Factors and Dates

Daniel Anderson

Agenda

- Basics of factors
    - creating/modifying
    - when you do/do not want factors
- Basics of Dates
    - Specifically, we'll focus date calculations

Disclaimer

- We're obviously not going to cover all there is to know about factors and dates in one smashed-together two-hour lecture.
- If we had more time, we'd spend a week on each. Instead you get one lecture.

Factors

Notice a difference?

```
library(tidyverse)
tibble(lets = letters[1:3])
```

```
## # A tibble: 3 x 1
##    lets
##   <chr>
## 1     a
## 2     b
## 3     c
```

```
data.frame(lets = letters[1:3])
```

```
##   lets
## 1    a
## 2    b
## 3    c
```

What about now?

```
str(tibble(lets = letters[1:3]))
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  1 variable:
##  $ lets: chr  "a" "b" "c"
```

```
str(data.frame(lets = letters[1:3]))
```

```
## 'data.frame':    3 obs. of  1 variable:
##  $ lets: Factor w/ 3 levels "a","b","c": 1 2 3
```

Why?

- Primarily historical reasons
  - Factors used to be much easier to work with
  - If you want to use the data for modeling, factors make more sense
    - R is increasingly used for all sorts of things besides analysis, so it makes less sense for everything to be a factor

What to do?

- Turn it off globally, but that's dangerous

```
options(default.stringsAsFactors = FALSE)
```

- Turn it off in only the functions it affects, but you might forget

```
str(data.frame(lets = letters[1:3], stringsAsFactors = FALSE))
```

```
## 'data.frame':    3 obs. of  1 variable:
##  $ lets: chr  "a" "b" "c"
```

- Use `rio::import` or *readr* (e.g., `readr::read_csv`), which will default to reading strings in as strings

Creating factors

- Imagine you have a vector of months

```
months <- c("Dec", "Apr", "Jan", "Mar")
```

- We could store this as a string, but there are issues with this.
  - There are only 12 possible months
    - factors will help us weed out values that don't conform to our predefined *levels*, which helps safeguard against typos, etc.
  - You can't sort this vector in a meaningful way (it defaults to alphabetic)

```
sort(months)
```

```
## [1] "Apr" "Dec" "Jan" "Mar"
```

Define it as a factor

```
month_levels <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
                  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")

months <- factor(months, levels = month_levels)
months
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

- Now, we can sort

```
sort(months)
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Also provides a safety net of sorts

```
months[5] <- "Jam"
```

```
## Warning in `[<-.factor`(`*tmp*`, 5, value = "Jam"): invalid factor level,
## NA generated
```

```
months
```

```
## [1] Dec  Apr  Jan  Mar  <NA>
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

What if we don't specify the levels?

- If you define a factor without specifying the levels, it will assign them alphabetically

```
mnths <- c("Dec", "Apr", "Jan", "Mar")
factor(mnths)
```

```
## [1] Dec Apr Jan Mar
## Levels: Apr Dec Jan Mar
```

- If you instead want them in the order they appeared in the data, use `unique` when specifying the levels (Why is `unique()` necessary? What's it doing?)

```
factor(mnths, levels = unique(mnths))
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

Accessing and modifying levels

Use the **levels** function

- To view the levels

```
levels(months)
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

- To modify the levels

```
levels(months) <- 1:12
months
```

```
## [1] 12   4    1    3    <NA>
## Levels: 1 2 3 4 5 6 7 8 9 10 11 12
```

If you need to, be specific

```
months <- factor(months, levels = 1:12, labels = month_levels)
months
```

```
## [1] Dec  Apr  Jan  Mar  <NA>
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

New package

- When working with factors, we can use the *forcats* package
  - *for* **cat** egorical variables *s*
  - anagram for factors
- Part of the tidyverse
  - Should be installed for you already, but won't load with `library(tidyverse)`

Changes factors back to the order they appeared

```r
c("Dec", "Apr", "Jan", "Mar") %>%
    factor()
```

```
## [1] Dec Apr Jan Mar
## Levels: Apr Dec Jan Mar
```

```r
c("Dec", "Apr", "Jan", "Mar") %>%
    factor(levels = c("Jan", "Mar", "Apr", "Dec"))
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Mar Apr Dec
```

see next slide

```r
library(forcats)
c("Dec", "Apr", "Jan", "Mar") %>%
    factor(levels = c("Jan", "Mar", "Apr", "Dec")) %>%
    fct_inorder()
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

Or order by frequency

```
c("b", "b", "c", "a", "a", "a") %>%
    fct_infreq()
```

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

- This can be particularly useful for plotting

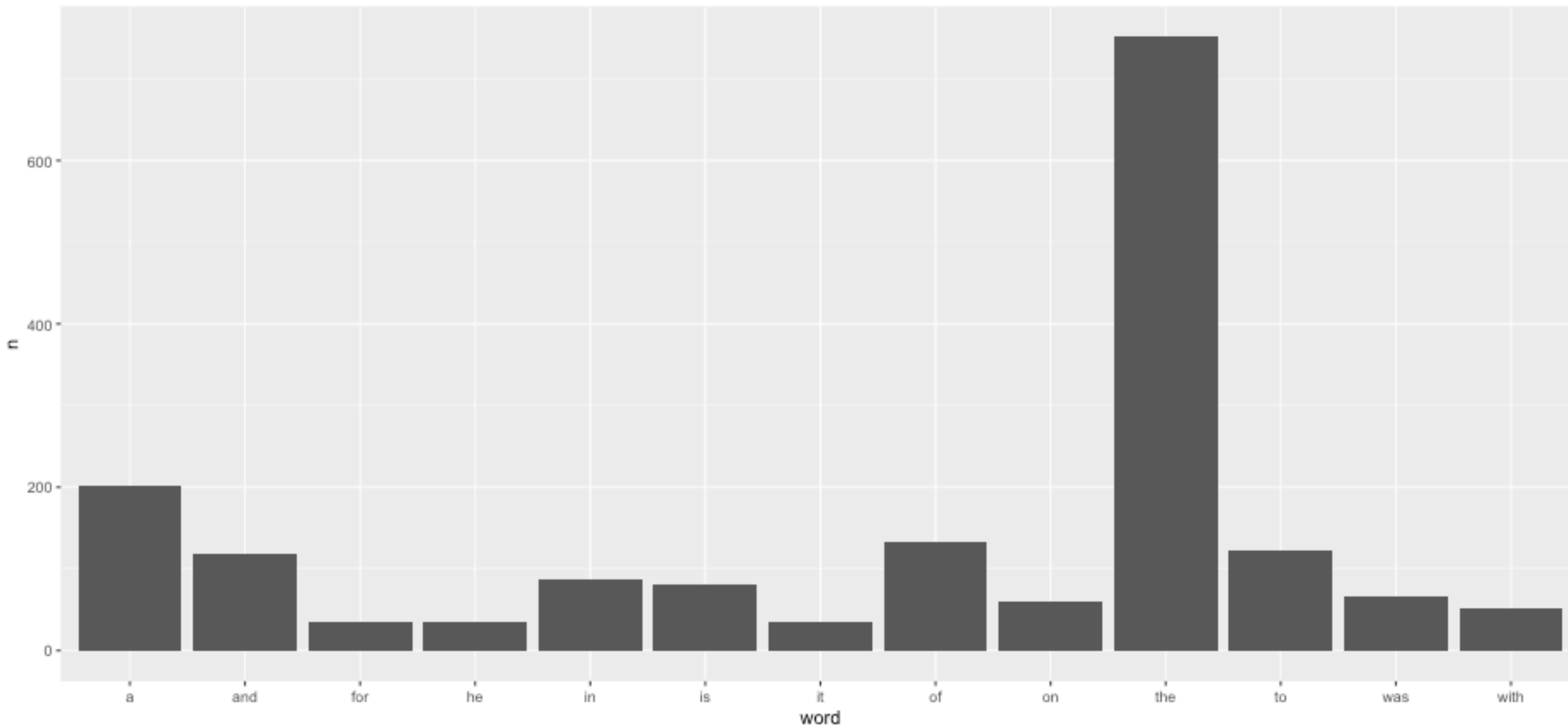
words example

```
data(sentences, package = "stringr")
sentences <- tibble(sent_num = seq_along(sentences), sentence = sentences)
library(tidytext)
words_freq <- sentences %>%
    unnest_tokens(word, sentence) %>%
    count(word) %>%
    filter(n > 30)
words_freq
```

```
## # A tibble: 13 x 2
##      word     n
##     <chr> <int>
##  1      a   202
##  2    and   118
##  3    for    35
##  4     he    34
##  5     in    87
##  6     is    81
##  7     it    36
##  8     of   132
##  9     on    60
```

Try to plot frequencies

