# Welcome to ID 543

Introduction to R

# About this class

- Quick! Intense!
  - Daily homeworks & final project
  - Use office hours! Your classmates! The internet!
  - It will require practice afterward, and time to sink in
  - The goal is to set you up for success and give you resources to learn more

> **Tip**
>
> Experiment! You are not going to break anything!

# About this class

- Everything you need is at http://id543.louisahsmith.com
  - Canvas will link you there, but good to bookmark as well
  - Everything admin/grade-related on Canvas
- General format:
  - Some overview slides
  - An example together
  - Practice on your own/with your classmates
  - Repeat

**Tip**

Try to solve a problem yourself first, classmate second, teaching team third

# Homeworks

Did you make a good-faith effort to answer the question using the tools we've covered in class?

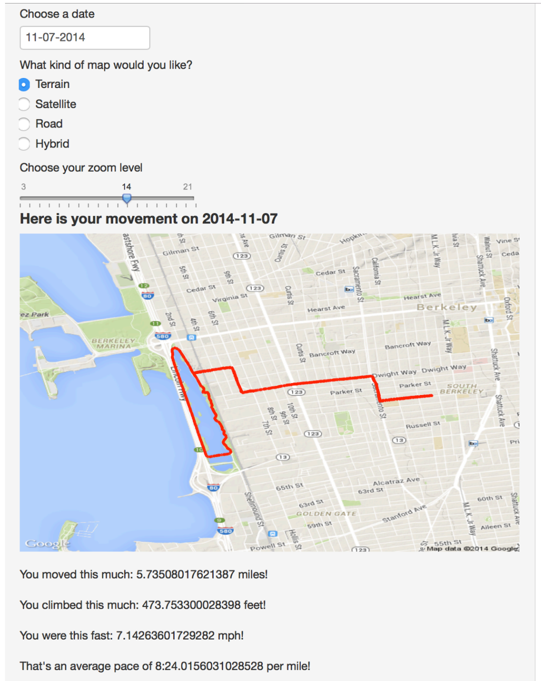
- Read the error message carefully. Check for missing/extra commas and parentheses. Restart R and reload the data.
- Go back to the slides. What were the day's goals? What were the functions we covered?
- Check out the reading – it can be good to get another perspective.
- Google using key words from the class. There are lots of ways to do things, but try to find strategies using tools we've covered in class (e.g., if you search with "tidyverse" you'll find a lot of what we cover).
- Ask a classmate how they approached it. Don't copy and paste – even if you end up writing exactly what they did, type it out yourself for practice.
- If you can't solve a problem, include code you tried and describe the strategies you used to try to solve it.

# About this class

- Day 1: dataframes and variables
- Day 2: data manipulation and management
- Day 3: models and tables
- Day 4: figures and more

# About Louisa

- Assistant professor at Northeastern University
    - Department of Public Health & Health Sciences and the Roux Institute (Portland)
- Started using R during my master's (so almost 10 years of experience)
    - Learned mostly by doing!
    - Twitter, blogs, RStudio::conf, meetups
- First iteration of this class when I was a PhD student here
- Basically everything I do is in R!



Choose a date

11-07-2014

What kind of map would you like?

● Terrain
○ Satellite
○ Road
○ Hybrid

Choose your zoom level

3 14 21

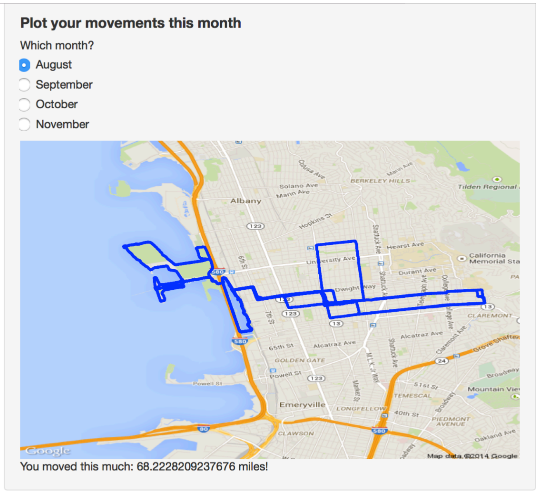
**Here is your movement on 2014-11-07**

You moved this much: 5.73508017621387 miles!

You climbed this much: 473.753300028398 feet!

You were this fast: 7.14263601729282 mph!

That's an average pace of 8:24.0156031028528 per mile!



**Plot your movements this month**

Which month?

● August
○ September
○ October
○ November

You moved this much: 68.2228209237676 miles!

# About Xiyue

## Education

- Bachelor’s Degree in Nutrition, University of Washington, Seattle
- Current MS Student in Epidemiology, Harvard University

## Previous Experience

- Research Assistant/Student Researcher at Duke Kunshan University, Tsinghua University, and Peking University
- Used STATA and R for research

