

fct_unique(f) Return the unique values, removing duplicates. *fct_unique(f)*

Combine Factors

fct_c(…) Combine factors with different levels.
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)

fct_unify(fs, levels = lvls_union(fs)) Standardize levels across a list of factors. *fct_unify(list(f2, f1))*

Change the order of levels

fct_relevel(.f, ..., after = 0L) Manually reorder factor levels. *fct_relevel(f, c("b", "c", "a"))*

fct_infreq(f, ordered = NA) Reorder levels by the frequency in which they appear in the data (highest frequency first).
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)

fct_inorder(f, ordered = NA) Reorder levels by order in which they appear in the data. *fct_inorder(f2)*

fct_rev(f) Reverse level order.
f4 <- factor(c("a","b","c"))
fct_rev(f4)

fct_shift(f) Shift levels to left or right, wrapping around end. *fct_shift(f4)*

fct_shuffle(f, n = 1L) Randomly permute order of factor levels. *fct_shuffle(f4)*

fct_reorder(.f, .x, .fun=median, ..., .desc = FALSE) Reorder levels by their relationship with another variable.
boxplot(data = iris, Sepal.Width ~ fct_reorder(Species, Sepal.Width))

fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE) Reorder levels by their final values when plotted with two other variables.
ggplot(data = iris, aes(Sepal.Width, Sepal.Length, color = fct_reorder2(Species, Sepal.Width, Sepal.Length))) + geom_smooth()

Change the value of levels

fct_recode(.f, ...) Manually change levels. Also **fct_relabel** which obeys purrr::map syntax to apply a function or expression to each level.
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))

fct_anon(f, prefix = "")) Anonymize levels with random integers. *fct_anon(f)*

fct_collapse(.f, ...) Collapse levels into manually defined groups.
fct_collapse(f, x = c("a", "b"))

fct_lump(f, n, prop, w = NULL, other_level = "Other", ties.method = c("min", "average", "first", "last", "random", "max")) Lump together least/most common levels into a single level. Also **fct_lump_min**.
fct_lump(f, n = 1)

fct_other(f, keep, drop, other_level = "Other") Replace levels with "other."
fct_other(f, keep = c("a", "b"))

Add or drop levels

fct_drop(f, only) Drop unused levels.
f5 <- factor(c("a","b"),c("a","b","x"))
f6 <- fct_drop(f5)

fct_expand(f, …) Add levels to a factor. *fct_expand(f6, "x")*

fct_explicit_na(f, na_level="(Missing)") Assigns a level to NAs to ensure they appear in plots, etc.
fct_explicit_na(factor(c("a", "b", NA)))

Date times

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?

Most useful skills

1. Creating dates/times (i.e. *parsing*)
2. Access and change parts of a date
3. Deal with time zones
4. Do math with instants and time spans