```
ggplot(words_freq, aes(word, n)) +
    geom_col()
```



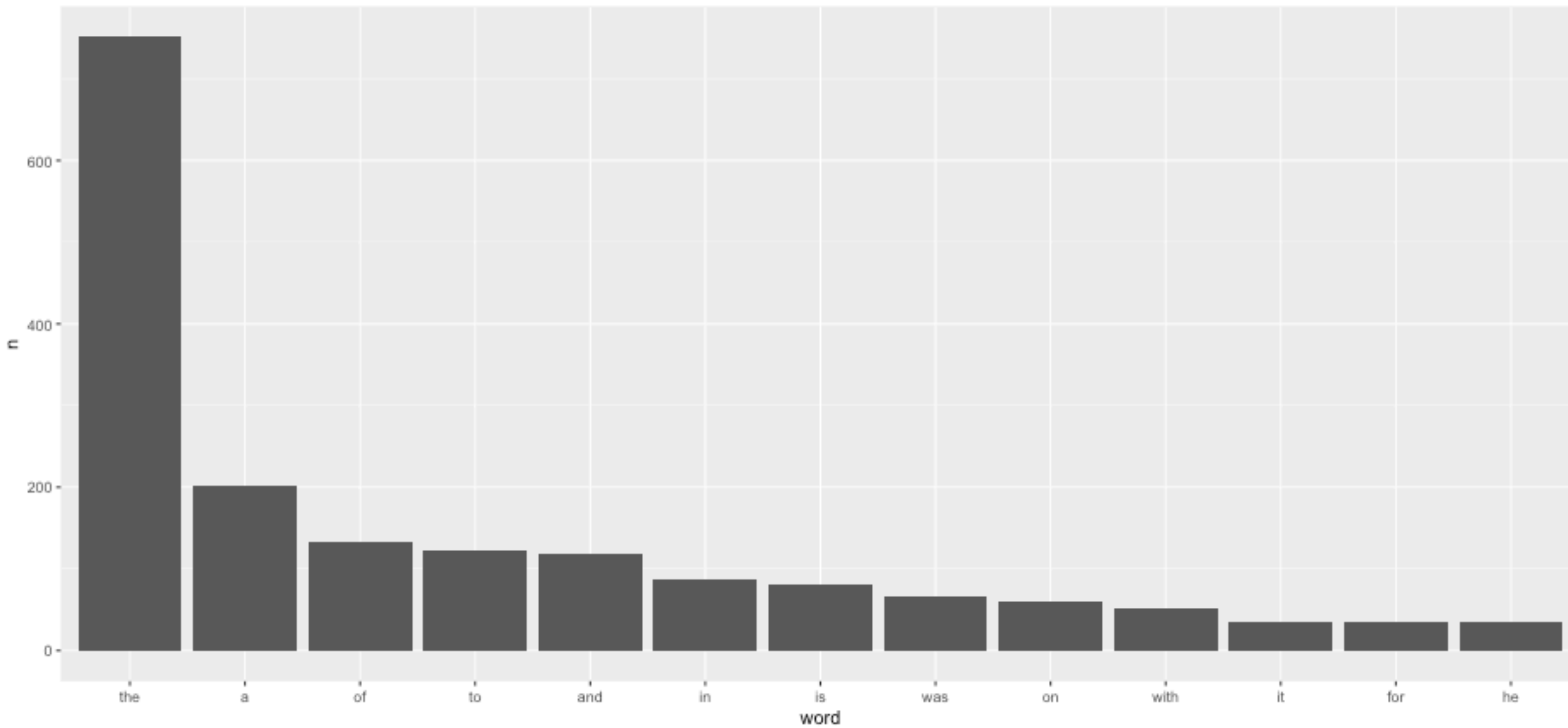
Reorder according to frequency

```
words_freq2 <- sentences %>%
    unnest_tokens(word, sentence) %>%
    mutate(word = fct_infreq(word)) %>%
    count(word) %>%
    filter(n > 30)
words_freq2
```

```
## # A tibble: 13 x 2
##      word     n
##    <fctr> <int>
##  1    the   751
##  2      a   202
##  3     of   132
##  4     to   123
##  5    and   118
##  6     in    87
##  7     is    81
##  8    was    66
##  9     on    60
## 10   with    51
## 11     it    36
```

Reproduce plot

```
ggplot(words_freq2, aes(word, n)) +
    geom_col()
```



Looking at the levels

```
levels(factor(words_freq$word))
```

```
##  [1] "a"    "and"  "for"  "he"   "in"   "is"   "it"   "of"   "on"   "the"
## [11] "to"   "was"  "with"
```

```
levels(words_freq2$word)
```

```
##    [1] "the"         "a"           "of"          "to"          "and"
##    [6] "in"          "is"          "was"         "on"          "with"
##   [11] "it"          "for"         "he"          "are"         "from"
##   [16] "will"        "his"         "we"          "at"          "but"
##   [21] "were"        "into"        "they"        "you"         "your"
##   [26] "that"        "when"        "this"        "by"          "be"
##   [31] "old"         "than"        "as"          "high"        "out"
##   [36] "red"         "there"       "these"       "down"        "fine"
##   [41] "green"       "hot"         "new"         "she"         "small"
##   [46] "strong"      "up"          "used"        "wall"        "before"
##   [51] "good"        "hard"        "her"         "makes"       "round"
##   [56] "thin"        "two"         "water"       "way"         "young"
##   [61] "best"        "blue"        "both"        "bright"      "dull"
##   [66] "each"        "gold"        "him"         "kept"        "last"
```

When do we really want factors?

Generally two reasons to declare a factor

- Only finite number of categories
    - Treatment/control
    - Income categories
    - Performance levels
    - etc.
- Use in modeling

GSS

General Social Survey

- We dealt with some of these data for a homework.
- Unbeknownst to me, Hadley also included a sample in the *forcats* dataset

```
gss_cat
```

```
## # A tibble: 21,483 x 9
##     year       marital   age  race        rincome            partyid
##    <int>        <fctr> <int> <fctr>        <fctr>             <fctr>
##  1  2000 Never married    26  White  $8000 to 9999       Ind,near rep
##  2  2000      Divorced    48  White  $8000 to 9999 Not str republican
##  3  2000       Widowed    67  White Not applicable        Independent
##  4  2000 Never married    39  White Not applicable       Ind,near rep
##  5  2000      Divorced    25  White Not applicable   Not str democrat
##  6  2000       Married    25  White $20000 - 24999    Strong democrat
##  7  2000 Never married    36  White $25000 or more Not str republican
##  8  2000      Divorced    44  White  $7000 to 7999       Ind,near dem
##  9  2000       Married    44  White $25000 or more   Not str democrat
## 10  2000       Married    47  White $25000 or more  Strong republican
## # ... with 21,473 more rows, and 3 more variables: relig <fctr>,
```

Investigate factors

Tidyverse gives you convenient ways to evaluate factors

- Use `count` - no need to use `group_by`
- Use `geom_bar` or `geom_col` with *ggplot*

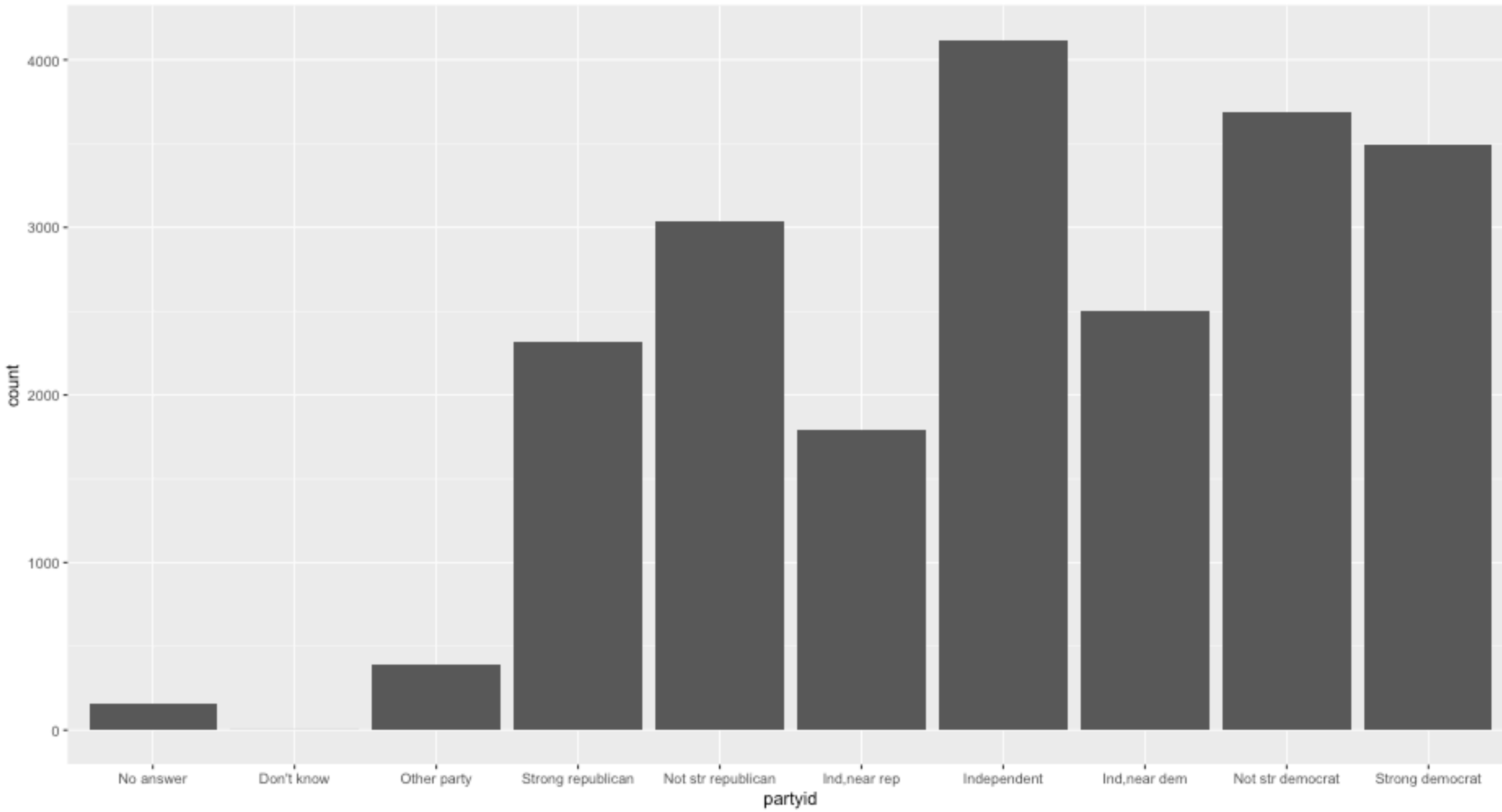
```r
gss_cat %>%
    count(partyid)
```