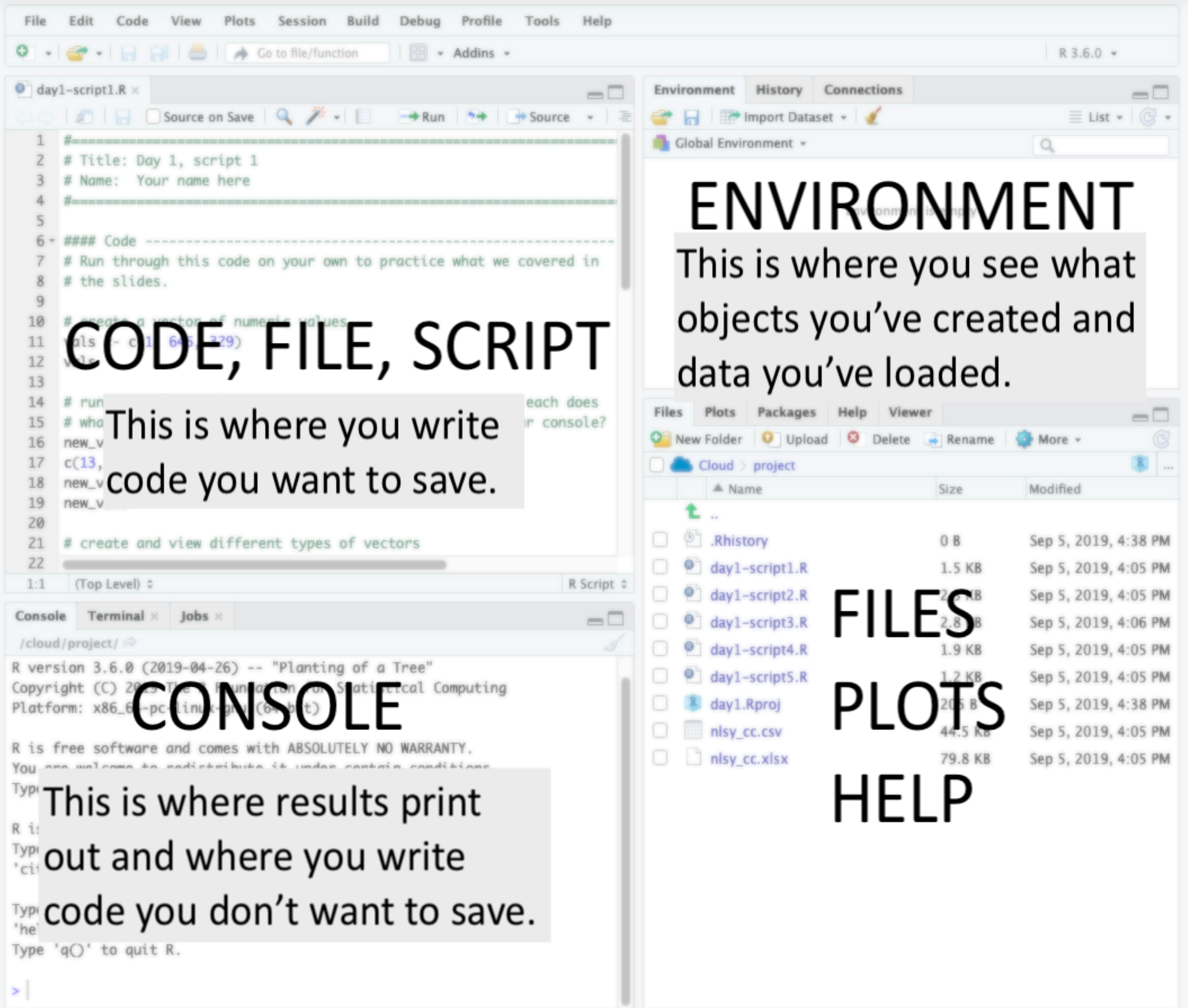
## Research Interests

- Diet and NAFLD, Liver Cancer in older adults

# Today's goals

- Familiarize yourselves with RStudio
- Introduce you to the `tidyverse` and the concept of packages
- Explore data stored in dataframes
- Create new variables
- Learn about factor variables and how to manipulate them

# RStudio



CODE, FILE, SCRIPT

This is where you write code you want to save.

```
#==============================================================================
# Title: Day 1, script 1
# Name:  Your name here
#==============================================================================

#### Code ---------------------------------------------------------------------
# Run through this code on your own to practice what we covered in
# the slides.

# create a vector of numeric values
vals <- c(1, 645, 329)
vals

# run                                                               each does
# wha                                                              r console?
new_v
c(13,
new_v
new_v

# create and view different types of vectors
```

ENVIRONMENT

This is where you see what objects you've created and data you've loaded.

FILES

PLOTS

HELP

| Name | Size | Modified |
|---|---|---|
| .. | | |
| .Rhistory | 0 B | Sep 5, 2019, 4:38 PM |
| day1-script1.R | 1.5 KB | Sep 5, 2019, 4:05 PM |
| day1-script2.R | [illegible] | Sep 5, 2019, 4:05 PM |
| day1-script3.R | 2.8 KB | Sep 5, 2019, 4:06 PM |
| day1-script4.R | 1.9 KB | Sep 5, 2019, 4:05 PM |
| day1-script5.R | 1.2 KB | Sep 5, 2019, 4:05 PM |
| day1.Rproj | 205 B | Sep 5, 2019, 4:38 PM |
| nlsy_cc.csv | 44.5 KB | Sep 5, 2019, 4:05 PM |
| nlsy_cc.xlsx | 79.8 KB | Sep 5, 2019, 4:05 PM |

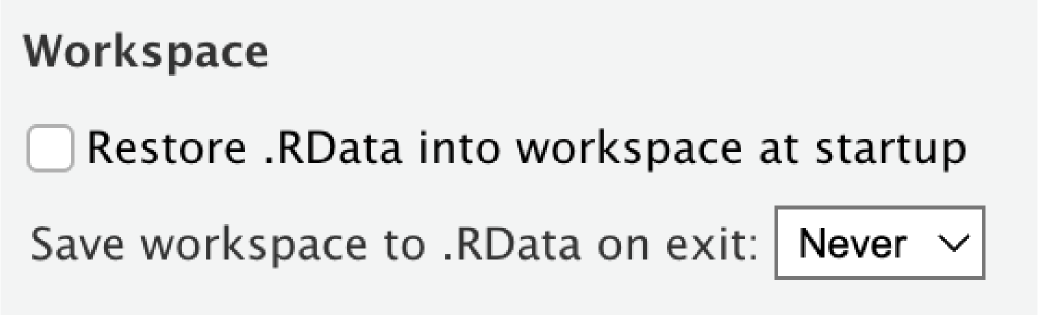
CONSOLE

This is where results print out and where you write code you don't want to save.

```
/cloud/project/
R version 3.6.0 (2019-04-26) -- "Planting of a Tree"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Typ

R i:
Typ
'ci

Typ
'he
Type 'q()' to quit R.

> 
```

# Start fresh

- If you have used R previously, an old workspace may still be active when you open RStudio
- You always want to start with a fresh session
- Go to Tools -> Global Options, and under General, change these settings:



**Workspace**

☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

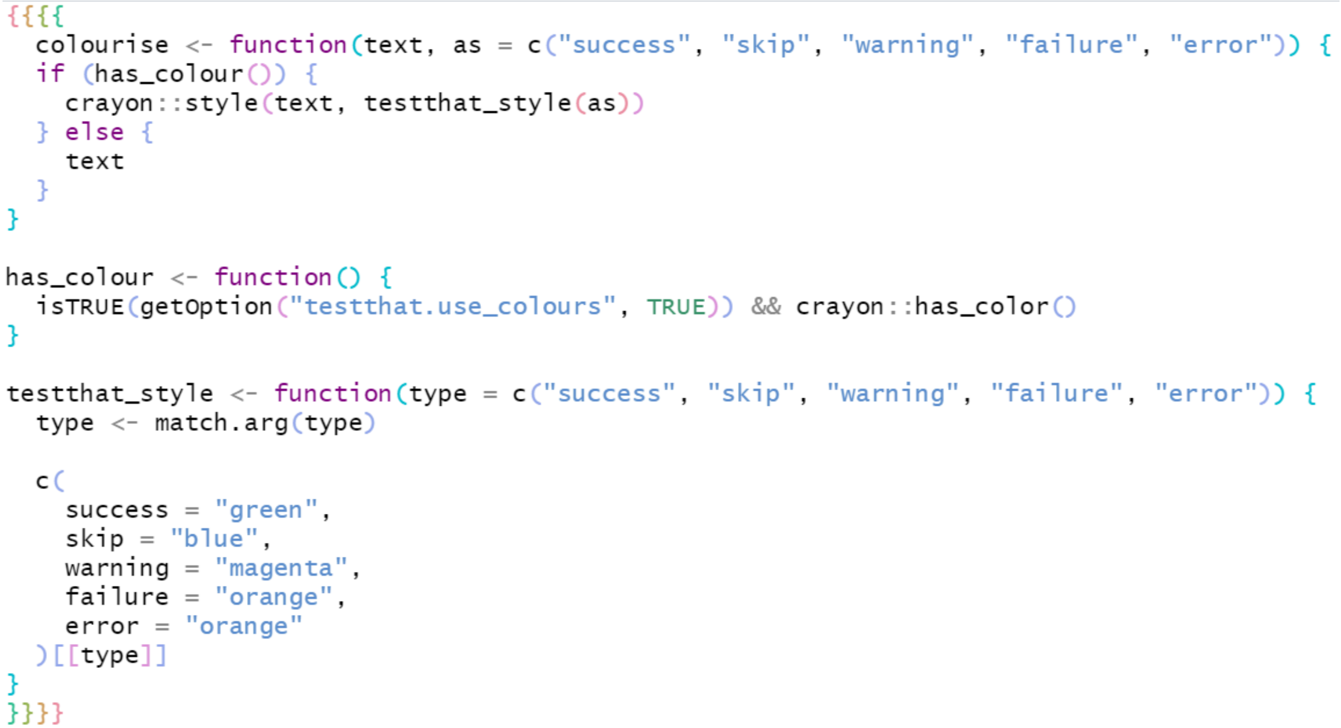
**Tip**

Now, you can just quit and restart RStudio if something goes wrong! You can also go to Session -> Restart R to clear your session.

