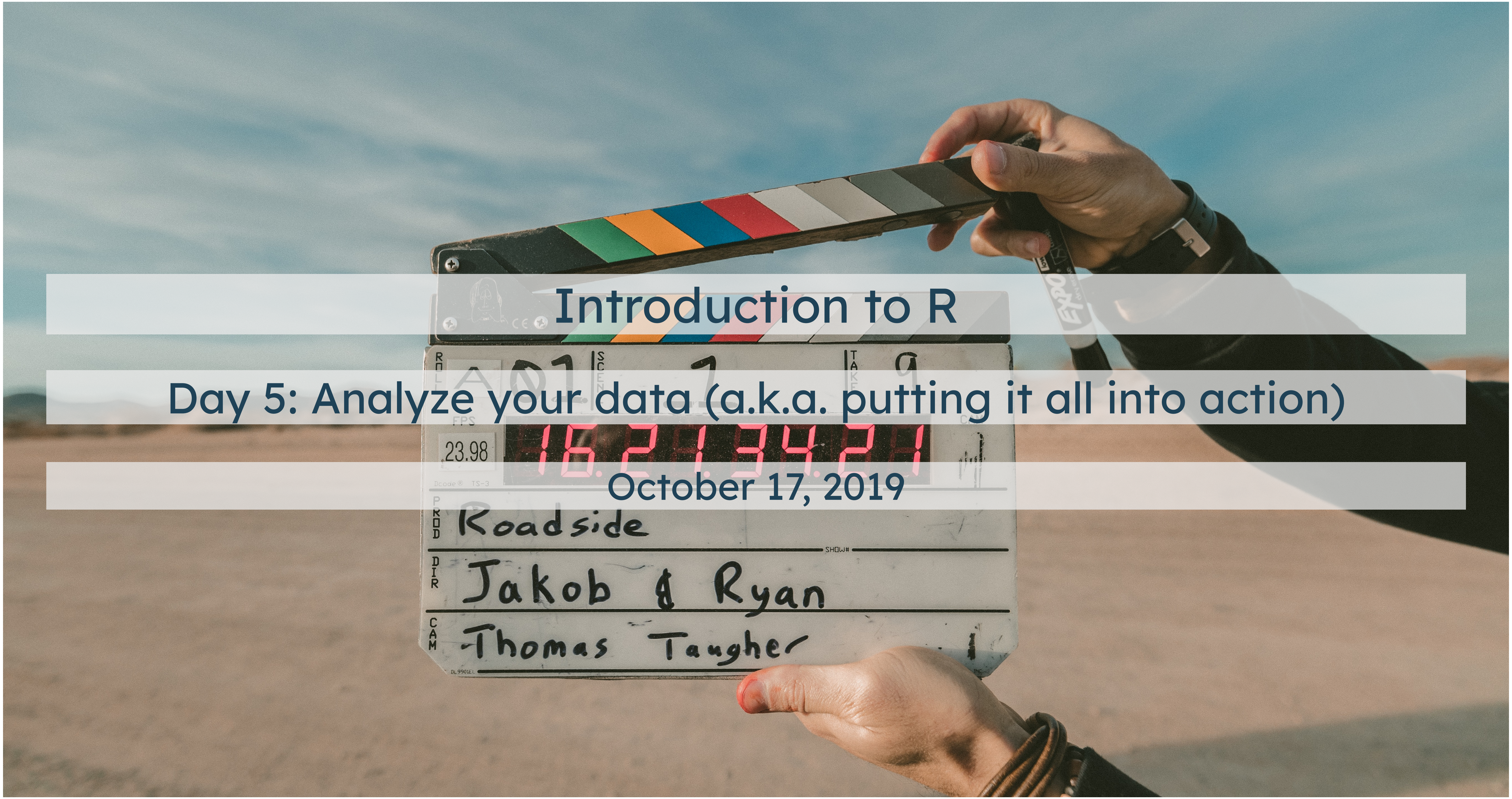
Introduction to R

Day 5: Analyze your data (a.k.a. putting it all into action)

October 17, 2019



Agenda

Day 1: Figures

Day 2: Selecting, filtering, and mutating

Day 3: Grouping and tables

Day 4: Functions

Day 5: Analyze your data

Agenda

Day 1: Figures ✅

Day 2: Selecting, filtering, and mutating ✅

Day 3: Grouping and tables ✅

Day 4: Functions ✅

Day 5: Analyze your data 😢

Put everything you've learned into action, and more!