```
## # A tibble: 10 x 2
##               partyid     n
##                <fctr> <int>
## 1           No answer   154
## 2          Don't know     1
## 3         Other party   393
## 4   Strong republican  2314
## 5  Not str republican  3032
## 6       Ind,near rep   1791
## 7        Independent   4119
## 8       Ind,near dem   2499
## 9    Not str democrat  3690
## 10    Strong democrat  3490
```

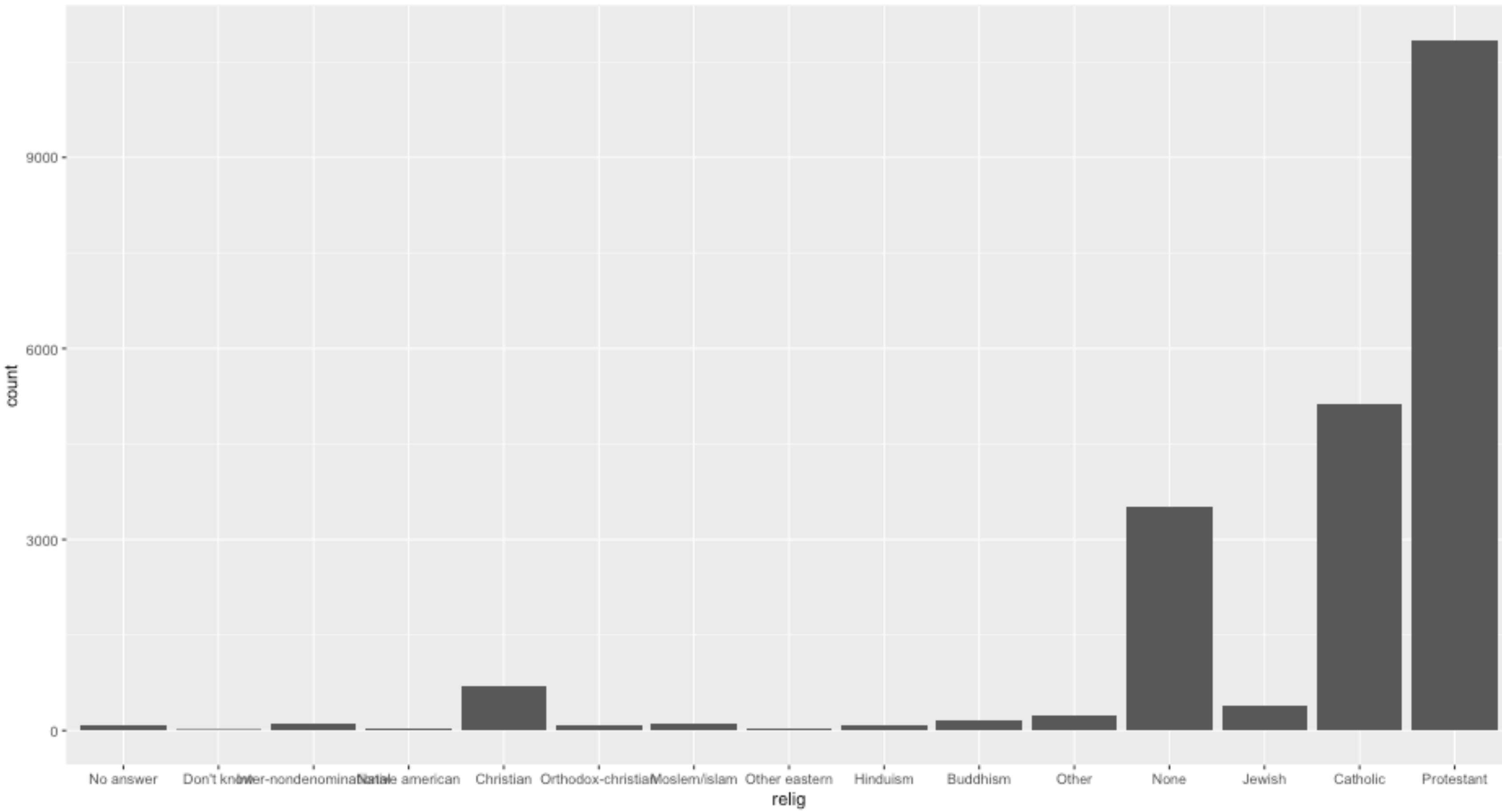
```r
gss_cat %>%
    count(relig)
```

```
## # A tibble: 15 x 2
##                       relig     n
##                      <fctr> <int>
## 1                 No answer    93
## 2                Don't know    15
## 3   Inter-nondenominational   109
## 4           Native american    23
## 5                 Christian   689
## 6        Orthodox-christian    95
## 7               Moslem/islam   104
## 8            Other eastern     32
## 9                  Hinduism    71
## 10                 Buddhism   147
## 11                    Other   224
## 12                     None  3523
## 13                   Jewish   388
## 14                 Catholic  5124
## 15              Protestant 10846
```

```r
ggplot(gss_cat, aes(partyid)) +
    geom_bar()
```

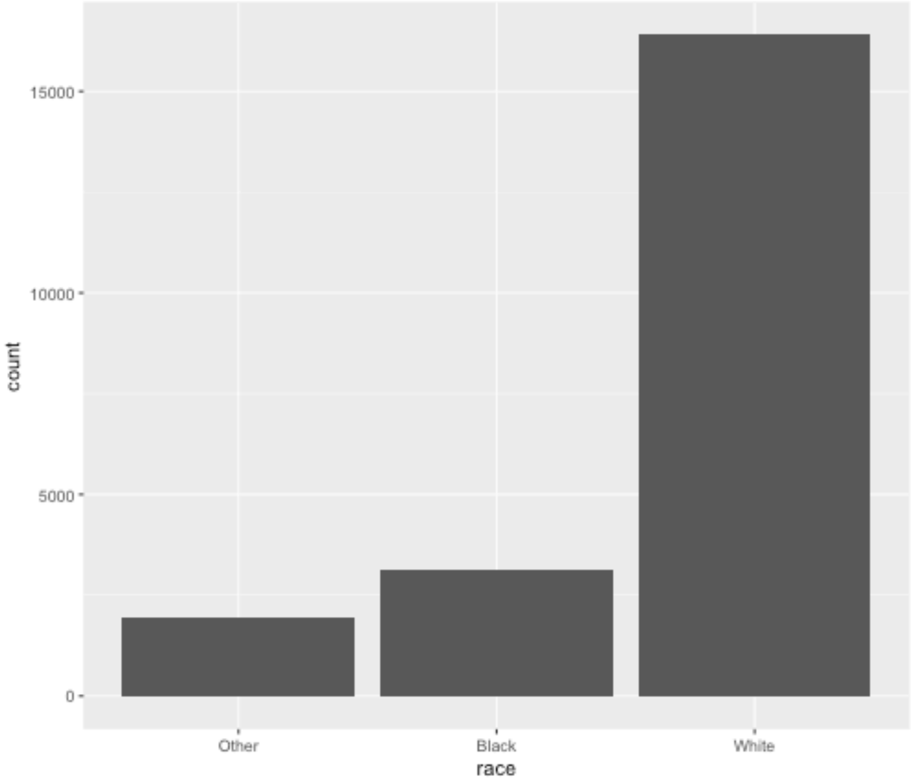