# Rainbow parentheses

Always confirm you are closing your parentheses!

Tools -> Global Options -> Code -> Display -> Rainbow Parentheses



```
{{{{
  colourise <- function(text, as = c("success", "skip", "warning", "failure", "error")) {
  if (has_colour()) {
    crayon::style(text, testthat_style(as))
  } else {
    text
  }
}

has_colour <- function() {
  isTRUE(getOption("testthat.use_colours", TRUE)) && crayon::has_color()
}

testthat_style <- function(type = c("success", "skip", "warning", "failure", "error")) {
  type <- match.arg(type)

  c(
    success = "green",
    skip = "blue",
    warning = "magenta",
    failure = "orange",
    error = "orange"
  )[[type]]
}
}}}}
```

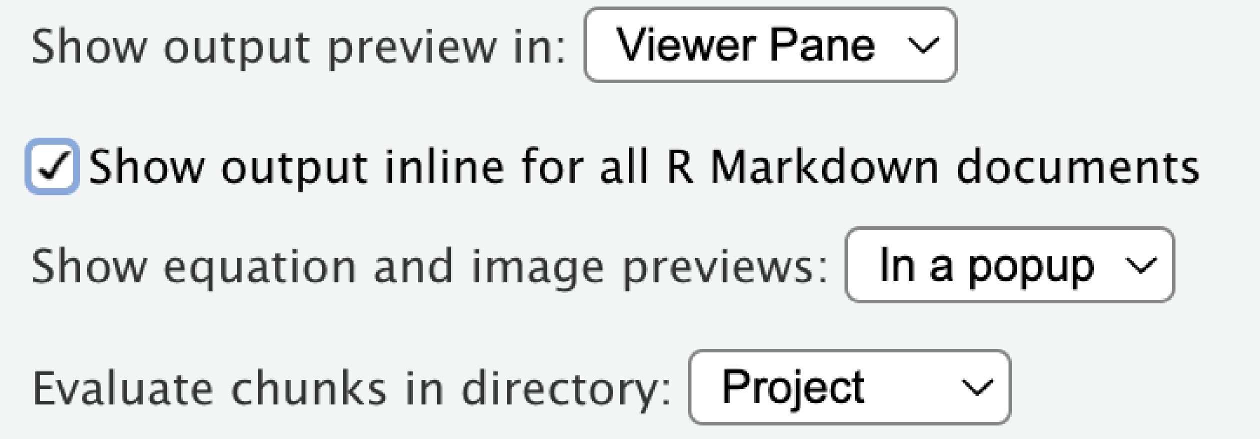
https://posit.co/blog/rstudio-1-4-preview-rainbow-parentheses/

# Print output to console

You can run…

- code that you type directly in the console
  - code you won't need to run again
- code in an `.R` script
- code in a `.qmd` (Quarto) or `.Rmd` (R Markdown) file
  - code you want to render to an html, word, or pdf file

## I like to have all code print to the console for consistency:



Show output preview in: Viewer Pane

☑ Show output inline for all R Markdown documents

Show equation and image previews: In a popup

Evaluate chunks in directory: Project

# Packages

- Some functions are built into R
  - `mean()`, `lm()`, `table()`, etc.
- They actually come from built-in packages
  - `base`, `stats`, `graphics`, etc.
- Anyone (yes, *anyone*) build their own package to add to the functionality of R
  - `{ggplot2}`, `{dplyr}`, `{data.table}`, `{survival}`, etc.

[1]

[1] Image from Zhi Yang

# Packages

- You have to **install** a package once[1]

```r
install.packages("survival")
```

- You then have to **load** the package every time you want to use it

```r
library(survival)
```

[1] Actually, with every new major R release, but we won’t worry about that.

# Packages

“You only have to buy the book once, but you have to go get it out of the bookshelf every time you want to read it.”

```r
install.packages("survival")
library(survival)
survfit(...)
```

SEVERAL DAYS LATER...

```r
library(survival)
coxph(...)
```

# Package details

- When you use `install.packages`, packages are downloaded from CRAN (The Comprehensive R Archive Network)
    - This is also where you downloaded R
- Packages can be hosted lots of other places, such as Bioconductor (for bioinformatics), and Github (for personal projects or while still developing)
- The folks at CRAN check to make things “work” in some sense, but don’t check on the statistical methods…
    - But because R is open-source, you can always read the code yourself
- Two functions from different packages can have the same name… if you load them both, you may have some trouble

# Demo

Script vs. console, installing packages, and changing settings

# The biggest difference between R and Stata is that R can have many different objects in its environment

- datasets, numbers, figures, etc.
- you have to be explicit about storing and retrieving objects
    - e.g., what dataset a variable belongs to

# R uses `<-` to store objects in the environment

I call this the “assignment arrow”

```r
# create values
vals <- c(1, 645, 329)
```

Now `vals` holds those values

> ⚠ **Warning**
>
> No assignment arrow means that the object will be printed to the console (and lost forever!)

# Objects

We can retrieve those values by running just the name of the object

```
vals
```

```
[1]   1 645 329
```

We can also perform operations on them using functions like `mean()`

```
mean(vals)
```

```
[1] 325
```

If we want to keep the result of that operation, we need to use `<-` again

```
mean_val <- mean(vals)
```

# Types of data (*classes*)

We could also create a character *vector*.

```r
chars <- c("dog", "cat", "rhino")
chars
```

```
[1] "dog"   "cat"   "rhino"
```

Or a *logical* vector:

```r
logs <- c(TRUE, FALSE, FALSE)
logs
```

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

**Note**

We'll see more options as we go along!

# Types of objects

We created *vectors* with the `c()` function (`c` stands for concatenate)

We could also create a *matrix* of values with the `matrix()` function:

```r
# turn the vector of numbers into a 2-row matrix
mat <- matrix(c(234, 7456, 12, 654, 183, 753), nrow = 2)
mat
```

```
     [,1] [,2] [,3]
[1,]  234   12  183
[2,] 7456  654  753
```

# Indices

The numbers in square brackets are *indices*, which we can use to pull out values:

```
# extract second animal
chars[2]
```

```
[1] "cat"
```

We can pull out rows or columns from matrices:

```
# extract second row
mat[2, ]
```

```
[1] 7456  654  753
```

```
# extract first column
mat[, 1]
```

```
[1]  234 7456
```

# Exercise

Pre-class challenges

# Dataframes

- We usually do analysis in R with dataframes (or some variant)
- Dataframes basically work like spreadsheets: here, columns are variables, and rows are observations
- Here's some data from the National Longitudinal Survey of Youth:

```r
nlsy
```

```
# A tibble: 1,205 × 15
      id glasses eyesight sleep_wkdy sleep_wknd nsibs race_eth   sex region
   <dbl>   <dbl>    <dbl>      <dbl>      <dbl> <dbl>    <dbl> <dbl>  <dbl>
 1     3       0        1          5          7     3        3     2      1
 2     6       1        2          6          7     1        3     1      1
 3     8       0        2          7          9     7        3     2      1
 4    16       1        3          6          7     3        3     2      1
 5    18       0        3         10         10     2        3     1      3
 6    20       1        2          7          8     2        3     2      1
 7    27       0        1          8          8     1        3     2      1
 8    49       1        1          8          8     6        3     2      1
```