Warm Up

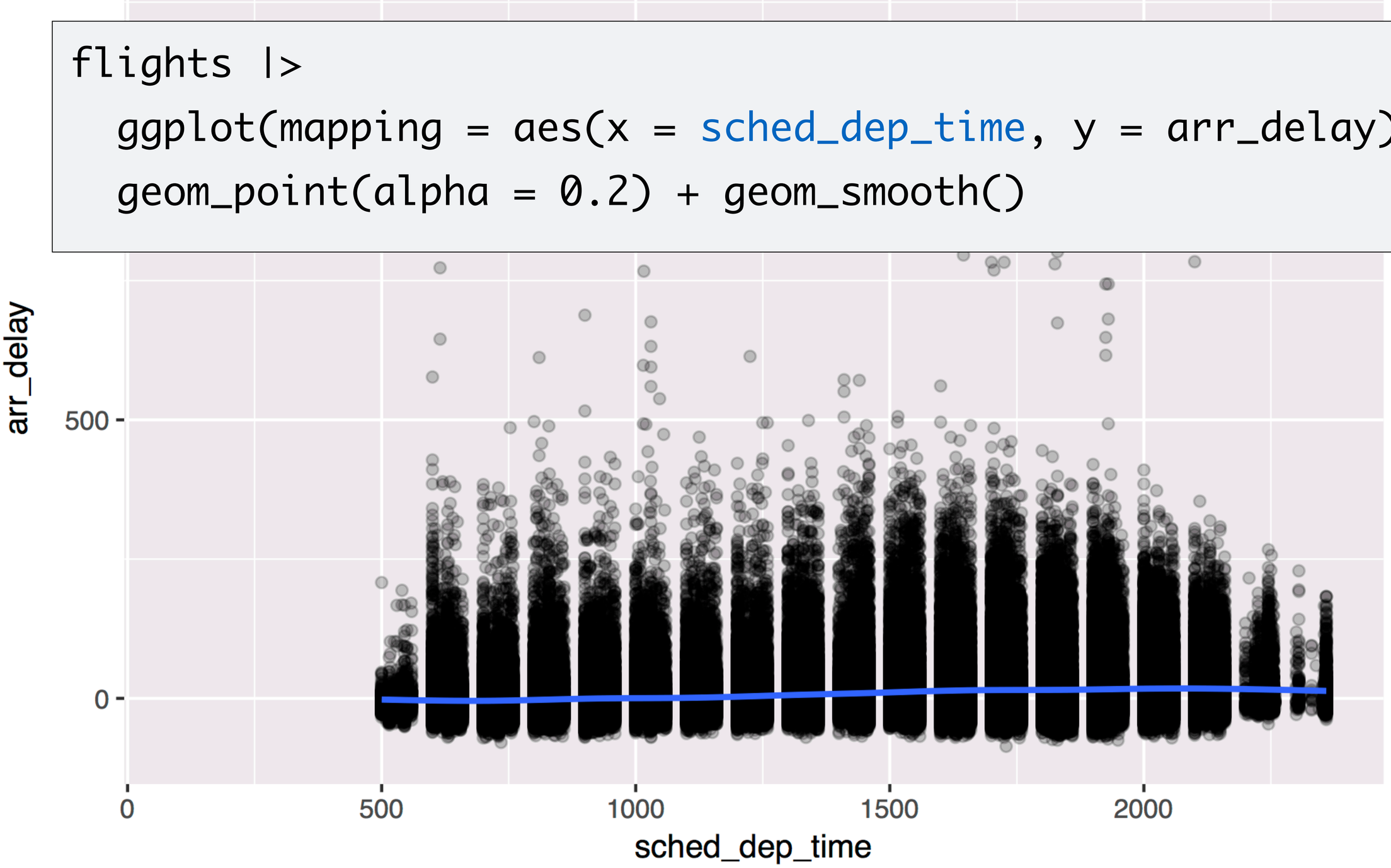
- What is the best time of day to fly?
- What is the best day of the week to fly?

```
flights |> select(c(1, 2, 3, 17, 18, 5, 19))
```

year <int>	month <int>	day <int>	hour <dbl>	minute <dbl>	sched_dep_time <int>	time_hour <S3: POSIXct>
2013	1	1	5	15	515	2013-01-01 05:00:00
2013	1	1	5	29	529	2013-01-01 05:00:00
2013	1	1	5	40	540	2013-01-01 05:00:00
2013	1	1	5	45	545	2013-01-01 05:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	5	58	558	2013-01-01 05:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00

1–10 of 336,776 rows Previous 1 2 3 4 5 6 ... 100 Ne:

```
flights |>
  ggplot(mapping = aes(x = sched_dep_time, y = arr_delay)) +
  geom_point(alpha = 0.2) + geom_smooth()
```



```
flights |> select(c(1, 2, 3, 17, 18, 5, 19))
```

year <int>	month <int>	day <int>	hour <dbl>	minute <dbl>	sched_dep_time <int>	time_hour <S3: POSIXct>
2013	1	1	5	15	515	2013-01-01 05:00:00
2013	1	1	5	29	529	2013-01-01 05:00:00
2013	1	1	5	40	540	2013-01-01 05:00:00
2013	1	1	5	45	545	2013-01-01 05:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	5	58	558	2013-01-01 05:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00
2013	1	1	6	0	600	2013-01-01 06:00:00

1–10 of 336,776 rows Previous 1 2 3 4 5 6 ... 100 Ne:

Creating dates and times

hms

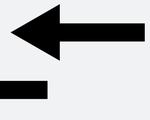
A class for representing just clock times.