```
ggplot(gss_cat, aes(relig)) +
  geom_bar()
```

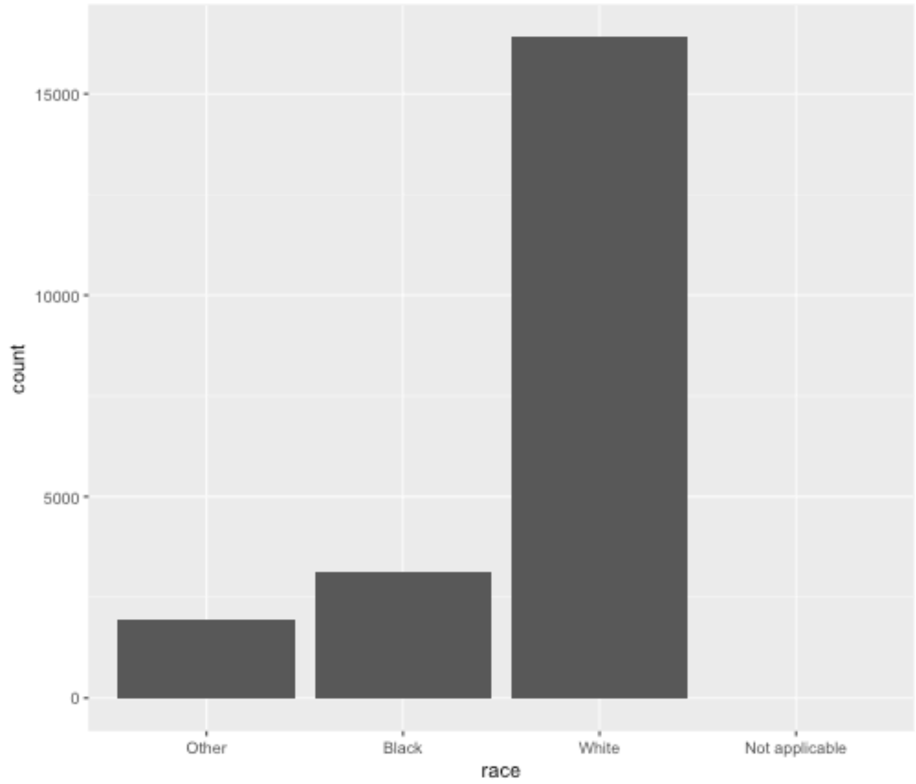


Include missing categories

```
ggplot(gss_cat, aes(race)) +
    geom_bar()
```

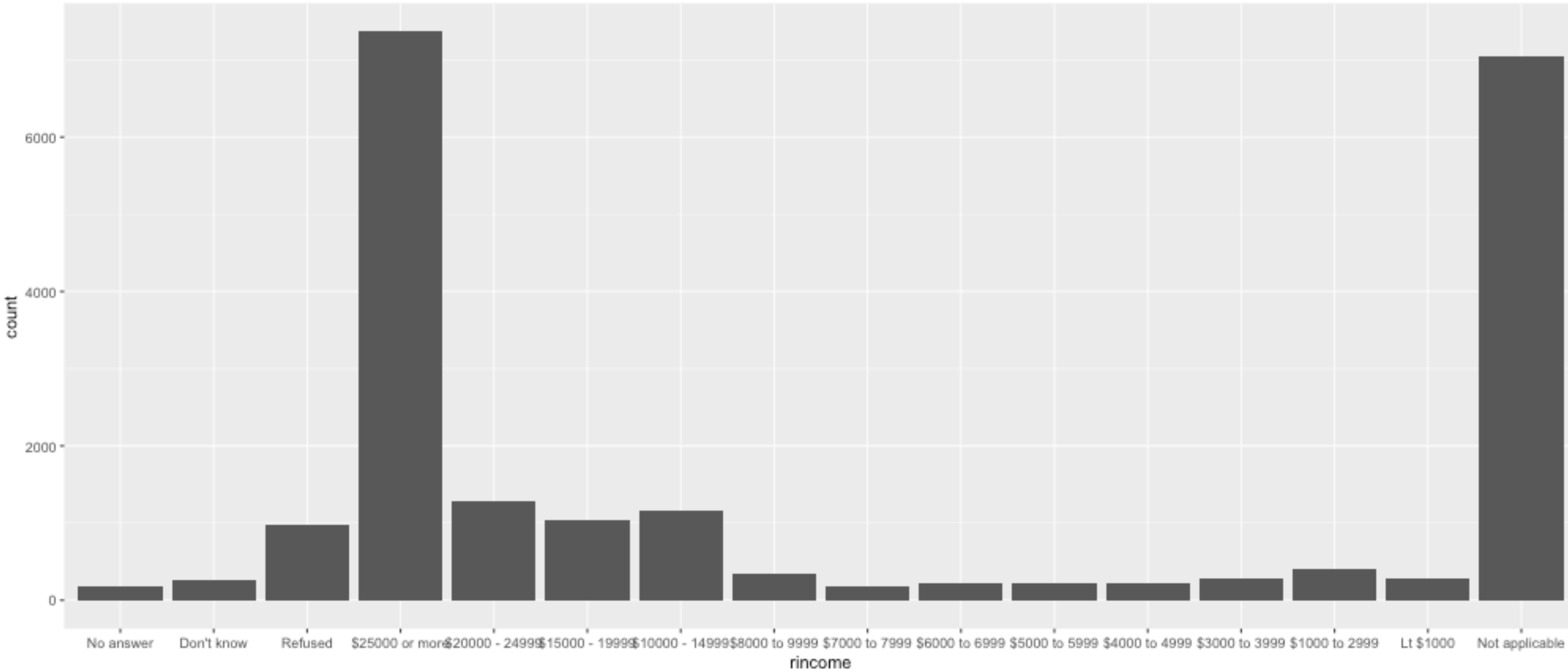



```
ggplot(gss_cat, aes(race)) +
    geom_bar() +
    scale_x_discrete(drop = FALSE)
```



What about this?

```
ggplot(gss_cat, aes(rincome)) +
    geom_bar()
```



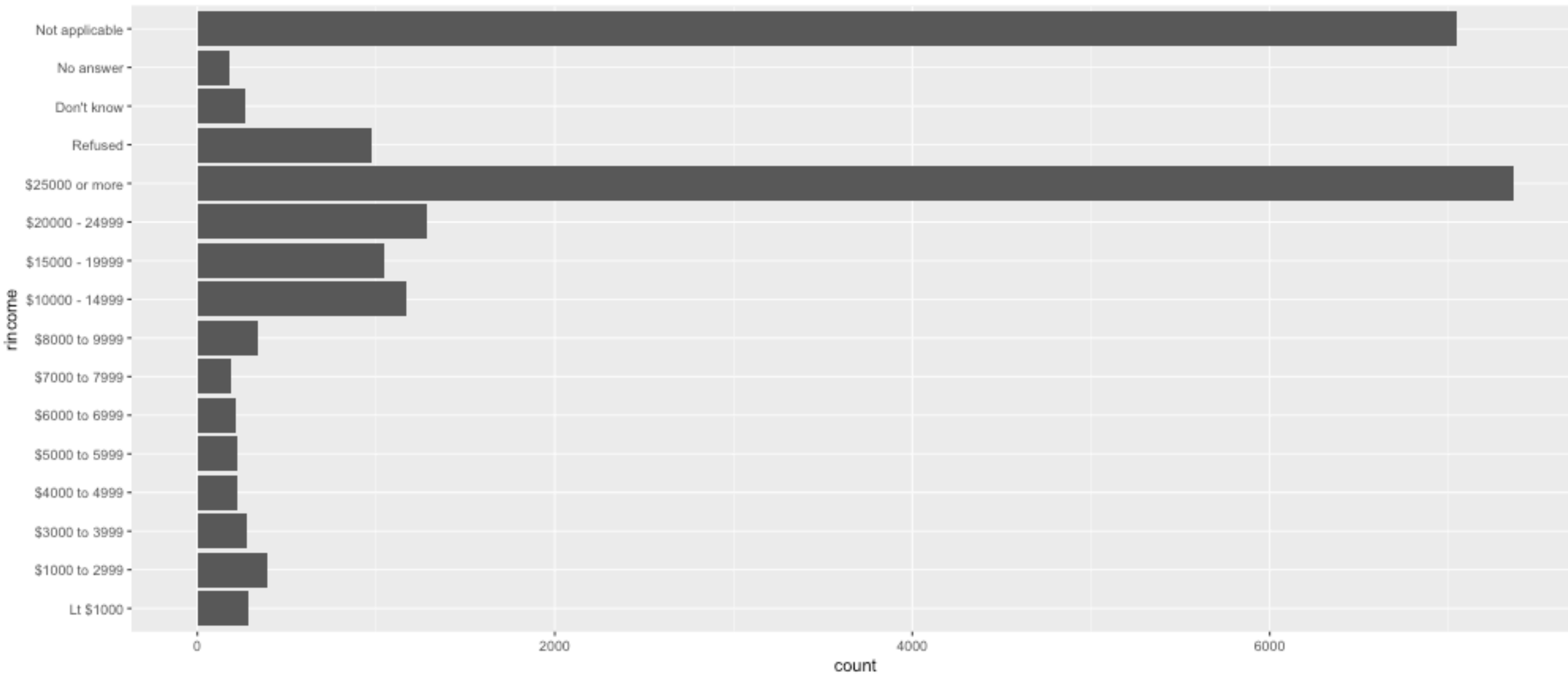
```
levels(gss_cat$rincome)
```

```
##  [1] "No answer"      "Don't know"     "Refused"        "$25000 or more"
##  [5] "$20000 - 24999" "$15000 - 19999" "$10000 - 14999" "$8000 to 9999"
##  [9] "$7000 to 7999"  "$6000 to 6999"  "$5000 to 5999"  "$4000 to 4999"
## [13] "$3000 to 3999"  "$1000 to 2999"  "Lt $1000"       "Not applicable"
```