# New function: `glimpse()`

We can get a quick overview of the data with the `glimpse()` function:

```r
glimpse(nlsy)
```

```
Rows: 1,205
Columns: 15
$ id          <dbl> 3, 6, 8, 16, 18, 20, 27, 49, 57, 67, 86, 96, 97, 98, 117,…
$ glasses     <dbl> 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, …
$ eyesight    <dbl> 1, 2, 2, 3, 3, 2, 1, 1, 2, 1, 3, 5, 1, 1, 1, 1, 3, 2, 3, …
$ sleep_wkdy  <dbl> 5, 6, 7, 6, 10, 7, 8, 8, 7, 8, 8, 7, 7, 7, 8, 7, 7, 8, 8,…
$ sleep_wknd  <dbl> 7, 7, 9, 7, 10, 8, 8, 8, 8, 8, 8, 7, 8, 7, 8, 7, 4, 8, 8,…
$ nsibs       <dbl> 3, 1, 7, 3, 2, 2, 1, 6, 1, 1, 7, 2, 7, 2, 2, 4, 9, 2, 2, …
$ race_eth    <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3, 3, 3, 3, 3, …
$ sex         <dbl> 2, 1, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, …
$ region      <dbl> 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
$ income      <dbl> 22390, 35000, 7227, 48000, 4510, 50000, 20000, 23900, 232…
$ age_bir     <dbl> 19, 30, 17, 31, 19, 30, 27, 24, 21, 36, 17, 19, 29, 30, 2…
$ eyesight_cat <fct> Excellent, Very Good, Very Good, Good, Good, Very Good, E…
$ glasses_cat  <fct> Doesn't wear glasses, Wears glasses/contacts, Doesn't wea…
$ race_eth_cat <fct> "Non-Black, Non-Hispanic", "Non-Black, Non-Hispanic", "No…
$ sex_cat     <fct> Female, Male, Female, Female, Male, Female, Female, Femal…
```

**Note**

Notice that I write a function name followed by parentheses to signal it is a function, and can take *arguments* within the parentheses

# New function: `summary()`

We can also get a summary of the data with the `summary()` function:

```r
summary(nlsy)
```

```
      id            glasses          eyesight      sleep_wkdy
Min.   :    3   Min.   :0.0000   Min.   :1.00   Min.   : 0.000
1st Qu.: 2317   1st Qu.:0.0000   1st Qu.:1.00   1st Qu.: 6.000
Median : 4744   Median :1.0000   Median :2.00   Median : 7.000
Mean   : 5229   Mean   :0.5178   Mean   :1.99   Mean   : 6.643
3rd Qu.: 7937   3rd Qu.:1.0000   3rd Qu.:3.00   3rd Qu.: 8.000
Max.   :12667   Max.   :1.0000   Max.   :5.00   Max.   :13.000
  sleep_wknd         nsibs           race_eth          sex
Min.   : 0.000   Min.   : 0.000   Min.   :1.000   Min.   :1.000
1st Qu.: 6.000   1st Qu.: 2.000   1st Qu.:2.000   1st Qu.:1.000
Median : 7.000   Median : 3.000   Median :3.000   Median :2.000
Mean   : 7.267   Mean   : 3.937   Mean   :2.395   Mean   :1.584
3rd Qu.: 8.000   3rd Qu.: 5.000   3rd Qu.:3.000   3rd Qu.:2.000
Max.   :14.000   Max.   :16.000   Max.   :3.000   Max.   :2.000
    region          income          age_bir         eyesight_cat
Min.   :1.000   Min.   :    0   Min.   :13.00   Excellent:474
1st Qu.:2.000   1st Qu.: 6000   1st Qu.:19.00   Very Good:385
Median :3.000   Median :11155   Median :22.00   Good     :249
Mean   :2.593   Mean   :15289   Mean   :23.45   Fair     : 78
3rd Qu.:3.000   3rd Qu.:20000   3rd Qu.:27.00   Poor     : 19
Max.   :4.000   Max.   :75001   Max.   :52.00
                glasses_cat                   race_eth_cat   sex_cat
Doesn't wear glasses  :581   Hispanic               :211   Male  :501
Wears glasses/contacts:624   Black                  :307   Female:704
                             Non-Black, Non-Hispanic:687
```

# Indices in dataframes

We can pull out data from dataframes using the “square bracket notation” we already saw:

```r
nlsy[3, ]
```

```
# A tibble: 1 × 15
     id glasses eyesight sleep_wkdy sleep_wknd nsibs race_eth   sex region
  <dbl>   <dbl>    <dbl>      <dbl>      <dbl> <dbl>    <dbl> <dbl>  <dbl>
1     8       0        2          7          9     7        3     2      1
# ℹ 6 more variables: income <dbl>, age_bir <dbl>, eyesight_cat <fct>,
#   glasses_cat <fct>, race_eth_cat <fct>, sex_cat <fct>
```

```r
nlsy[, 3]
```

```
# A tibble: 1,205 × 1
   eyesight
      <dbl>
 1        1
 2        2
 3        2
 4        3
 5        3
 6        2
 7        1
```

# Dollar sign notation

It’s much more useful to be able to pull out a variable by its name, though:

```r
nlsy$sex_cat
```

```
  [1] Female Male   Female Female Male   Female Female Female Female Male  
 [11] Female Female Female Female Male   Female Female Female Female Female
 [21] Female Male   Female Female Female Male   Female Male   Female Female
 [31] Male   Male   Male   Female Female Male   Female Female Male   Female
 [41] Female Male   Female Male   Female Male   Male   Female Male   Male  
 [51] Male   Female Female Female Male   Female Male   Female Male   Male  
 [61] Male   Female Male   Female Female Male   Male   Female Female Male  
 [71] Female Male   Male   Male   Female Male   Male   Male   Male   Female
 [81] Female Female Female Female Female Female Male   Female Male   Female
 [91] Male   Female Female Female Female Female Female Female Male   Female
[101] Female Female Female Female Male   Male   Female Male   Male   Female
[111] Female Male   Male   Male   Male   Female Male   Male   Male   Male  
[121] Female Female Male   Female Female Female Female Female Male   Male  
[131] Female Female Male   Female Female Female Female Female Male   Female
[141] Male   Male   Female Female Male   Male   Female Female Female Female
[151] Female Female Female Female Male   Female Female Male   Male   Male  
[161] Female Male   Female Female Male   Female Female Female Female Male  
[171] Male   Female Female Male   Female Female Female Female Female Female
[181] Female Male   Male   Female Male   Female Male   Female Female Female
[191] Male   Female Female Female Female Male   Female Male   Male   Male  
```

# Summarize a single variable

We can also get a summary of a single variable:

```r
summary(nlsy$sex_cat)
```

```
Male Female
 501    704
```

```r
summary(nlsy$income)
```

```
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0    6000   11155   15289   20000   75001
```