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

```
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("forcats")
install.packages("lubridate")
install.packages("hms")
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")
library("stringr")
library("forcats")
```

hms()

2017-01-01 12:34:56

```
hms(seconds, minutes, hours, days)
```

numbers of each unit to add to the time

* on a typical day

hms

2017-01-01 **12:34:56**

Stored as the number of seconds since 00:00:00.*

```
hms(seconds = 56, min = 34, hour = 12)
## 12:34:56

unclass(hms(56, 34, 12))
## 45296
```

* on a typical day

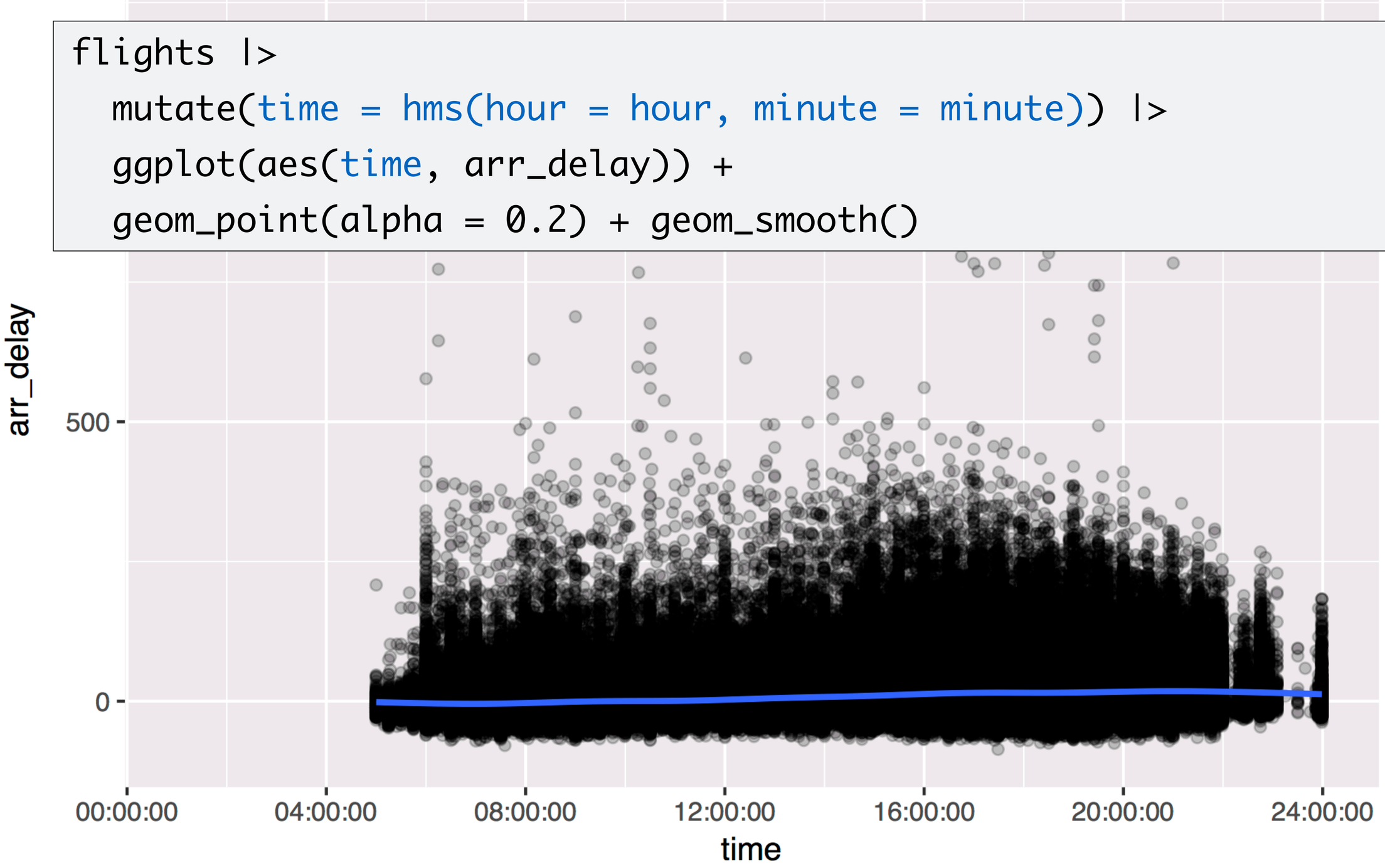
Your Turn 5

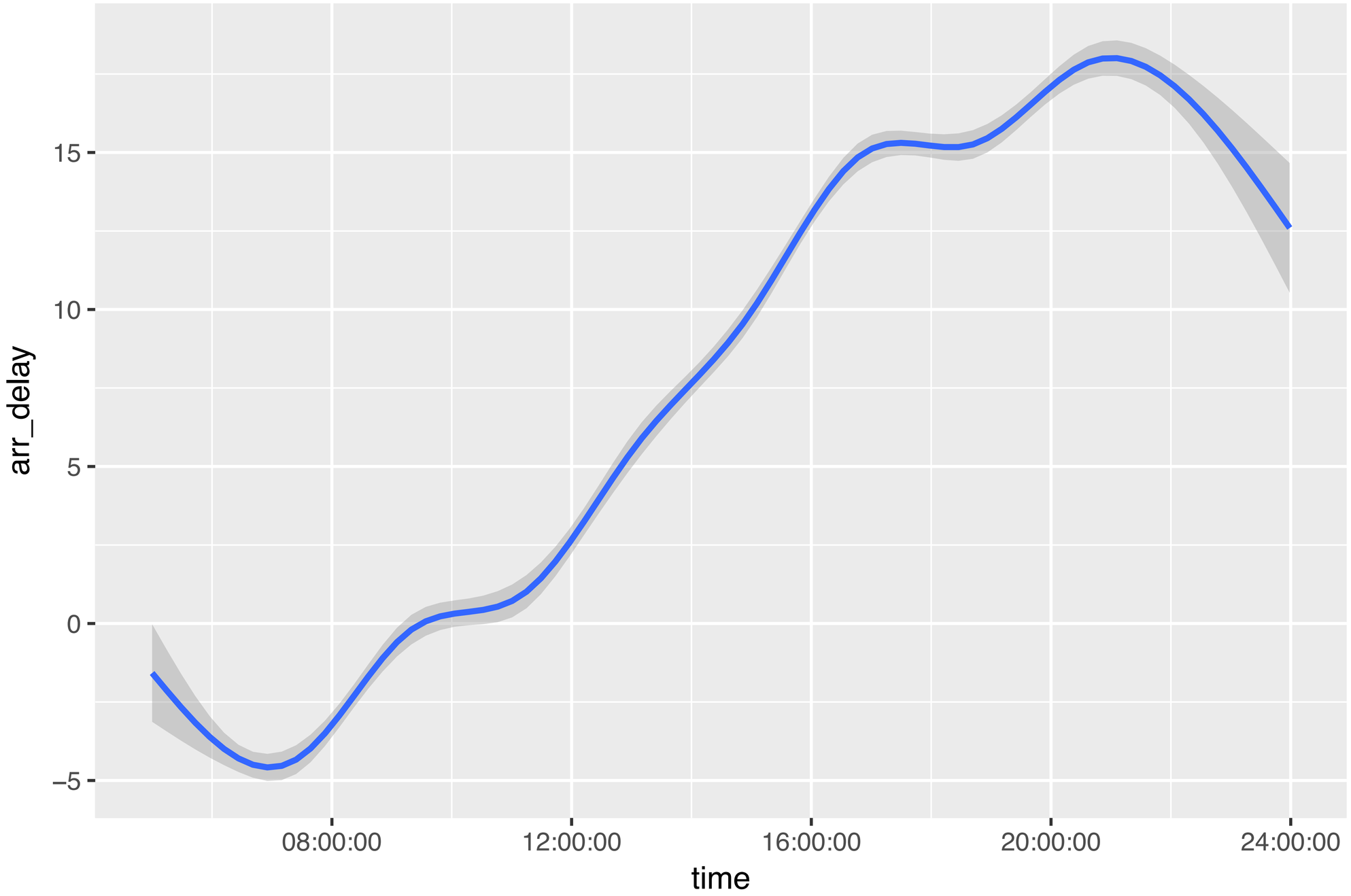
What is the best time of day to fly?