Organization

```
my-project/
 ├─ my-project.Rproj
 ├─ README
 ├─ data/
 |    ├── raw/
 |    └── processed/
 ├─ code/
 ├─ results/
 |    ├── tables/
 |    ├── figures/
 |    └── output/
 └─ docs/
```

- An `.Rproj` file is mostly just a placeholder. It remembers various options, and makes it easy to open a new RStudio session that starts up in the correct working directory. You never need to edit it directly.
- A README file can just be a text file that includes notes for yourself or future users.
- I like to have a folder for raw data -- which I never touch -- and a folder(s) for datasets that I create along the way.

Referring to files with the `here` package

```r
source(here::here("code", "functions.R"))

dat <- read_csv(here::here(
              "data", "raw", "data.csv"))

p <- ggplot(dat) + geom_point(aes(x, y))

ggsave(plot = p,
       filename = here::here(
       "results", "figures", "fig.pdf"))
```

- The `here` package lets you refer to files without worrying too much about relative paths.
- Construct file paths with reference to the top directory holding your `.Rproj` file.
- `here::here("data", "raw", "data.csv")` for me, here, becomes `"/Users/louisahsmith/Google Drive/Teaching/R course/materials/data/raw/data.csv"`
- But if I send you this file, it will become whatever file path *you* need it to be.

Referring to the `here` package

```
here::here()
```

is equivalent to

```
library(here)
here()
```

I just prefer to write out the package name whenever I need it, but you can load the package for your entire session if you want.

Note that you can refer to any function without loading the whole package this way, e.g. **`tableone::CreateTableOne()`** instead of **`library(tableone); CreateTableOne()`**

The `source()` function

Will run code from another file.

```
source("script.R")
source(here::here("code", "functions.R"))
```

All the objects will be created, packages loaded, etc. as if you had run the code directly.

The `source()` function

Can even run code directly from a URL.

```
source("https://raw.githubusercontent.com/
        louisahsmith/intro-to-R/master/day1/
        day1-script1.R")
```

```
## Error in eval(ei, envir): object 'new_vals' not found
```

- Reading code from another file can make it a bit harder to debug.
- But it's nice when you have functions, etc. that you use a lot and want to include them at the start of every script.

Reading in data

You could also begin your scripts by reading in your data via a data-cleaning file with `source()`.

Each of these have different arguments that will allow you to read in specific columns only, skip rows, give the variables names, etc. There are also better options out there if your dataset is really big (look into the `data.table` or the `vroom` package), and if you have other types of data.

```
# the readr functions are loaded with library(tidyverse)
dat <- readr::read_csv("data.csv")
dat <- readr::read_table("data.dat")
# saved as an R object with write_rds()
dat <- readr::read_rds("data.rds")
dat <- readxl::read_excel("data.xlsx")
dat <- haven::read_sas("data.sas7bdat")
dat <- haven::read_stata("data.dta")
```

Saving your data

Once you've cleaned your data and created your dataset, you probably want to save another copy so you don't need to perform all your data cleaning functions every time you read it in.

- You can basically do the opposite of most of the `read` functions: `write`.
- The one I usually use, if I'm creating data for myself, is `write_rds()`. It creates an R object you can read in with `read_rds()`, so you can guarantee nothing will change in between writing and reading.
- If I'm sharing data, I usually use `write_csv()`.

Note: these are the tidyverse versions of the functions, which have better defaults, are more consistent, and are just more likely to do what you want. The "base R" versions are: `read.csv()`, `write.csv()`, `readRDS()` and `saveRDS()`.

Walk through the code so far and ask questions as needed

Exercises saved for a final challenge!

Missing values

- R uses `NA` for missing values
- Unlike some other statistical software, it will return `NA` to any logical statement
  - This makes it somewhat harder to deal with but also harder to make mistakes

```r
NA > 3
```

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

```r
mean(c(1, 2, NA))
```

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

```r
mean(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] 1.5
```

Special NA functions

Certain functions deal with missing values explicitly

```
vals <- c(1, 2, NA)
is.na(vals)
```

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

```
anyNA(vals)
```

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

```
na.omit(vals)
```

```
## [1] 1 2
```

Creating NAs with `na_if()`

You might read in data that has been created in another program or has special values to indicate missingness.