# Variables

- Variables can be different types, including numeric, character, logical, and factor.
- You can check what type of variable you're dealing with: `class(nlsy$sex_cat)` (factor!)
- A special type of dataframe called a "tibble" will show you at the top:

```r
nlsy
```

```
# A tibble: 1,205 × 15
      id glasses eyesight sleep_wkdy sleep_wknd nsibs race_eth   sex region
   <dbl>   <dbl>    <dbl>      <dbl>      <dbl> <dbl>    <dbl> <dbl>  <dbl>
 1     3       0        1          5          7     3        3     2      1
 2     6       1        2          6          7     1        3     1      1
 3     8       0        2          7          9     7        3     2      1
 4    16       1        3          6          7     3        3     2      1
 5    18       0        3         10         10     2        3     1      3
 6    20       1        2          7          8     2        3     2      1
 7    27       0        1          8          8     1        3     2      1
 8    49       1        1          8          8     6        3     2      1
```

TIBBLE

# tibbles are basically just pretty dataframes

```r
as_tibble(nlsy)[, 1:4]
```

```
# A tibble: 1,205 × 4
      id glasses eyesight sleep_wkdy
   <dbl>   <dbl>    <dbl>      <dbl>
 1     3       0        1          5
 2     6       1        2          6
 3     8       0        2          7
 4    16       1        3          6
 5    18       0        3         10
 6    20       1        2          7
 7    27       0        1          8
 8    49       1        1          8
 9    57       1        2          7
10    67       0        1          8
# ℹ 1,195 more rows
```

```r
as.data.frame(nlsy)[, 1:4]
```

```
        id glasses eyesight sleep_wkdy
1        3       0        1          5
2        6       1        2          6
3        8       0        2          7
4       16       1        3          6
5       18       0        3         10
6       20       1        2          7
7       27       0        1          8
8       49       1        1          8
9       57       1        2          7
10      67       0        1          8
11      86       0        3          8
12      96       1        5          7
13      97       1        1          7
14      98       0        1          7
15     117       0        1          8
16     137       0        1          7
17     172       0        3          7
18     179       1        2          8
19     186       1        3          8
20     200       1        3          8
21     205       0        4          7
22     218       1        2          6
23     227       0        2          8
24     237       0        5          7
```

# Different ways to do the same thing

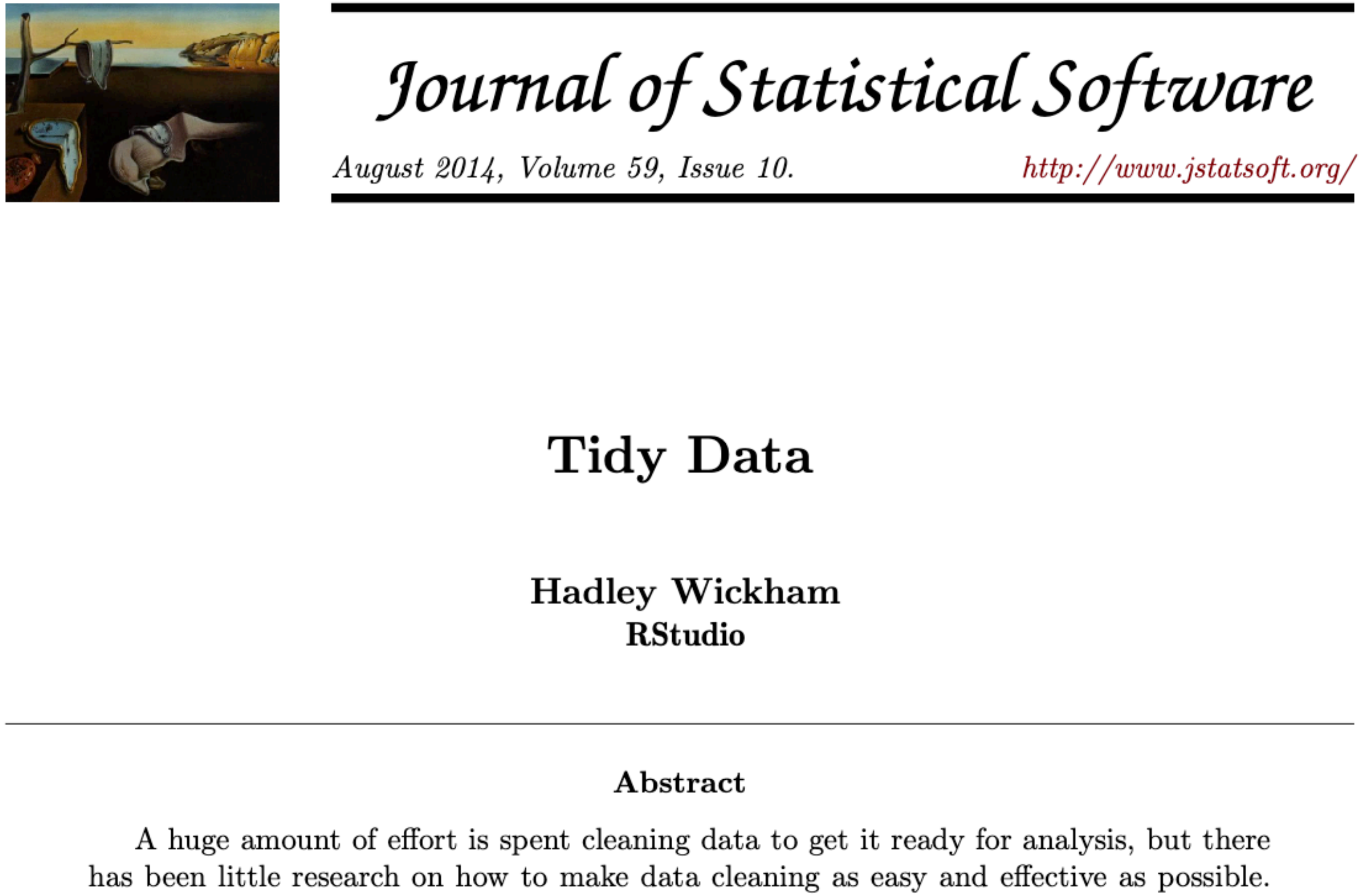
There are usually multiple ways to achieve a task in R. Ideally we'd like solutions that are:

- **readable**: If you share your code with someone, can they figure out what you're doing?
- **reliable**: Is this way always going to work, even if the data is slightly different?
- **safe**: Is this way going to introduce errors into your code without you noticing?
- **fast**: Is this an efficient way to do things, given all of the above?

We'll focus on the *tidyverse* because I think it's the optimal mix of those characteristics

# tidyverse

The same people who make RStudio also are responsible for a set of packages called the `tidyverse`



Journal of Statistical Software

August 2014, Volume 59, Issue 10. http://www.jstatsoft.org/

## Tidy Data

**Hadley Wickham**
**RStudio**

**Abstract**

A huge amount of effort is spent cleaning data to get it ready for analysis, but there has been little research on how to make data cleaning as easy and effective as possible.

# tidyverse

- `install.packages("tidyverse")` actually downloads more than a dozen packages[1]
- `library(tidyverse)` loads:

`ggplot2`, `dplyr`, `tidyr`, `readr`, `purrr`, `tibble`, `stringr`, `forcats`, `lubridate`

This is by no means the only way to manage your data, but I find that a lot of the time, it's the easiest and simplest way to get things done.

[1] See which ones at https://tidyverse.tidyverse.org

# Exercise

Intro to dataframes

# Creating variables

Two (of several) ways to take the (natural) log of income and store it in the dataframe:

```r
nlsy$log_income <- log(nlsy$income)
```

*OR*

```r
nlsy <- mutate(nlsy,
               log_income = log(income))
```

> **Note**
>
> The second way may look longer now, but we’ll see later why it’s useful when we make lots of variables at once!