Use the **hour** and **minute** variables in flights to make a new variable that shows the **time** of each flight as an hms.

Then use a smooth line to plot the relationship between time of day and **arr_delay**.

```
flights |>
  mutate(time = hms(hour = hour, minute = minute)) |>
  ggplot(aes(time, arr_delay)) +
  geom_point(alpha = 0.2) + geom_smooth()
```




arr_delay
15
10
5
0
−5
08:00:00
12:00:00
16:00:00
20:00:00
24:00:00
time


hms

What is the best day of the week to fly?

Your Turn 6

Look at the code skeleton for Your Turn 7. Discuss with your neighbor:

- What does each line do?
- What will the missing parts need to do?

lubridate

Functions for working with dates and time spans

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


ymd() family

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

To parse strings as dates, use the function whose name is y, m, d, h, m, s in the correct order.

```
ymd("2012/01/11")
mdy("January 11, 2012")
ymd_hms("2012-01-11  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("2019-01-11")
year(date)
## 2019
```

Setting components

Use the same function to set components

```
date
## "2019-01-11"


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("2019-01-11"))
## 6
wday(ymd("2019-01-11"), label = TRUE)
## [1] Fri
## 7 Levels: Sun < Mon < Tues < Wed < Thurs < ... < Sat
wday(ymd("2019-01-11"), label = TRUE, abbr = FALSE)
## [1] Friday
## 7 Levels: Sunday < Monday < Tuesday < ... < Saturday
```