For example, in the NLSY data, -1 = Refused, -2 = Don't know, -3 = Invalid missing, -4 = Valid missing, -5 = Non-interview

```
nlsy[1, c("id", "glasses", "age_bir")]
```

```
## # A tibble: 1 x 3
##      id glasses age_bir
##   <dbl>   <dbl>   <dbl>
## 1     1      -4      -5
```

```
nlsy_na <- nlsy %>% na_if(-1) %>% na_if(-2) %>%
  na_if(-3) %>% na_if(-4) %>% na_if(-5)
nlsy_na[1, c("id", "glasses", "age_bir")]
```

```
## # A tibble: 1 x 3
##      id glasses age_bir
##   <dbl>   <dbl>   <dbl>
## 1     1      NA      NA
```

This is obviously a bit annoying if you have a lot of values that indicate missingness. In that case, you may want to look into the naniar package.

More `na_if()`

The `na_if()` strategy is generally the most useful if you're determining NA's over the course of your analysis, or if you have different NA values for different variables.

```
nlsy_bad <- nlsy %>%
  mutate(id = na_if(id, 1))
nlsy_bad[1:2, c("id", "glasses", "age_bir")]
```

```
## # A tibble: 2 x 3
##       id glasses age_bir
##    <dbl>   <dbl>   <dbl>
## 1     NA      -4      -5
## 2      2       0      34
```

Read in NA's directly

Or, if you know a priori which values indicate missingness (e.g., "."), you can specify that when reading in the data.

```
nlsy <- read_csv(here::here("day5", "nlsy.csv"),
          col_names = colnames_nlsy, skip = 1,
          na = c("-1", "-2", "-3", "-4", "-5"))
```

(You have to write the values as strings, even if they're numbers)

Complete cases

Sometimes you may just want to get rid of all the rows with missing values.

Don't do this without good reason!

```
nrow(nlsy)
```

```
## [1] 12686
```

```
nlsy_cc <- nlsy %>% filter(complete.cases(nlsy))
nrow(nlsy_cc)
```

```
## [1] 1436
```

```
nlsy2 <- nlsy %>% select(id, glasses, eyesight) %>% na.omit()
nrow(nlsy2)
```

```
## [1] 8444
```

Walk through the code so far and ask questions as needed

Exercises saved for a final challenge!

Sharing your results

First: some quick analysis

```
# load packages
library(tidyverse)
# must install if haven't already
library(broom) # for making pretty model output
library(splines) # for adding splines

# read in data
nlsy_clean <- read_rds(here::here("data", "nlsy_clean.rds"))
```

Quick analysis, cont.

Model formulas will automatically make indicator variables for factors, with the first level the reference. An intercept will be included unless suppressed with `y ~ -1 + x`.

```
# linear regression (OLS)
mod_lin1 <- lm(log_inc ~ age_bir + sex + race_eth,
                data = nlsy_clean)
# another way to do linear regression (GLM)
mod_lin2 <- glm(log_inc ~ age_bir + sex + race_eth,
               family = gaussian(link = "identity"),
               data = nlsy_clean)
# logistic regression
mod_log <- glm(glasses ~ eyesight + sex + race_eth,
              family = binomial(link = "logit"),
              data = nlsy_clean)
# poisson regression
mod_pois <- glm(nsibs ~ sleep_wkdy + sleep_wknd,
                family = poisson(link = "log"),
                data = nlsy_clean)
```

You can use the `survival` package for time-to-event models.

Quick analysis, cont.

- Create interactions with `*` (will automatically include main terms too).
- Create polynomial terms with `I(x^2)`.
- Create splines with the splines package and the `ns()` function.

```r
mod_big <- glm(log_inc ~ sex * age_bir +
                 nsibs + I(nsibs^2) +
                 ns(sleep_wkdy, knots = 3),
               family = gaussian(link = "identity"),
               data = nlsy_clean)
```

