# Programming day 2: Recoding and for loops

## Princeton Sociology Methods Camp

Angela Li[1]

Princeton University

August 25, 2023

[1] These slides are the collective effort of everyone who has contributed to the Princeton Sociology Methods Camp.

# Logistics

- Feedback from last session
- Homework going forward
- ChatGPT to start writing code
- Day 4 project

# Feedback from last session

*See Coding Feedback slides*

# Homework going forward

We will expect you to turn these in by the next day. You can use any time during the day to work with your buddy!

# ChatGPT for starting code

Another thing to try is to ask ChatGPT[2] to **explain how to get started on R code** if you get stuck on a task.

Try prompting it in the following way:

> *"I am a social scientist using the tidyverse to analyze a dataset called addh. It has the following columns: age, gender.... Could you show me how to change the income variable to a binary variable?"*

[2] https://chat.openai.com/

# Final project

We will send out information about this, but we'll be working with the Stanford Open Policing Project's data.

Your job is to clean, analyze, and make a small PDF report on the data. This will be our Day 4 project!

# Outline of today

- Recoding Variables: data types, logical statements, and case_when
- Review Assignment 1 as a class
- For Loops

Section 1

# Recoding variables: data types, logical statements, and case_when

# Recoding variables

We'll use the case of recoding variables as a way to learn about a few important programming concepts: **data types**, **logical statements**, and **control flow**.

A few examples of things that social scientists do when recoding variables:

- change a variable type from character to numeric (so two datasets will merge)
- add labels to numeric data (ie. 1 = male, 2 = female)
- create indicator, or binary, variables
- create categorical variables based on conditions

If you've used another statistical programming language to clean data, these tasks should be familiar to you! We'll use these commonly-used social science tasks to dig into some programming concepts.[3]

[3] For details about all of this, see Advanced R: https://adv-r.hadley.nz/index.html

# Recoding variables: changing data types

To change the type of a column in our data, use `mutate()` with `as.numeric()` or `as.character()` commands.

This becomes useful when two datasets do not merge despite a column that looks similar. It is often because one column is a character while the other is numeric.

```
class(addh$age)
```

```
## [1] "numeric"
```

```
addh2 <- addh %>%
  mutate(agechar = as.character(age))

class(addh2$agechar)
```

```
## [1] "character"
```

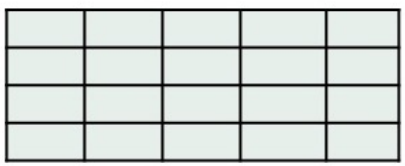
# Detour: vectors and data structures

Every data frame (or tibble in R) can be thought of as a bunch of **vectors** of different **types** stuck together as columns.

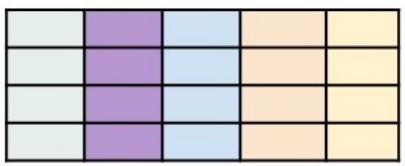
| Variables | Example |
|---|---|
| integer | 100 |
| numeric | 0.05 |
| character | “hello” |
| logical | TRUE |
| factor | “Green” |

Vector

Matrix



Data frame



We’ll introduce the basics about vectors that you need to know to manage data.

# vectors: examples and basic commands

- **What is a vector?**: 1-dimensional sequence of data elements of the same type
- Common examples of vectors:
  - Any individual column in a data.frame
  - Variable names: *colnames(data)*
  - Vector of numeric indices to subset data (e.g., vector of randomly sampled numbers)
- A data frame is a bunch of same-length vectors of differing types as columns!

# vectors: data types

- Different types of atomic vectors:
  1. numeric (encompasses integer and double)

# vectors: data types

- Different types of atomic vectors:
  1. numeric (encompasses integer and double)
  2. character: when creating, use quotes around each element

# vectors: data types

- Different types of atomic vectors:
  1. numeric (encompasses integer and double)
  2. character: when creating, use quotes around each element
  3. logical: TRUE/FALSE statements

# vectors: data types

- Different types of atomic vectors:
  1. numeric (encompasses integer and double)
  2. character: when creating, use quotes around each element
  3. logical: TRUE/FALSE statements
  4. Non-"atomic" type- factor: R treats differently than numeric– e.g., can't calculate averages or other things that do not make sense for categories. Has a "levels" attribute that codes levels of the factor and that you might label. More on this in a bit.