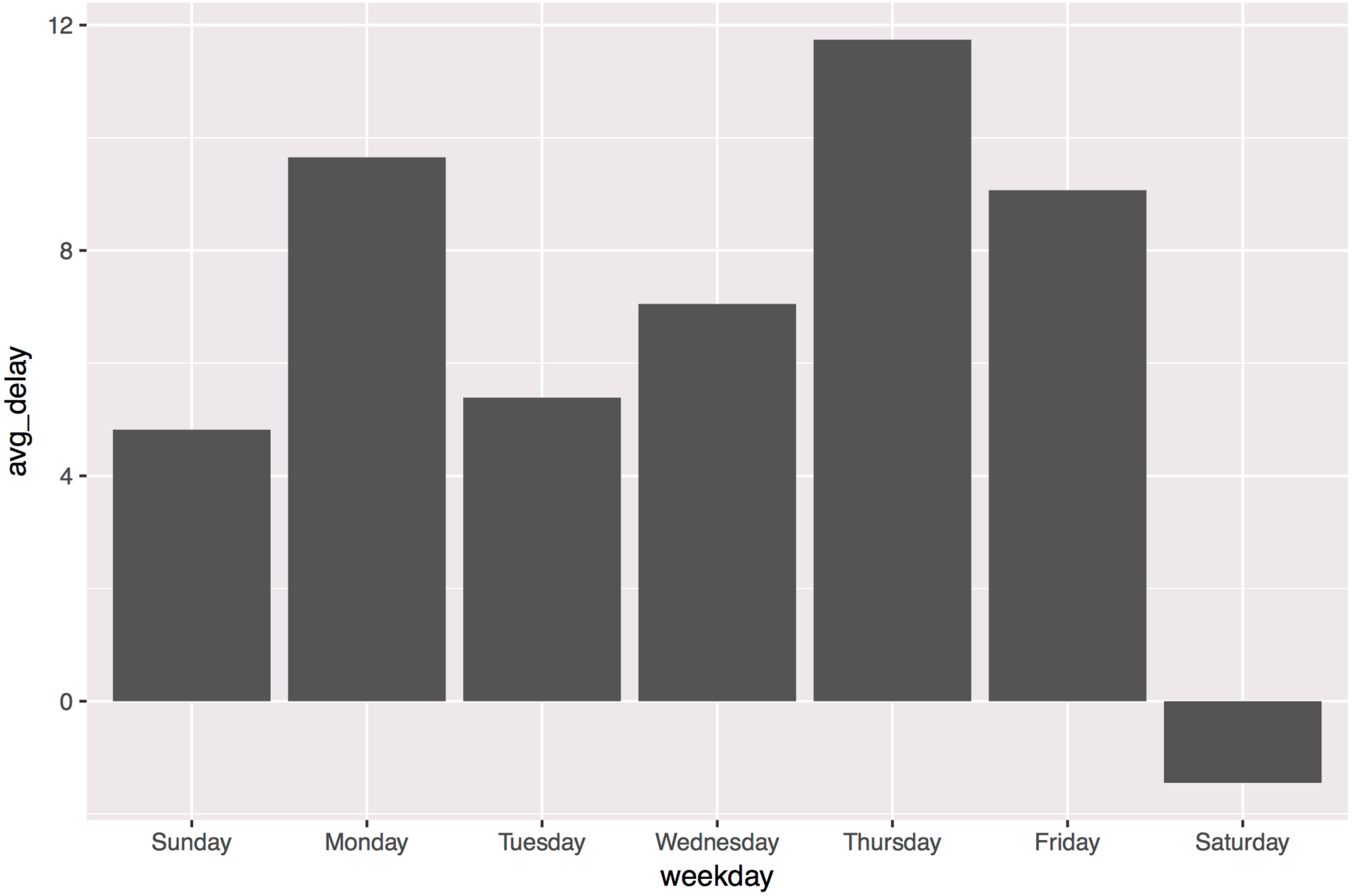
Your Turn 7

Fill in the blank to:

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

Plot the average arrival delay by day as a column chart (bar chart).

```
flights |>
  mutate(weekday = wday(time_hour, label = TRUE, abbr = FALSE)) |>
  group_by(weekday) |>
  filter(!is.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

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

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

Same name as hms() in hms

hms::hms()

package name

function name

* on a typical day

hms::hms()

lubridate::hms()

* on a typical day

hms()

```
hms::hms(seconds = 3, hours = 5)
```

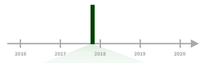
Use the hms() function in the hms package

* on a typical day

Dates and Times

Dates and times with lubridate : : CHEAT SHEET

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**(), **hm**() and **ms**(), which return periods.* *hms::hms(sec = 0, min= 1, hours = 2)*

`2017.5` **date_decimal**(decimal, tz = "UTC") *date_decimal(2017.5)*

now(tzone = "") Current time in tz (defaults to system tz). *now()*

today(tzone = "") Current date in a tz (defaults to system tz). *today()*

fast_strptime() Faster strptime. *fast_strptime('9/1/01', '%y/%m/%d')*

parse_date_time() Easier strptime. *parse_date_time("9/1/01", "ymd")*

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)*
isoweek() ISO 8601 week.
epiweek() Epidemiological week.

quarter(x, with_year = FALSE) Quarter. *quarter(dt)*

semester(x, with_year = FALSE) Semester. *semester(dt)*

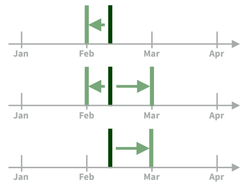
am(x) Is it in the am? *am(dt)*
pm(x) Is it in the pm? *pm(dt)*

dst(x) Is it daylight savings? *dst(d)*

leap_year(x) Is it a leap year? *leap_year(d)*

update(object, ..., simple = FALSE) *update(dt, mday = 2, hour = 1)*

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"

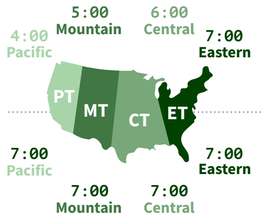
Tip: use a date with day > 12

Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns *one* time zone per vector.

Use the **UTC** time zone to avoid Daylight Savings.

OlsonNames() Returns a list of valid time zone names. *OlsonNames()*



with_tz(time, tzone = "") Get the **same date-time** in a new time zone (a new clock time). *with_tz(dt, "US/Pacific")*

force_tz(time, tzone = "") Get the **same clock time** in a new time zone (a new date-time). *force_tz(dt, "US/Pacific")*

* on a typical day

Data types with