Look at results

```
summary(mod_log)
```

```
##
## Call:
## glm(formula = glasses ~ eyesight + sex + race_eth, family = binomial(link = "logit"),
##     data = nlsy_clean)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4275  -1.0825  -0.8343   1.2221   1.6261
##
## Coefficients:
##                                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)                     -0.51260    0.06317  -8.114 4.89e-16 ***
## eyesightVery Good               -0.07920    0.05359  -1.478   0.1394
## eyesightGood                    -0.07146    0.06188  -1.155   0.2481
## eyesightFair                    -0.21488    0.09105  -2.360   0.0183 *
## eyesightPoor                     0.10558    0.18152   0.582   0.5608
## sexFemale                        0.69281    0.04493  15.420  < 2e-16 ***
## race_ethNon-Hispanic Black      -0.28460    0.06493  -4.383 1.17e-05 ***
## race_ethNon-Black, Non-Hispanic  0.28528    0.05945   4.799 1.60e-06 ***
## ---
```

Look at results

Or use the `tidy()` function from the `broom` package, which nicely summarizes all types of models.

```
# from the broom package
tidy(mod_log)
```

```
## # A tibble: 8 x 5
##   term                              estimate std.error statistic  p.value
##   <chr>                                <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)                        -0.513     0.0632    -8.11  4.89e-16
## 2 eyesightVery Good                  -0.0792    0.0536    -1.48  1.39e- 1
## 3 eyesightGood                       -0.0715    0.0619    -1.15  2.48e- 1
## 4 eyesightFair                       -0.215     0.0911    -2.36  1.83e- 2
## 5 eyesightPoor                        0.106     0.182      0.582 5.61e- 1
## 6 sexFemale                           0.693     0.0449    15.4   1.21e-53
## 7 race_ethNon-Hispanic Black         -0.285     0.0649    -4.38  1.17e- 5
## 8 race_ethNon-Black, Non-Hispanic     0.285     0.0594     4.80  1.60e- 6
```

Pull off a coefficient

```r
coef(mod_log)
```

```
##                      (Intercept)                 eyesightVery Good
##                      -0.51260479                       -0.07920222
##                     eyesightGood                      eyesightFair
##                      -0.07145996                       -0.21487546
##                     eyesightPoor                         sexFemale
##                       0.10557518                        0.69280825
##       race_ethNon-Hispanic Black race_ethNon-Black, Non-Hispanic
##                      -0.28460369                        0.28527712
```

```r
coef(mod_log)[6]
```

```
## sexFemale
## 0.6928082
```

```r
tidy(mod_log) %>% slice(6) %>% pull(estimate)
```

```
## [1] 0.6928082
```

Creating new values

But you can create new values in this dataframe!

```
res_mod_log <- mod_log %>% tidy() %>%
  mutate(lci = estimate - 1.96 * std.error,
         uci = estimate + 1.96 * std.error)
res_mod_log
```

```
## # A tibble: 8 x 7
##   term                           estimate std.error statistic  p.value    lci     uci
##   <chr>                             <dbl>     <dbl>     <dbl>    <dbl>  <dbl>   <dbl>
## 1 (Intercept)                     -0.513     0.0632    -8.11  4.89e-16 -0.636 -0.389
## 2 eyesightVery Good               -0.0792    0.0536    -1.48  1.39e- 1 -0.184  0.0258
## 3 eyesightGood                    -0.0715    0.0619    -1.15  2.48e- 1 -0.193  0.0498
## 4 eyesightFair                    -0.215     0.0911    -2.36  1.83e- 2 -0.393 -0.0364
## 5 eyesightPoor                     0.106     0.182      0.582 5.61e- 1 -0.250  0.461
## 6 sexFemale                        0.693     0.0449    15.4   1.21e-53  0.605  0.781
## 7 race_ethNon-Hispanic Black      -0.285     0.0649    -4.38  1.17e- 5 -0.412 -0.157
## 8 race_ethNon-Black, Non-Hispanic  0.285     0.0594     4.80  1.60e- 6  0.169  0.402
```

We could also clean up the `term` variable, perhaps with `fct_recode()`.