# New function: Creating a new variable with `mutate()`

General format:

```r
dataframe <- mutate(dataframe,
                    new_variable = function(old_variable))
```

We can do whatever we want to a variable to make a new one:

```r
nlsy <- mutate(nlsy,
               new_id = id + 1)
```

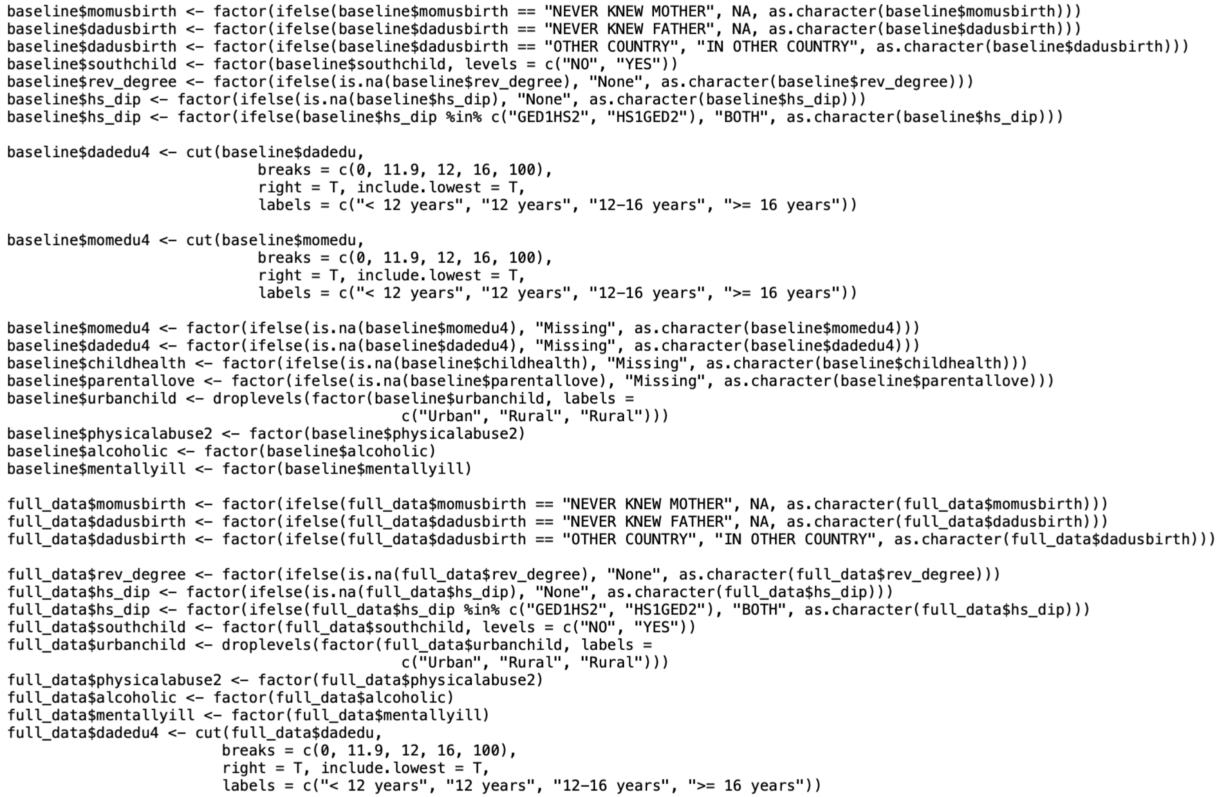
**Tip**

`mutate()` is a function that acts on a dataframe, so when we use the assignment arrow, it’s to store the dataframe with the new variable back in the same place

# Making variables in “Base R”

```r
nlsy$region_cat <- factor(nlsy$region)
nlsy$income <- round(nlsy$income)
nlsy$age_bir_cent <- nlsy$age_bir - mean(nlsy$age_bir)
nlsy$index <- 1:nrow(nlsy)
nlsy$slp_wkdy_cat <- ifelse(nlsy$sleep_wkdy < 5, "little",
                            ifelse(nlsy$sleep_wkdy < 7, "some",
                                   ifelse(nlsy$sleep_wkdy < 9, "i
                                          ifelse(nlsy$sleep_wkdy <
                                   )
                            )
)
```

# Very quickly your code can get overrun with dollar signs (and parentheses, and arrows)



```r
baseline$momusbirth <- factor(ifelse(baseline$momusbirth == "NEVER KNEW MOTHER", NA, as.character(baseline$momusbirth)))
baseline$dadusbirth <- factor(ifelse(baseline$dadusbirth == "NEVER KNEW FATHER", NA, as.character(baseline$dadusbirth)))
baseline$dadusbirth <- factor(ifelse(baseline$dadusbirth == "OTHER COUNTRY", "IN OTHER COUNTRY", as.character(baseline$dadusbirth)))
baseline$southchild <- factor(baseline$southchild, levels = c("NO", "YES"))
baseline$rev_degree <- factor(ifelse(is.na(baseline$rev_degree), "None", as.character(baseline$rev_degree)))
baseline$hs_dip <- factor(ifelse(is.na(baseline$hs_dip), "None", as.character(baseline$hs_dip)))
baseline$hs_dip <- factor(ifelse(baseline$hs_dip %in% c("GED1HS2", "HS1GED2"), "BOTH", as.character(baseline$hs_dip)))

baseline$dadedu4 <- cut(baseline$dadedu,
                            breaks = c(0, 11.9, 12, 16, 100),
                            right = T, include.lowest = T,
                            labels = c("< 12 years", "12 years", "12-16 years", ">= 16 years"))

baseline$momedu4 <- cut(baseline$momedu,
                            breaks = c(0, 11.9, 12, 16, 100),
                            right = T, include.lowest = T,
                            labels = c("< 12 years", "12 years", "12-16 years", ">= 16 years"))

baseline$momedu4 <- factor(ifelse(is.na(baseline$momedu4), "Missing", as.character(baseline$momedu4)))
baseline$dadedu4 <- factor(ifelse(is.na(baseline$dadedu4), "Missing", as.character(baseline$dadedu4)))
baseline$childhealth <- factor(ifelse(is.na(baseline$childhealth), "Missing", as.character(baseline$childhealth)))
baseline$parentallove <- factor(ifelse(is.na(baseline$parentallove), "Missing", as.character(baseline$parentallove)))
baseline$urbanchild <- droplevels(factor(baseline$urbanchild, labels =
                                            c("Urban", "Rural", "Rural")))
baseline$physicalabuse2 <- factor(baseline$physicalabuse2)
baseline$alcoholic <- factor(baseline$alcoholic)
baseline$mentallyill <- factor(baseline$mentallyill)

full_data$momusbirth <- factor(ifelse(full_data$momusbirth == "NEVER KNEW MOTHER", NA, as.character(full_data$momusbirth)))
full_data$dadusbirth <- factor(ifelse(full_data$dadusbirth == "NEVER KNEW FATHER", NA, as.character(full_data$dadusbirth)))
full_data$dadusbirth <- factor(ifelse(full_data$dadusbirth == "OTHER COUNTRY", "IN OTHER COUNTRY", as.character(full_data$dadusbirth)))

full_data$rev_degree <- factor(ifelse(is.na(full_data$rev_degree), "None", as.character(full_data$rev_degree)))
full_data$hs_dip <- factor(ifelse(is.na(full_data$hs_dip), "None", as.character(full_data$hs_dip)))
full_data$hs_dip <- factor(ifelse(full_data$hs_dip %in% c("GED1HS2", "HS1GED2"), "BOTH", as.character(full_data$hs_dip)))
full_data$southchild <- factor(full_data$southchild, levels = c("NO", "YES"))
full_data$urbanchild <- droplevels(factor(full_data$urbanchild, labels =
                                            c("Urban", "Rural", "Rural")))
full_data$physicalabuse2 <- factor(full_data$physicalabuse2)
full_data$alcoholic <- factor(full_data$alcoholic)
full_data$mentallyill <- factor(full_data$mentallyill)
full_data$dadedu4 <- cut(full_data$dadedu,
                        breaks = c(0, 11.9, 12, 16, 100),
                        right = T, include.lowest = T,
                        labels = c("< 12 years", "12 years", "12-16 years", ">= 16 years"))
```