```
gss <- gss_cat %>%
    mutate(rincome = factor(rincome, levels = levels(rincome)[c(15:1, 16)]))
levels(gss$rincome)
```

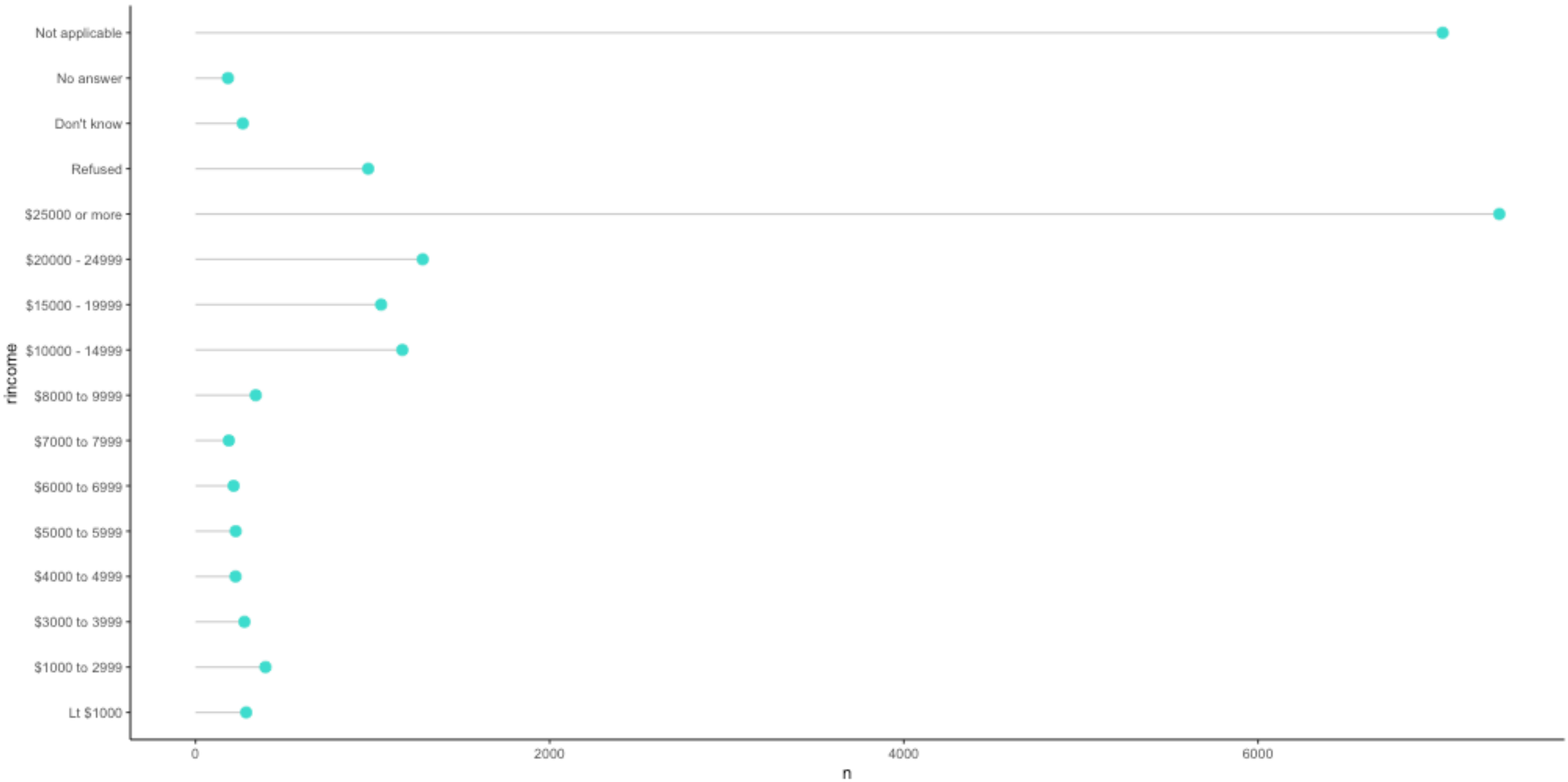
```
##  [1] "Lt $1000"       "$1000 to 2999"  "$3000 to 3999"  "$4000 to 4999"
##  [5] "$5000 to 5999"  "$6000 to 6999"  "$7000 to 7999"  "$8000 to 9999"
##  [9] "$10000 - 14999" "$15000 - 19999" "$20000 - 24999" "$25000 or more"
## [13] "Refused"        "Don't know"     "No answer"      "Not applicable"
```

```r
ggplot(gss, aes(rincome)) +
    geom_bar() +
    coord_flip()
```



Quick aside (and somewhat controversial)

Lollypop charts!



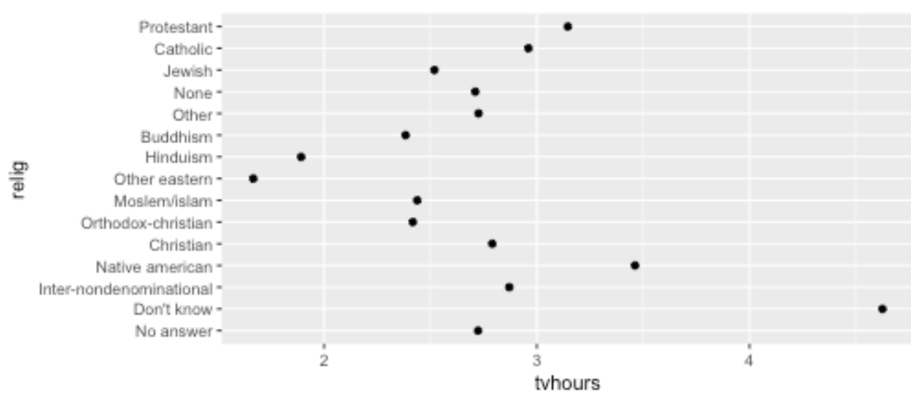
code

```
counts <- gss %>%
    count(rincome)
ggplot(counts, aes(rincome, n)) +
    geom_segment(aes(x = rincome, xend = rincome,
                     y = 0, yend = n),
                 col = "gray80") +
    geom_point(size = 3, col = "turquoise") +
    coord_flip() +
    theme_classic()
```

Reorder factors

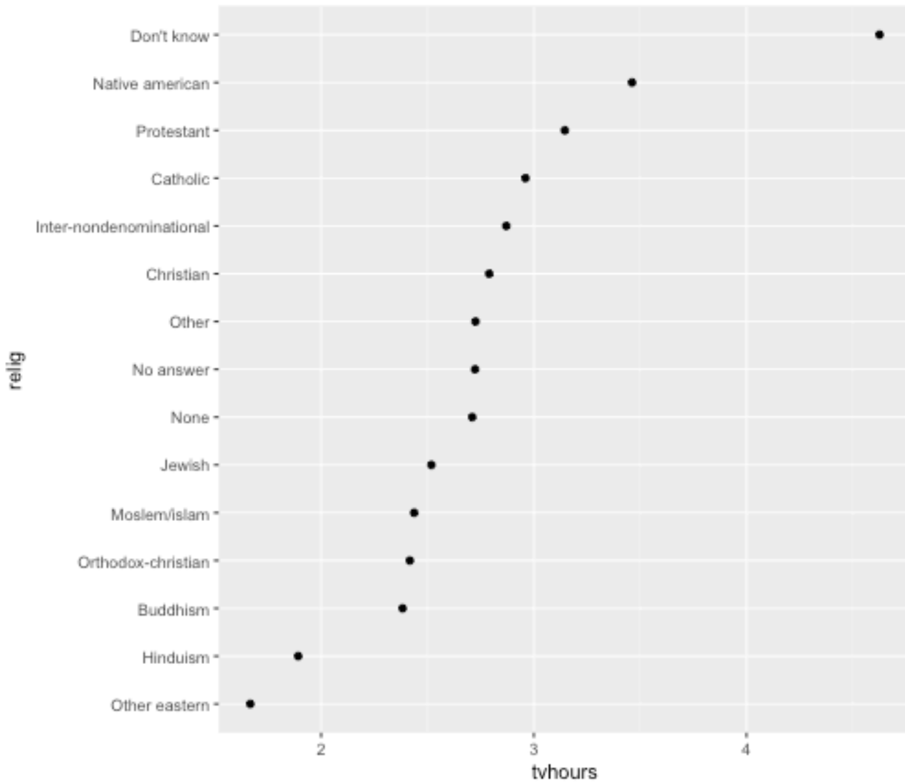
The `forcats::fct_reorder` function allows you to easily reorder factors according to another variable