Calculating ORs

Since these are results from a logistic regression, we'll probably want to exponentiate the coefficients and their CIs.

```
res_mod_log <- res_mod_log %>% select(term, estimate, lci, uci) %>%
  filter(term != "(Intercept)") %>%
  mutate_at(vars(estimate, lci, uci), exp)
res_mod_log
```

```
## # A tibble: 7 x 4
##   term                              estimate   lci   uci
##   <chr>                                <dbl> <dbl> <dbl>
## 1 eyesightVery Good                    0.924 0.832 1.03
## 2 eyesightGood                         0.931 0.825 1.05
## 3 eyesightFair                         0.807 0.675 0.964
## 4 eyesightPoor                         1.11  0.779 1.59
## 5 sexFemale                            2.00  1.83  2.18
## 6 race_ethNon-Hispanic Black           0.752 0.662 0.854
## 7 race_ethNon-Black, Non-Hispanic      1.33  1.18  1.49
```

Confidence intervals with `str_glue()`

Now we want to combine the lower and upper CI limits.

```
res_mod_log %>% select(term, estimate, lci, uci) %>%
  filter(term != "(Intercept)") %>%
  mutate(ci = str_glue("({lci}, {uci})"))
```

```
## # A tibble: 7 x 5
##   term                              estimate   lci   uci ci
##   <chr>                                <dbl> <dbl> <dbl> <glue>
## 1 eyesightVery Good                    0.924 0.832 1.03  (0.831734362089001, 1.02617441582346)
## 2 eyesightGood                         0.931 0.825 1.05  (0.824698936937584, 1.05107868439189)
## 3 eyesightFair                         0.807 0.675 0.964 (0.674797666860532, 0.96424628337116)
## 4 eyesightPoor                         1.11  0.779 1.59  (0.778639992096712, 1.58622477434497)
## 5 sexFemale                            2.00  1.83  2.18  (1.8307856398006, 2.18337382956281)
## 6 race_ethNon-Hispanic Black           0.752 0.662 0.854 (0.662416384318316, 0.85440801346314)
## 7 race_ethNon-Black, Non-Hispanic      1.33  1.18  1.49  (1.18383962963298, 1.49449918933821)
```

We can paste text and R code together with `str_glue()`. Everything goes in quotation marks. R code to be evaluated goes in `{}`.

- Which means we can use the `round()` function within the curly braces too!

More `str_glue()`

You can paste any R expression you want evaluated in the curly braces.

You can break up chunks of your string to make it easier to read in your code.

```
str_glue(
  "The intercept from the regression is ",
  "{round(coef(lm(income~sex, data = nlsy_clean))[1])} and a random ",
  "number that I generated is {round(rnorm(1, 0, 1), 3)}."
)
```

```
## The intercept from the regression is 14880 and a random number that I generated is -1.116.
```

More functions are available in the `glue` package. For example, you could make the right-hand side of a model like this:

```
glue::glue_collapse(
  c("age_bir", "sex", "nsibs", "race_eth"),
  sep = " + "
)
```

```
## age_bir + sex + nsibs + race_eth
```

Better: Create a function

We want to take these values and print "OR (95% CI LCI, UCI)" for each one. Let's make a function to put together everything we've done so far!

```
ci_func <- function(estimate, lci, uci) {
  OR <- round(exp(estimate), 2)
  lci <- round(exp(lci), 2)
  uci <- round(exp(uci), 2)
  to_print <- str_glue("{OR} (95% CI {lci}, {uci})")
  return(to_print)
}
```

Let's test on some made-up values:

```
ci_func(.2523421, -.142433, .851234)
```

```
## 1.29 (95% CI 0.87, 2.34)
```

From start to finish

```
new_mod <- glm(glasses ~ eyesight*sex, family = binomial(link = "logit"),
               data = nlsy_clean)
new_mod %>% tidy() %>%
  filter(term != "(Intercept)") %>%
  mutate(lci = estimate - 1.96 * std.error,
         uci = estimate + 1.96 * std.error,
         OR = ci_func(estimate, lci, uci),
         p.value = scales::pvalue(p.value)) %>%
  select(term, OR, p.value)
```

```
## # A tibble: 9 x 3
##   term                         OR                          p.value
##   <chr>                        <glue>                      <chr>
## 1 eyesightVery Good            1.07 (95% CI 0.92, 1.23) 0.392
## 2 eyesightGood                 0.97 (95% CI 0.81, 1.15) 0.702
## 3 eyesightFair                 0.73 (95% CI 0.55, 0.97) 0.029
## 4 eyesightPoor                 1.23 (95% CI 0.68, 2.2)  0.497
## 5 sexFemale                    2.37 (95% CI 2.04, 2.75) <0.001
## 6 eyesightVery Good:sexFemale 0.71 (95% CI 0.57, 0.87) 0.001
## 7 eyesightGood:sexFemale      0.83 (95% CI 0.65, 1.05) 0.121
## 8 eyesightFair:sexFemale      0.97 (95% CI 0.67, 1.39) 0.867
## 9 eyesightPoor:sexFemale      0.71 (95% CI 0.34, 1.47) 0.353
```