# Cleaner way to make lots of new variables

```r
nlsy <- mutate(nlsy, # dataset
      # new variables
      region_cat = factor(region, labels = c("Northeast", "North Ce
      income = round(income),
      age_bir_cent = age_bir - mean(age_bir),
      index = row_number() # a special function that gives the row
      # could make as many as we want....
)
```

**Tip**

We can refer to variables within the same dataset (`region`, `income`, `age_bir`) without the `$` notation

# `mutate()` tips and tricks

You still need to store your dataset somewhere, so make sure to include the assignment arrow

- Good practice to make new copies with different names as you go along

```r
nlsy_w_cats <- mutate(nlsy, # dataset
               region_cat = factor(region),
               sex_cat = factor(sex),
               race_eth_cat = factor(race_eth))

nlsy_clean <- mutate(nlsy_w_cats, # dataset
                     region_cat = fct_recode(region_cat,
                                             "Northeast" = "1",
                                             "North Central" = "2",
                                             "South" = "3",
                                             "West" = "4"),
                     sex_cat = fct_relevel(sex_cat,
                                           "Female", "Male"))
```

# `mutate()` tips and tricks

- You can refer immediately to variables you just made:

```r
nlsy_new <- mutate(nlsy,
                   age_bir_cent = age_bir - mean(age_bir),
                   age_bir_stand = age_bir_cent / sd(age_bir_cent)
)
```

**Tip**

"Chunk" your work on the same/similar variables so you can keep track of how a variable is derived.

# Exercise

Making variables

# Factor variables

When I downloaded the data originally, it was all numeric ("double")

I already converted some variables into categorical ("factor") variables (using the codebook)

- factors have *levels*
- the first level is the *reference level* when you include it in a regression

# New function: `count()`

We can explore factor variables (and other types!) using `count()`:

```r
count(nlsy, glasses_cat)
```

```
# A tibble: 2 × 2
  glasses_cat                 n
  <fct>                   <int>
1 Doesn't wear glasses      581
2 Wears glasses/contacts    624
```

### Tip

Like `mutate()`, this function takes a dataframe as its first argument. The second argument is the variable you want to count.

# Cross-tabulations

Actually, `count()` can take a whole series of variable names:

```r
count(nlsy, glasses_cat, sex_cat)
```

```
# A tibble: 4 × 3
  glasses_cat              sex_cat     n
  <fct>                    <fct>   <int>
1 Doesn't wear glasses     Male      280
2 Doesn't wear glasses     Female    301
3 Wears glasses/contacts Male      221
4 Wears glasses/contacts Female    403
```

**Note**

If this isn't in the format you want your cross-tab in, don't worry – we'll see other funtions that make better tables later. This output is handy though, because it's a dataframe! (Actually, a tibble!)

# New function: converting a variable with `factor()`

Again, two ways of doing the same thing:

```r
nlsy$region_cat <- factor(nlsy$region)
```

*OR*

```r
nlsy <- mutate(nlsy,
               region_cat = factor(region))
```

# The `factor()` function does nothing to the names of the values

```r
nlsy <- mutate(nlsy,
               region_cat = factor(region))
class(nlsy$region_cat)
```

```
[1] "factor"
```

```r
levels(nlsy$region_cat)
```

```
[1] "1" "2" "3" "4"
```

⚠ **Warning**

The levels will be in numeric order, or alphabetical order if a character variable. This means that `factor(c(1, 2, ..., 10))` will have a different ordering than `factor(c("1", "2", ..., "10"))`.

# We can assign names to the values

```r
nlsy <- mutate(nlsy,
               region_cat = factor(region,
                                   levels = c(1, 2, 3, 4),
                                   labels = c("Northeast",
                                   "North Central", "South",
                                   "West")))
```

**Warning**

Make sure the order of the `levels =` and `labels =` arguments always match!

# It’s always good practice to confirm everything looks right

```r
count(nlsy, region_cat, region)
```

```
# A tibble: 4 × 3
  region_cat    region     n
  <fct>          <dbl> <int>
1 Northeast          1   206
2 North Central      2   333
3 South              3   411
4 West               4   255
```

# Exercise

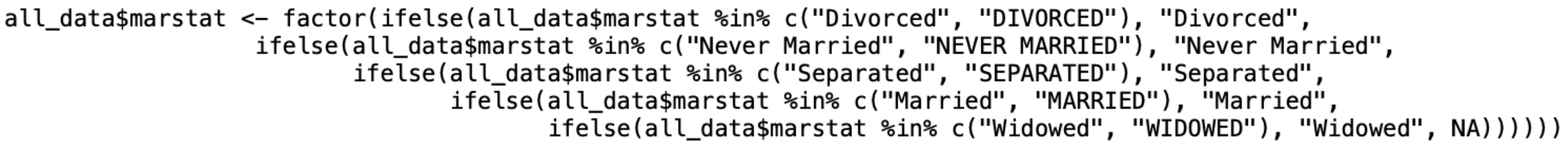
Intro to factors

# My favorite R function: `case_when()`

I used to write endless strings of `ifelse()` statements

- If A is TRUE, then B; if not, then if C is true, then D; if not, then if E is true, then F; if not, ...

```r
nlsy <- mutate(nlsy,
               ifelse(sleep_wkdy < 5,  "little",
                      ifelse(sleep_wkdy < 7, "some",
                             ifelse(sleep_wkdy < 9, "ideal",
                                    ifelse(sleep_wkdy < 12, "lots'
```



```r
all_data$marstat <- factor(ifelse(all_data$marstat %in% c("Divorced", "DIVORCED"), "Divorced",
                  ifelse(all_data$marstat %in% c("Never Married", "NEVER MARRIED"), "Never Married",
                         ifelse(all_data$marstat %in% c("Separated", "SEPARATED"), "Separated",
                                ifelse(all_data$marstat %in% c("Married", "MARRIED"), "Married",
                                       ifelse(all_data$marstat %in% c("Widowed", "WIDOWED"), "Widowed", NA))))))
```

This can be extremely hard to follow!

# `case_when()` syntax

- Ask a question (i.e., something that will give `TRUE` or `FALSE`) on the left-hand side of a `~`
- `sleep_wkdy < 5 ~`
- If `TRUE`, variable will take on value of whatever is on the right-hand side of the `~`
- `~ "little"`
- Proceeds in order … if TRUE, takes that value and **stops**
- If you want some default value, you can end with `.default = {something}`, which every observation will get if everything else is `FALSE`
- `.default = NA` is the default default

# Logicals: answers to TRUE/FALSE questions

When we want to know if something is

- equal: `==`
- not equal: `!=`
- greater than or equal to: `>=`
- less than or equal to: `<=`

We also can ask about multiple conditions with `&` (and) and `|` (or).