```
relig_summary <- gss_cat %>%
  group_by(relig) %>%
  summarise(age = mean(age, na.rm = TRUE),
            tvhours = mean(tvhours, na.rm = TRUE),
            n = n())

ggplot(relig_summary, aes(tvhours, relig)) + geom_point()
```



Note - you could actually include the factor reorder right within the `ggplot` call.

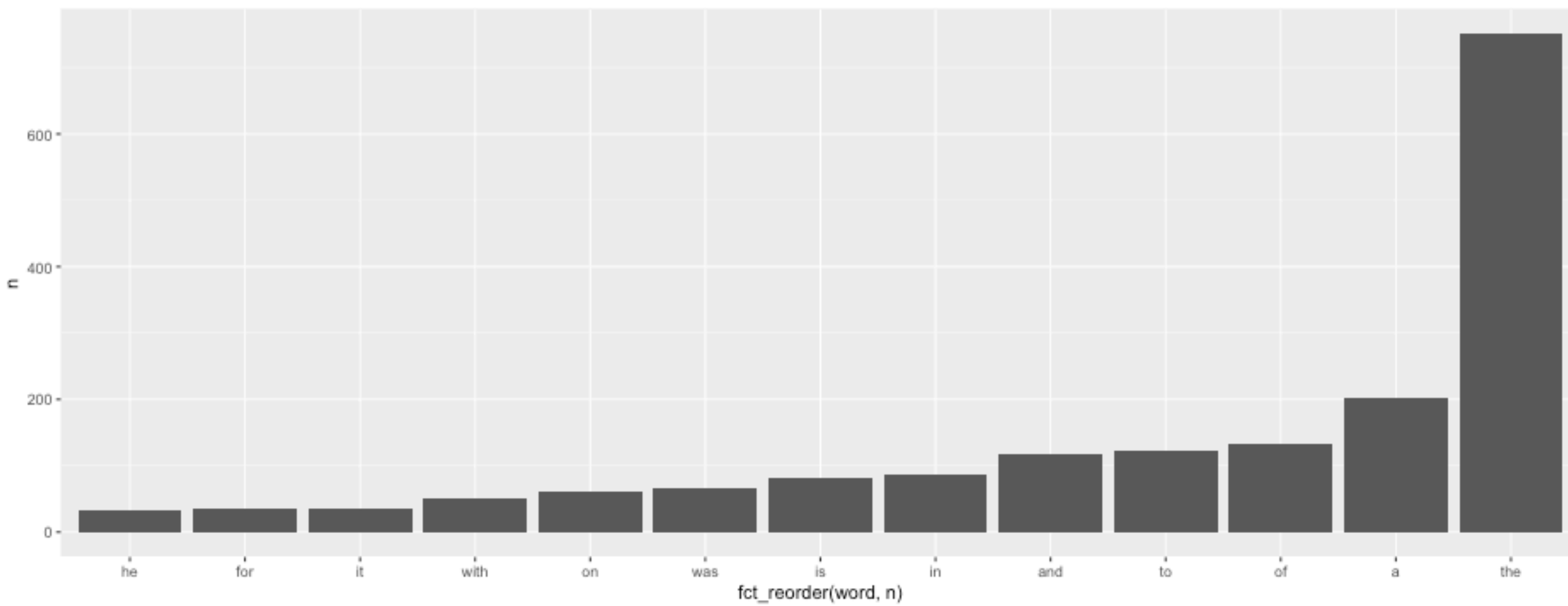
```
relig_summary <- relig_summary %>%
    mutate(relig = fct_reorder(relig, tvhours))

ggplot(relig_summary, aes(tvhours, relig)) + geom_point()
```



Revisiting our word frequency example

- An easier way to do what we did before, would be to just include the reorder call right within the call to ggplot

```
ggplot(words_freq, aes(fct_reorder(word, n), n)) +
    geom_col()
```



More on modifying factor levels

- The `forcats::fct_recode` function can make modifying factors more explicit

```
gss_cat %>%
  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")) %>%
  count(partyid)
```

```
## # A tibble: 10 x 2
##                  partyid     n
##                   <fctr> <int>
##  1             No answer   154
##  2            Don't know     1
##  3           Other party   393
##  4    Republican, strong  2314
##  5      Republican, weak  3032
##  6 Independent, near rep  1791
```

But this can be pretty easily done with base code too

```
levels(gss_cat$partyid)
```

```
##  [1] "No answer"          "Don't know"         "Other party"
##  [4] "Strong republican"  "Not str republican" "Ind,near rep"
##  [7] "Independent"        "Ind,near dem"       "Not str democrat"
## [10] "Strong democrat"
```

```
levels(gss_cat$partyid)[c(4:6, 8:10)] <- c("Republican, strong",
    "Republican, weak", "Independent, near rep", "Independent, near dem",
    "Democrat, weak", "Democrat, strong")
levels(gss_cat$partyid)
```

```
##  [1] "No answer"             "Don't know"
##  [3] "Other party"           "Republican, strong"
##  [5] "Republican, weak"      "Independent, near rep"
##  [7] "Independent"           "Independent, near dem"
##  [9] "Democrat, weak"        "Democrat, strong"
```

Collapsing levels

- `fct_recode` can also be used to collapse levels easily

```
gss_cat %>%
  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",
    "Other"                 = "No answer",
    "Other"                 = "Don't know",
    "Other"                 = "Other party")) %>%
  count(partyid)
```

```
## # A tibble: 8 x 2
##                partyid     n
##                 <fctr> <int>
## 1                Other   548
## 2    Republican, strong  2314
## 3      Republican, weak  3032
## 4 Independent, near rep  1791
## 5           Independent  4119
## 6 Independent, near dem  2499
## 7        Democrat, weak  3690
## 8      Democrat, strong  3490
```

Or with base syntax

```
data(gss_cat)
levels(gss_cat$partyid)
```

```
##  [1] "No answer"          "Don't know"         "Other party"
##  [4] "Strong republican"  "Not str republican" "Ind,near rep"
##  [7] "Independent"        "Ind,near dem"       "Not str democrat"
## [10] "Strong democrat"
```

```
levels(gss_cat$partyid)[-7] <- c("other", "other", "other",
    "Republican, strong", "Republican, weak",
    "Independent, near rep", "Independent, near dem",
    "Democrat, weak", "Democrat, strong")
```

Collapse a lot of categories

- In my mind, the most useful functions in *forcats* are for collapsing a lot of categories.
- For example, collapse all categories into *republican*, *democrat*, *independent*, or *other*.

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
  )) %>%
  count(partyid)
```

```
## # A tibble: 4 x 2
##   partyid     n
##    <fctr> <int>
## 1   other   548
## 2     rep  5346
## 3     ind  8409
## 4     dem  7180
```

Sometimes even better

- We can "lump" a bunch of categories together using `fct_lump`.
- Default behavior of `fct_lump` is to create an "other" group that includes all the smallest groups while maintaining "other" as the smallest group represented.
- Can also take optional *n* argument, where *n* represents the number of groups to collapse to

```
gss_cat %>%
    mutate(rel = fct_lump(relig)) %>%
    count(rel)
```

```
## # A tibble: 2 x 2
##          rel     n
##       <fctr> <int>
## 1 Protestant 10846
## 2      Other 10637
```

Collapse to 10 religious groups

```
gss_cat %>%
    mutate(rel = fct_lump(relig, n = 10)) %>%
    count(rel)
```

```
## # A tibble: 10 x 2
##                         rel     n
##                      <fctr> <int>
##  1 Inter-nondenominational   109
##  2               Christian   689
##  3      Orthodox-christian    95
##  4             Moslem/islam   104
##  5                Buddhism   147
##  6                    None  3523
##  7                  Jewish   388
##  8                Catholic  5124
##  9              Protestant 10846
## 10                   Other   458
```

One last thing...

Factors with modeling

```r
colors <- factor(c("black", "green", "blue", "blue", "black"))
```

- No need for multiple variables to define a categorical variable: internal dummy-coding

```r
contrasts(colors)
```

```
##       blue green
## black    0     0
## blue     1     0
## green    0     1
```

- Change the reference group by defining a new contrast matrix. For example, we can set green to the reference group with the following code.

```r
contrasts(colors) <- matrix(c(1, 0,
                              0, 1,
                              0, 0),
                            byrow = TRUE,
                            ncol = 2)
```

Contrast coding (continued)

Alternatively, use some of the built in functions for defining new contrasts matrices

```
contr.helmert(3)
```

```
##   [,1] [,2]
## 1   -1   -1
## 2    1   -1
## 3    0    2
```

```
contr.sum(3)
```

```
##   [,1] [,2]
## 1    1    0
## 2    0    1
## 3   -1   -1
```

```
contrasts(colors) <- contr.helmert(3)
contrasts(colors)
```

```
##       [,1] [,2]
## black   -1   -1
## blue     1   -1
## green    0    2
```

```
contrasts(colors) <- contr.sum(3)
contrasts(colors)
```

```
##       [,1] [,2]
## black    1    0
## blue     0    1
## green   -1   -1
```

(see: http://www.ats.ucla.edu/stat/r/library/contrast_coding.htm)

Pause...

Questions?

Dates

Intro

- Dates are hard - harder than they might seem
- Base syntax can be tricky
- Lots of different packages for helping with dates and time-series data
- We'll focus on the tidyverse version: *lubridate*

Three different types of "Dates"

- date
- date-time (POSIXct)
- time (doesn't have its own class, *hms* package can help here, if you need it)

POSIXct data are much more complicated than dates, so use regular dates if possible.

Date variables look like this:

```
library(lubridate)
today()
```

```
## [1] "2017-05-29"
```

- This is the standard ISO date format: YYYY-MM-DD.
- Any date variable you have, in any format, will end up looking like this after you convert it to a date.