# vectors: creating a vector

- How to create a vector: use *c* to string together the elements
  - This stands for "**c**ombine"
  - `c()` is often useful in context of the tidyverse (ie renaming, joins)
  - See Appendix for useful shortcuts to create vectors (rep, seq, paste, and sample)

```
agevec <- c(18, 21, 23, 20)
agevec
```

```
## [1] 18 21 23 20
```

```
class(agevec)
```

```
## [1] "numeric"
```

```
gendervec <- c("male", "female", "other", "female")
gendervec
```

```
## [1] "male"   "female" "other"  "female"
```

```
class(gendervec)
```

```
## [1] "character"
```

# vectors: creating a vector

- Elements in a vector need to be of the same type, otherwise, **type coercion** happens

```
# What happened here?
c(18, "20", FALSE)
```

```
## [1] "18"    "20"    "FALSE"
```

```
# How about here?
c(1, 2, 3, TRUE, FALSE)
```

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

# vectors: data types

As we saw, convert from one type to another using the following functions:

- as.numeric()
- as.character()
- as.factor()

```
agevec
```

```
## [1] 18 21 23 20
```

```
as.character(agevec)
```

```
## [1] "18" "21" "23" "20"
```

```
gendervec
```

```
## [1] "male"   "female" "other"  "female"
```

```
as.numeric(gendervec) # why doesn't this work?
```

```
## Warning: NAs introduced by coercion
```

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

# vectors: data types

- Vectors can have a factor type: looks like a character vector but is actually a number under the hood ("labelled data")
  - Not as common in R anymore
  - Used to store data when computers weren't as powerful

```
genderfactorvec <- factor(gendervec,
                          levels = c("male", "female", "other"))
genderfactorvec
```

```
## [1] male   female other  female
## Levels: male female other
```

```
class(genderfactorvec)
```

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

```
as.numeric(genderfactorvec) # what happened here?
```

```
## [1] 1 2 3 2
```

# vectors: the building block of dataframes

- One way to **create** a dataframe: use `bind_cols()` to attach same-length vectors together as columns in a tibble
  - Vectors can be different types
  - Also can bind new columns to existing dataframes (but use mutate instead)
- Also a `bind_rows()` function

```
agevec
```

```
## [1] 18 21 23 20
```

```
gendervec
```

```
## [1] "male"   "female" "other"  "female"
```

```
bind_cols(age = agevec, gender = gendervec)
```

```
## # A tibble: 4 x 2
##     age gender
##   <dbl> <chr>
## 1    18 male
## 2    21 female
## 3    23 other
## 4    20 female
```

# Recoding variables: factor variables

Often, when we work with categorical survey data, we can use **factor variables**. The Add Health codebook for gender gives us the labels.[4]

**Example**: convert gender to a factor variable and save it in a new object addh2

```
addh2 <- addh %>% # Ctrl-Shift-M
  mutate(gender = factor(gender,
                         levels = c("male", "female"),
                         labels = c("male", "female")))

# Check new variable against old variable
str(addh$gender)
```

```
##  chr [1:3000] "female" "male" "female" "female" "female" "female" "male" ...
```

```
str(addh2$gender)
```

```
##  Factor w/ 2 levels "male","female": 2 1 2 2 2 2 1 2 1 2 ...
```

[4]The Add Health data included in the open-source version of these slides is fake for privacy purposes. However, you can download publicly released Add Health data from the following link: https://www.icpsr.umich.edu/web/ICPSR/studies/21600/datadocumentation

# Recoding variables: factor variables

**Exercise**: Before running the following code, what do you think each of these vectors look like? What does each line of code produce?

```
vec1 <- as.character(addh2$gender)
vec2 <- as.numeric(addh2$gender)

class(addh2$gender)
class(vec1)
class(vec2)
```

# Recoding variables: factor variables

**Exercise**: Before running the following code, what do you think each of these vectors look like? What does each line of code produce

```
vec1 <- as.character(addh2$gender)
vec2 <- as.numeric(addh2$gender)

head(vec1)
```

```
## [1] "female" "male"   "female" "female" "female" "female"
```

```
head(vec2)
```

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

# Recoding variables: factor variables

**Exercise**: Before running the following code, what do you think each of these vectors look like? What does each line of code produce

```
class(addh2$gender)
```

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

```
class(vec1)
```

```
## [1] "character"
```

```
class(vec2)
```

```
## [1] "numeric"
```

# Creating binary variables

To make a binary variable, you can use the `ifelse()` function.[5]

The syntax is:

> *ifelse(condition, value if logical condition is true, value if logical condition is false)*

The syntax for creating a new variable uses `mutate()`:

> *mutate(dataframename, newvariablename = ifelse(condition based on an existing variable, value of new variable if true, value of new variable if false))*

[5] This is slightly different from `if()` and `else()` on their own, which you can also use for "conditional execution" - most often in functions that you write: https://r4ds.had.co.nz/functions.html#conditional-execution. Also see Appendix for a more detailed breakdown.

# Detour, logical operators

**The main logical operators**:

- equals: ==

# Detour, logical operators

**The main logical operators**:

- equals: ==
- not equals: != (or ! in front of an identity statement)

# Detour, logical operators

**The main logical operators**:

- equals: $==$
- not equals: $!=$ (or ! in front of an identity statement)
- comparison: $<$, $\leq$, $>$, $\geq$

# Detour, logical operators

**The main logical operators**:

- equals: $==$
- not equals: $!=$ (or ! in front of an identity statement)
- comparison: $<$, $\leq$, $>$, $\geq$
- and: &

# Detour, logical operators

**The main logical operators**:

- equals: $==$
- not equals: $!=$ (or ! in front of an identity statement)
- comparison: $<$, $\leq$, $>$, $\geq$
- and: &
- or: |