# `case_when()` combines a lot of “if-else” statements

```r
nlsy <- mutate(nlsy, slp_cat_wkdy =
                 case_when(sleep_wkdy < 5 ~ "little",
                           sleep_wkdy < 7 ~ "some",
                           sleep_wkdy < 9 ~ "ideal",
                           sleep_wkdy < 12 ~ "lots",
                           .default = NA
                 )
)

count(nlsy, sleep_wkdy, slp_cat_wkdy)
```

```
# A tibble: 13 × 3
   sleep_wkdy slp_cat_wkdy     n
        <dbl> <chr>        <int>
 1          0 little           1
 2          2 little           4
 3          3 little          14
 4          4 little          48
```

# `case_when()` example

```r
nlsy <- mutate(nlsy, total_sleep =
                 case_when(
                   sleep_wknd > 8 & sleep_wkdy > 8 ~ 1,
                   sleep_wknd + sleep_wkdy > 15 ~ 2,
                   sleep_wknd - sleep_wkdy > 3 ~ 3
                 )
)
```

- Which value would someone with `sleep_wknd = 8` and `sleep_wkdy = 4` go?
- What about someone with `sleep_wknd = 11` and `sleep_wkdy = 4`?
- What about someone with `sleep_wknd = 7` and `sleep_wkdy = 7`?

# Creating a factor variable from a character variable after using `case_when()`

```r
nlsy <- mutate(nlsy, slp_chr_wkdy =
                 case_when(
                   sleep_wkdy < 5 ~ "little",
                   sleep_wkdy < 7 ~ "some",
                   sleep_wkdy < 9 ~ "ideal",
                   sleep_wkdy < 12 ~ "lots"
                 ),
               slp_cat_wkdy = factor(slp_chr_wkdy)
)
```

What order will these levels be in?

# Side note: another way to look at factors

In the next few slides, I'll use the `summary()` function (rather than `count()`) to look at factors

- It's easier to fit the output on slides
- However, it doesn't show anything interesting for *character* variables so I usually prefer `count()`, which does

```r
summary(nlsy$slp_chr_wkdy)
```

```
  Length     Class      Mode
    1205 character character
```

```r
summary(nlsy$slp_cat_wkdy)
```

```
ideal little   lots   some   NA's
  626     67     47    462      3
```

# `forcats` package

- Tries to make working with factors safe and convenient
- Functions to make new levels, reorder levels, combine levels, etc.
- All the functions start with `fct_` so they're easy to find using tab-complete!
- Automatically loads with `library(tidyverse)`

# Reorder factors

The `fct_relevel()` function allows us just to rewrite the names of the categories out in the order we want them (safely).

```r
nlsy <- mutate(nlsy,
               slp_cat_wkdy_ord = fct_relevel(slp_cat_wkdy,
                                              "little",
                                              "some",
                                              "ideal",
                                              "lots"
               )
)

summary(nlsy$slp_cat_wkdy_ord)
```

```
little   some  ideal   lots   NA's
    67    462    626     47      3
```

# What if you misspell something?

```r
nlsy <- mutate(nlsy,
               slp_cat_wkdy_ord2 = fct_relevel(slp_cat_wkdy,
                                               "little",
                                               "soome",
                                               "ideal",
                                               "lots"))
```

```
Warning: There was 1 warning in `mutate()`.
ℹ In argument: `slp_cat_wkdy_ord2 = fct_relevel(slp_cat_wkdy, "little",
  "soome", "ideal", "lots")`.
Caused by warning:
! 1 unknown level in `f`: soome
```

```r
summary(nlsy$slp_cat_wkdy_ord2)
```

```
little  ideal   lots   some   NA's 
    67    626     47    462      3 
```

You get a warning, and levels you didn't mention are pushed to the end.

# Recode a factor

```r
nlsy <- mutate(nlsy,
               region_cat2 = fct_recode(region_cat,
                                        "NE" = "Northeast",
                                        "NC" = "North Central",
                                        "S" = "South",
                                        "W" = "West"))
summary(nlsy$region_cat2)
```

```
 NE  NC   S   W
206 333 411 255
```

# Other orders

## How about from most people to least?

```r
nlsy <- mutate(nlsy, region_cat = fct_infreq(region_cat))
summary(nlsy$region_cat)
```

```
        South North Central          West     Northeast 
          411           333           255           206 
```

## Or the reverse of that?

```r
nlsy <- mutate(nlsy, region_cat = fct_rev(region_cat))
summary(nlsy$region_cat)
```

```
    Northeast          West North Central         South 
          206           255           333           411 
```

**Tip**

This will be handy when running regressions and creating graphs.

# Add levels

We have some missing values – let’s say we want to include them as a group in a table, figure, or regression.

```r
nlsy <- mutate(nlsy, slp_cat_wkdy_out =
                    fct_na_value_to_level(slp_cat_wkdy, level = "out
summary(nlsy$slp_cat_wkdy_out)
```

```
  ideal  little    lots    some outlier 
    626      67      47     462       3 
```

# Remove levels

Or maybe we want to combine some levels that don’t have a lot of observations in them:

```r
nlsy <- mutate(nlsy, slp_cat_wkdy_comb =
                 fct_collapse(slp_cat_wkdy,
                              "less" = c("little", "some"),
                              "more" = c("ideal", "lots")
)
)
summary(nlsy$slp_cat_wkdy_comb)
```

```
more less NA's
 673  529    3
```

# Add and remove

Or we can have R choose which ones to combine based on how few observations they have:

```
nlsy <- mutate(nlsy, slp_cat_wkdy_lump =
                     fct_lump(slp_cat_wkdy, n = 2))
summary(nlsy$slp_cat_wkdy_lump)
```

```
ideal  some Other  NA's 
  626   462   114     3 
```

- Probably not a good idea for factors with an inherent order

# There are 25 `fct_` functions in the package. The sky's the limit when it comes to manipulating your categorical variables in R!

I never remember all of them – the goal is not for you to either, but for you to be able to find what you need!

# Exercise

Factor functions

# Today's summary

- We learned about the `tidyverse` and how to install and load packages
- We learned about the `tibble` and how to create new variables in a dataframe
- We learned about factor variables and how to manipulate them

# Today's functions

- `install.packages("package")`: install a package (once)
- `library(package)`: load a package (every time you want to use it)
- `c(value, value, value)`: concatenate values into a vector
- `mean(vector)`; `sd(vector)`: calculate the mean and standard deviation of a vector
- `glimpse(dataframe)`: get a quick overview of a dataframe
- `summary(dataframe)`; `summary(dataframe$variable)`: get a summary of a dataframe or single variable
- `mutate(dataframe, new_variable = function(old_variable))`: create a new variable
- `factor(variable, labels = , levels = )`: convert a variable to a factor
- `case_when(variable < value ~ "label", variable == value ~ "label")`: create a new variable based on a series of conditions
- `fct_relevel()`, `fct_recode()`, `fct_infreq()`, `fct_rev()`, `fct_na_value_to_level()`, `fct_collapse()`, `fct_lump()`, etc.: functions to manipulate factors (don't worry about memorizing, look up when you need to!)