POSIXct/date-time variables look like this:

```
now()
```

```
## [1] "2017-05-29 12:47:14 PDT"
```

- Notice they include the date, but also the specific time (in military/24 hour format), down to the specific second.
- Also includes the timezone, which is of course important if you're dealing with seconds of data.

Creating dates

- When you read in data, the dates are likely to be in all sorts of different formats.
- Hopefully, they're at least consistent within a column
- *lubridate* makes individual conversions relatively easy.

```
ymd("2012/02/14")
```

```
## [1] "2012-02-14"
```

```
mdy("03/10/2015")
```

```
## [1] "2015-03-10"
```

```
mdy("03 10 15")
```

```
## [1] "2015-03-10"
```

Conversions

```
ymd()
ydm()
mdy()
myd()
dmy()
dym()
yq()
```

Need to convert a date-time?

```
mdy_hms("04/16/12 11:48.32 AM")
```

```
## [1] "2012-04-16 11:48:32 UTC"
```

Enforce a time zone

```
mdy_hms("04/16/12 11:48.32 AM", tz = "America/Los_Angeles")
```

```
## [1] "2012-04-16 11:48:32 PDT"
```

```
mdy_hms("04/16/12 11:48.32 AM", tz = "America/New_York")
```

```
## [1] "2012-04-16 11:48:32 EDT"
```

Switch between types

```
as_datetime(today())
```

```
## [1] "2017-05-29 UTC"
```

```
as_date(now())
```

```
## [1] "2017-05-29"
```

Numerical dates

- Sometimes you'll run up against dates like `16750` or `-1250`
- These are number deviating from the "Unix Epoch", which is 1970-01-01

```
as_date(4380) # interpreted as days
```

```
## [1] "1981-12-29"
```

```
as_datetime(4380) # interpreted as seconds
```

```
## [1] "1970-01-01 01:13:00 UTC"
```

Parsing other formats

- What format is *dep_time* in?

```
library(nycflights13)
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
```

So how could we parse this to be usable?

```
unique(flights$dep_time)
```

```
##   [1]  517  533  542  544  554  555  557  558  559  600  601  602  606
##  [14]  607  608  611  613  615  622  623  624  627  628  629  632  635
##  [27]  637  639  643  644  645  646  651  652  653  655  656  657  658
##  [40]  659  701  702  709  711  712  715  717  719  723  724  725  727
##  [53]  728  729  732  733  734  739  741  743  745  746  749  752  753
##  [66]  754  758  759  800  801  803  804  805  807  809  810  811  812
##  [79]  813  814  817  820  821  822  823  824  825  826  828  829  830
##  [92]  831  832  833  835  839  840  846  848  851  852  853  855  856
## [105]  857  858  859  900  902  903  904  905  906  908  909  912  913
## [118]  914  917  920  921  923  926  927  929  930  931  932  933  936
## [131]  937  940  941  946  947  950  953  955  956  957  959 1003 1005
## [144] 1007 1009 1010 1011 1021 1024 1025 1026 1028 1029 1030 1031 1032
## [157] 1033 1037 1038 1042 1044 1047 1048 1053 1054 1056 1058 1059 1101
## [170] 1103 1105 1107 1109 1111 1112 1113 1114 1120 1123 1124 1125 1127
## [183] 1128 1130 1132 1133 1135 1137 1143 1144 1147 1150 1153 1154 1155
## [196] 1157 1158 1200 1202 1203 1204 1205 1206 1208 1211 1217 1219 1220
## [209] 1222 1228 1230 1231 1237 1238 1240 1241 1245 1246 1248 1251 1252
## [222] 1253 1255 1257 1258 1301 1302 1304 1305 1306 1310 1314 1315 1316
## [235] 1317 1318 1320 1323 1325 1327 1330 1333 1336 1337 1339 1341 1342
## [248] 1343 1344 1346 1350 1351 1353 1354 1355 1356 1358 1400 1402 1408
## [261] 1411 1416 1418 1419 1421 1422 1423 1424 1428 1430 1431 1433 1436
```

The way I'd probably do it

```
flights %>%
    mutate(dep_time = stringr::str_pad(dep_time, 4, pad = "0")) %>%
    separate(dep_time, c("dep_hour", "dep_minute"), 2, convert = TRUE)
```

```
## # A tibble: 336,776 x 20
##     year month   day dep_hour dep_minute sched_dep_time dep_delay arr_time
##  * <int> <int> <int>    <int>      <int>          <int>     <dbl>    <int>
##  1  2013     1     1        5         17            515         2      830
##  2  2013     1     1        5         33            529         4      850
##  3  2013     1     1        5         42            540         2      923
##  4  2013     1     1        5         44            545        -1     1004
##  5  2013     1     1        5         54            600        -6      812
##  6  2013     1     1        5         54            558        -4      740
##  7  2013     1     1        5         55            600        -5      913
##  8  2013     1     1        5         57            600        -3      709
##  9  2013     1     1        5         57            600        -3      838
## 10  2013     1     1        5         58            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

How they handle it in the book

Modulo operators

- `%/%`: Integer division
- `%%`: Remainder

```
123 %/% 100
```

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

```
123 %% 100
```

```
## [1] 23
```

```r
flights %>%
    mutate(dep_hour = dep_time %/% 100,
           dep_min = dep_time %% 100) %>%
    select(tailnum, dep_time, dep_hour, dep_min)
```

```
## # A tibble: 336,776 x 4
##    tailnum dep_time dep_hour dep_min
##      <chr>    <int>    <dbl>   <dbl>
##  1  N14228      517        5      17
##  2  N24211      533        5      33
##  3  N619AA      542        5      42
##  4  N804JB      544        5      44
##  5  N668DN      554        5      54
##  6  N39463      554        5      54
##  7  N516JB      555        5      55
##  8  N829AS      557        5      57
##  9  N593JB      557        5      57
## 10  N3ALAA      558        5      58
## # ... with 336,766 more rows
```

Creating dates from multiple variables

- Take a minute… How might you think we could create a single date variable?

```
flights %>%
    select(year, month, day, hour, minute)
```

```
## # A tibble: 336,776 x 5
##     year month   day  hour minute
##    <int> <int> <int> <dbl>  <dbl>
##  1  2013     1     1     5     15
##  2  2013     1     1     5     29
##  3  2013     1     1     5     40
##  4  2013     1     1     5     45
##  5  2013     1     1     6      0
##  6  2013     1     1     5     58
##  7  2013     1     1     6      0
##  8  2013     1     1     6      0
##  9  2013     1     1     6      0
## 10  2013     1     1     6      0
## # ... with 336,766 more rows
```

Nice *lubridate* functions

- `make_date()` and `make_datetime()` functions that can save us a boatload of time.
- Arguments are: year, month, day, hour, min, sec, and tz.
- All arguments have defaults, which are: `1970L`, `1L`, `0L`, `0L`, `0`, and `"UTC"`

```
flights %>%
    select(year, month, day, hour, minute) %>%
    mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
## # A tibble: 336,776 x 6
##      year month   day  hour minute           departure
##     <int> <int> <int> <dbl>  <dbl>              <dttm>
##  1   2013     1     1     5     15 2013-01-01 05:15:00
##  2   2013     1     1     5     29 2013-01-01 05:29:00
##  3   2013     1     1     5     40 2013-01-01 05:40:00
##  4   2013     1     1     5     45 2013-01-01 05:45:00
##  5   2013     1     1     6      0 2013-01-01 06:00:00
##  6   2013     1     1     5     58 2013-01-01 05:58:00
##  7   2013     1     1     6      0 2013-01-01 06:00:00
##  8   2013     1     1     6      0 2013-01-01 06:00:00
##  9   2013     1     1     6      0 2013-01-01 06:00:00
```

Going in reverse

```
datetime <- ymd_hms("2016-07-08 12:34:56")

year(datetime)
```

```
## [1] 2016
```

```
month(datetime)
```

```
## [1] 7
```

```
mday(datetime)
```

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

Calculating Time Spans

- Really common situation for me: Dataset like the below, need to calculate number of weeks/months, etc., either between dates, or from a specific date.

```
sid <- rep(1:4, each = 3)
date <- c("9/3/08", "12/10/08", "4/22/09", "8/29/08", "12/5/08", "4/17/09", "8/29/08", "12/4/(
score <- c(222, 225, 223, 194, 196, 201, 194, 209, 197, 191, 197, 214)
d <- data.frame(sid = sid, date = date, score = score)
d
```

```
##    sid     date score
## 1    1   9/3/08   222
## 2    1 12/10/08   225
## 3    1  4/22/09   223
## 4    2  8/29/08   194
## 5    2  12/5/08   196
## 6    2  4/17/09   201
## 7    3  8/29/08   194
## 8    3  12/4/08   209
## 9    3  4/23/09   197
## 10   4   9/3/08   191
## 11   4  12/1/08   197
```

First - convert to date

```
d <- d %>%
    mutate(date = mdy(date))
d
```

```
##    sid       date score
## 1    1 2008-09-03   222
## 2    1 2008-12-10   225
## 3    1 2009-04-22   223
## 4    2 2008-08-29   194
## 5    2 2008-12-05   196
## 6    2 2009-04-17   201
## 7    3 2008-08-29   194
## 8    3 2008-12-04   209
## 9    3 2009-04-23   197
## 10   4 2008-09-03   191
## 11   4 2008-12-01   197
## 12   4 2009-04-20   214
```

What to compute from?

- In my case, I often want to calculate the date from the first day of the school year.
- First, create a date object with that date

```
first_day <- mdy("08/05/2008")
first_day
```

```
## [1] "2008-08-05"
```

- Next, compute the difference between that date and the corresponding date the test was administered.