Final challenge

Data analysis from start to finish

1. Prepare and organize your project
2. Load and clean the data
3. Do some exploratory analysis (table 1, figure)
4. Do some regression analysis (results table, figure)

Prepare your project

- File -> New Project -> New Directory -> New Project
- Name it something like NLSY and put it in an appropriate folder on your computer
- Within that folder, make new folders as follows:

```
NLSY/
 ├─ NLSY.Rproj
 ├─ data/
 |    ├── raw/
 |    └── processed/
 ├─ code/
 └── results/
      ├── tables/
      └── figures/
```

Prepare the data

- Copy and paste `nlsy.csv` into `data/raw`.
- Create a new file and save it as `clean_data.R`.
- In that file, read in the NLSY data and load any packages you need. Make sure you replace any missing values with NA. Hint: there are extra missnig values in the `age_bir` variable. Also, the variable names might be useful:

```
colnames_nlsy <- c(
  "glasses", "eyesight", "sleep_wkdy", "sleep_wknd",
  "id", "nsibs", "samp", "race_eth", "sex", "region",
  "income", "res_1980", "res_2002", "age_bir"
)
```

- Add factor labels to `eyesight`, `sex`, `race_eth`, `region`, as in earlier slides. Select those variables plus `income`, `id`, `nsibs`, `age_bir`, and the sleep variables. Then restrict to complete cases and people with incomes < $30,000. Make a variable for the log of income (replace with NA if income <= 0).
- Also in that file, save your new dataset as a `.rds` file to the `data/processed` folder.

Do some exploratory analysis

- Create a file called `create_figure.R`. In this file, read in the cleaned dataset. Load any packages you need. Then make a `ggplot` figure of your choosing to show something about the distribution of the data. Save it to the `results/figures` folder as a `.png` file using the `ggsave()` function.
- Create a file called `table_1.R`. In this file, read in the cleaned dataset and use the `tableone` package to create a table 1 with the variables of your choosing. Modify the following code to save it as a `.csv` file. Open it in Excel/Numbers/Google Sheets/etc. to make sure it worked.

```
tab1 <- CreateTableOne(...) %>% print() %>% as_tibble(rownames = "id")
write_csv(tab1, ...)
```

Do some regression analysis

- In another file called `lin_reg.R`, read in the data and run the following linear regression: `lm(log_inc ~ age_bir + sex + race_eth + nsibs, data = nlsy)`. Modify the CI function to produce a table of results for a *linear* regression. Add an argument `digits =`, with a default of 2, to allow you to choose the number of digits you'd like. Save it in a separate file called `functions.R`. Use `source()` to read in the function at the beginning of your script.
- Save a table of your results as a `.csv` file. Make the names of the coefficients nice!
- Using the results, use `ggplot` to make a figure. Use `geom_point()` for the point estimates and `geom_errorbar()` for the confidence intervals. It will look something like this:

```
ggplot(data) +
  geom_point(aes(x = , y = )) +
  geom_errorbar(aes(x = , ymin = , ymax = ))
```

- Save that figure as a `.pdf` using `ggsave()`. You may want to play around with the `height =` and `width =` arguments to make it look like you want.

Appendix: some other packages I like but haven't mentioned

- `rmarkdown`: I write most of my documents (manuscripts, slides, homeworks) in RMarkdown. I couldn't live without it. (https://rmarkdown.rstudio.com)
- `lubridate`: Work with dates and times really easily. (https://lubridate.tidyverse.org)
- `janitor`: Helps clean variable names, etc. (http://sfirke.github.io/janitor/)
- `furrr`: Speed up your code with parallel processing. (https://davisvaughan.github.io/furrr/)
- `shiny`: Make interactive apps. I made http://selection-bias.louisahsmith.com in shiny. (http://shiny.rstudio.com)
- `drake`: Pipeline for analysis. (https://docs.ropensci.org/drake)
- `rvest`: Scrape data from websites. (https://rvest.tidyverse.org)