```
d %>%
    mutate(days_elapsed = date - first_day)
```

```
##    sid       date score days_elapsed
## 1    1 2008-09-03   222      29 days
## 2    1 2008-12-10   225     127 days
## 3    1 2009-04-22   223     260 days
## 4    2 2008-08-29   194      24 days
## 5    2 2008-12-05   196     122 days
## 6    2 2009-04-17   201     255 days
## 7    3 2008-08-29   194      24 days
## 8    3 2008-12-04   209     121 days
## 9    3 2009-04-23   197     261 days
## 10   4 2008-09-03   191      29 days
## 11   4 2008-12-01   197     118 days
## 12   4 2009-04-20   214     258 days
```

Take a second...

What if I wanted to calculate date from the first assessment?

One method

```
d %>%
    group_by(sid) %>%
  arrange(date) %>%
    mutate(first_date = first(date),
           days_elapsed = date - first_date) %>%
  arrange(sid)
```

```
## # A tibble: 12 x 5
## # Groups:   sid [4]
##      sid       date score first_date days_elapsed
##    <int>     <date> <dbl>     <date>       <time>
##  1     1 2008-09-03   222 2008-09-03       0 days
##  2     1 2008-12-10   225 2008-09-03      98 days
##  3     1 2009-04-22   223 2008-09-03     231 days
##  4     2 2008-08-29   194 2008-08-29       0 days
##  5     2 2008-12-05   196 2008-08-29      98 days
##  6     2 2009-04-17   201 2008-08-29     231 days
##  7     3 2008-08-29   194 2008-08-29       0 days
##  8     3 2008-12-04   209 2008-08-29      97 days
##  9     3 2009-04-23   197 2008-08-29     237 days
## 10     4 2008-09-03   191 2008-09-03       0 days
```

What if I wanted days between each assessment?

- Some knowledge of base R comes in handy here: `lag`

```
d %>%
    group_by(sid) %>%
  arrange(date) %>%
    mutate(days_between = date - lag(date)) %>%
  arrange(sid)
```

```
## # A tibble: 12 x 4
## # Groups:   sid [4]
##       sid       date score days_between
##     <int>     <date> <dbl>       <time>
##  1      1 2008-09-03   222      NA days
##  2      1 2008-12-10   225      98 days
##  3      1 2009-04-22   223     133 days
##  4      2 2008-08-29   194      NA days
##  5      2 2008-12-05   196      98 days
##  6      2 2009-04-17   201     133 days
##  7      3 2008-08-29   194      NA days
##  8      3 2008-12-04   209      97 days
##  9      3 2009-04-23   197     140 days
```

Different metric?

- Suppose I instead wanted weeks

```
first_day_weeks <- week(first_day)
first_day_weeks
```

```
## [1] 32
```

```
d <- d %>%
    mutate(weeks_elapsed = week(date) - first_day_weeks)
d
```

```
## Source: local data frame [12 x 5]
## Groups: sid [4]
##
## # A tibble: 12 x 5
##      sid       date score occasion weeks_elapsed
##    <int>     <date> <dbl>    <int>         <dbl>
##  1     1 2008-09-03   222        1             4
##  2     1 2008-12-10   225        2            18
##  3     1 2009-04-22   223        3           -16
##  4     2 2008-08-29   194        1             3
```

Check

Uh oh...What to do?

```
d %>%
    mutate(days_elapsed = date - first_day,
           check = days_elapsed / 7) %>%
    select(weeks_elapsed, check)
```

```
## Source: local data frame [12 x 3]
## Groups: sid [4]
##
## # A tibble: 12 x 3
##       sid weeks_elapsed          check
##     <int>         <dbl>         <time>
## 1       1             4  4.142857 days
## 2       1            18 18.142857 days
## 3       1           -16 37.142857 days
## 4       2             3  3.428571 days
## 5       2            17 17.428571 days
## 6       2           -16 36.428571 days
## 7       3             3  3.428571 days
## 8       3            17 17.285714 days
## 9       3           -15 37.285714 days
```

What about months?

One method...

```
first_day_months <- month(first_day)
first_day_months
```

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

```
d <- d %>%
    mutate(months_elapsed = ifelse(year(date) == "2008",
                            month(date) - first_day_months,
                            (month(date) - first_day_months) + 12))
```

```r
d
```

```
## Source: local data frame [12 x 6]
## Groups: sid [4]
##
## # A tibble: 12 x 6
##      sid       date score occasion weeks_elapsed months_elapsed
##    <int>     <date> <dbl>    <int>         <dbl>          <dbl>
##  1     1 2008-09-03   222        1             4              1
##  2     1 2008-12-10   225        2            18              4
##  3     1 2009-04-22   223        3           -16              8
##  4     2 2008-08-29   194        1             3              0
##  5     2 2008-12-05   196        2            17              4
##  6     2 2009-04-17   201        3           -16              8
##  7     3 2008-08-29   194        1             3              0
##  8     3 2008-12-04   209        2            17              4
##  9     3 2009-04-23   197        3           -15              8
## 10     4 2008-09-03   191        1             4              1
## 11     4 2008-12-01   197        2            16              4
## 12     4 2009-04-20   214        3           -16              8
```

Alternative

Non-tidyverse package but useful for months, specifically *mondate*

```
# install.packages("mondate")
library(mondate)
first_day_mondate <- as.mondate(first_day)
first_day_mondate
```

```
## mondate: timeunits="months"
## [1] 08/05/2008
```

```r
d <- d %>%
    mutate(mondate = as.mondate(date),
           months_elapsed2 = mondate - first_day_mondate)
d
```

```
##    sid       date score months_elapsed    mondate  months_elapsed2
## 1    1 2008-09-03   222              1 09/03/2008 0.9387097 months
## 2    1 2008-12-10   225              4 12/10/2008 4.1612903 months
## 3    1 2009-04-22   223              8 04/22/2009 8.5720430 months
## 4    2 2008-08-29   194              0 08/29/2008 0.7741935 months
## 5    2 2008-12-05   196              4 12/05/2008 4.0000000 months
## 6    2 2009-04-17   201              8 04/17/2009 8.4053763 months
## 7    3 2008-08-29   194              0 08/29/2008 0.7741935 months
## 8    3 2008-12-04   209              4 12/04/2008 3.9677419 months
## 9    3 2009-04-23   197              8 04/23/2009 8.6053763 months
## 10   4 2008-09-03   191              1 09/03/2008 0.9387097 months
## 11   4 2008-12-01   197              4 12/01/2008 3.8709677 months
## 12   4 2009-04-20   214              8 04/20/2009 8.5053763 months
```

A few last notes on dates

- *lubridate* provides **duration** and **period** classes that may be helpful
  - durations are always reported in seconds
- Periods help account for things like time zones and leap years

Use durations to calculate dates

```
today() + ddays(123)
```

```
## [1] "2017-09-29"
```

```
today() + dweeks(1)
```

```
## [1] "2017-06-05"
```

```
today() - dyears(1)
```

```
## [1] "2016-05-29"
```

Another alternative for months

```
d %>%
    mutate(days_elapsed = date - first_day,
            seconds_elapsed = as.duration(days_elapsed),
            months_elapsed3 = seconds_elapsed / 2.628e+6) %>%
    select(contains("month"))
```

```
##    months_elapsed  months_elapsed2    months_elapsed3
## 1               1 0.9387097 months 0.953424657534247s
## 2               4 4.1612903 months  4.17534246575342s
## 3               8 8.5720430 months  8.54794520547945s
## 4               0 0.7741935 months 0.789041095890411s
## 5               4 4.0000000 months  4.01095890410959s
## 6               8 8.4053763 months  8.38356164383562s
## 7               0 0.7741935 months 0.789041095890411s
## 8               4 3.9677419 months  3.97808219178082s
## 9               8 8.6053763 months  8.58082191780822s
## 10              1 0.9387097 months 0.953424657534247s
## 11              4 3.8709677 months  3.87945205479452s
## 12              8 8.5053763 months  8.48219178082192s
```

periods

```
one_pm <- ymd_hms("2016-03-12 13:00:00", tz = "America/New_York")
one_pm
```

```
## [1] "2016-03-12 13:00:00 EST"
```

```
one_pm + ddays(1)
```

```
## [1] "2016-03-13 14:00:00 EDT"
```

```
one_pm + days(1)
```

```
## [1] "2016-03-13 13:00:00 EDT"
```

Summary

- Dates are harder than expected
  - time zones
  - leap years
  - daylight savings, etc.
- *lubridate* can help, but you always need to be careful
- We didn't talk about calculating seconds, milliseconds, etc., but that's easily done as well.