# Creating binary variables

**Example**: create a new variable called `moneymoreimport` that takes a value of 1 if money is ranked higher than love, and 0 if it is ranked equally or ranked lower + move column to beginning

See Appendix for example with income.

```
addh2 <- mutate(addh,
                moneymoreimport = ifelse(money > love, 1, 0),
                .before = id) # put this new column *before* the id column - notice the period

head(addh2)
```

```
## # A tibble: 6 x 12
##   moneymoreimport    id   age gender income logincome debt     love nocheating
##             <dbl> <dbl> <dbl> <chr>   <dbl>     <dbl> <chr>   <dbl>      <dbl>
## 1               1     1    18 female 19252.      9.87 yesdebt     1          7
## 2               0     2    22 male   11617.      9.36 nodebt     10         10
## 3               0     3    18 female 16189.      9.69 yesdebt    10          3
## 4               1     4    26 female 18194.      9.81 yesdebt     2          1
## 5               1     5    27 female 24484.     10.1  yesdebt     5         10
## 6               0     6    21 female 22353.     10.0  nodebt     10          4
## # i 3 more variables: money <dbl>, paypercent <dbl>, logpaypercent <dbl>
```

# Creating categorical variables based on conditions

**Example**: want to create a new variable, *loveormoney*, that takes on one of three values:

- Equally important: respondent ranked the two as equally important
- Love more important: respondent ranked love as more important than money
- Money more important: respondent ranked money as more important than love

`ifelse()` only allows for two conditions. So we nest `ifelse()` statements.

But you can see how this might get complicated with many conditions. (What if we had 12 conditions and needed to have 11 nested ifelse statements?)

```
addh3 <- addh %>%
  mutate(loveormoney = ifelse(love == money,
                              "same",
                              ifelse(love > money,
                                     "lovegreater", "moneygreater")))
```

# Creating categorical variables based on conditions

Instead, use `case_when()` if there are 3 or more conditions for creating a variable.

The syntax for this is:

*casewhen(logical condition ~ value assigned, logical condition 2 ~ value assigned... .default = value if does not fit other logical conditions)*

```
addh3 <- addh %>%
  mutate(loveormoney = case_when(love == money ~ "same",
                                 love > money ~ "lovegreater",
                                 love < money ~ "moneygreater",
                                 .default = NA))
```

```
## Error in `mutate()`:
## ! Problem while computing `loveormoney = case_when(...)`.
## Caused by error in `case_when()`:
## ! Case 4 (`love == money ~ "same"`) must be a two-sided formula, not a
##   logical vector.
```

```
addh3 %>%
  select(id, love, money, loveormoney) %>%
  head(3)
```

```
## # A tibble: 3 x 4
##      id  love money loveormoney
##   <dbl> <dbl> <dbl> <chr>
## 1     1     1     9 moneygreater
```

# Creating categorical variables based on conditions

**Exercise**: create a new variable, *loveormoney2*, that takes on the following values:

- "extreme" if person either codes love or money as 9 or 10
- "lovegreater" if love $>$ money
- "same" if love $==$ money
- "moneygreater" if money $>$ love
- NA if none of the above

# Exercise:

In your groups, work on the homework Steps 1-3.

# Section 2

# Homework Review

# Logistics

We will walk through the homework as a class. We will take volunteers to demonstrate how they did various questions.

Please join the Zoom link. As a buddy pair, screenshare your solutions and explain to us what you did!

You can talk about: 1) how you approached the question 2) challenges you ran into/wrong turns 3) what you decided to do 4) differences between the posted solutions and your method of solving the question 4) anything else you want to share!

# Common issues with the assignment

- Not saving changes to new R object, just running code
- Names of new variables (overwrite vs. create new variable)
- Keeping track of new objects

Section 3

# For loops

# For loops

Two main ways to construct:

1. Go through every element of a vector:

```
for(i in vector){
  what to do
}
```

# For loops

Two main ways to construct:

1. Go through every element of a vector:

   for(i in vector){

   *what to do*

   }
2. Iterate through a set number of elements:

# For loops

Two main ways to construct:

1. Go through every element of a vector:

```
for(i in vector){
  what to do
}
```

2. Iterate through a set number of elements:
   2.1 Another way of writing the for loop from number 1:

```
for(i in 1:length(vector)){
  what to do
}
```

# For loops

Two main ways to construct:

1. Go through every element of a vector:

```
for(i in vector){
  what to do
}
```

2. Iterate through a set number of elements:
   2.1 Another way of writing the for loop from number 1:

```
for(i in 1:length(vector)){
  what to do
}
```

   2.2 A for loop that iterates starting at the second element:

```
for(i in 2:length(vector)){
  what to do
}
```

# For loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:

# For loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*

# For loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*
   1.2 Initialize an empty vector: *vec <- c()*

# For loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*
   1.2 Initialize an empty vector: *vec <- c()*
2. Use the for statement to tell the loop what to iterate through.

# For loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*
   1.2 Initialize an empty vector: *vec <- c()*
2. Use the for statement to tell the loop what to iterate through.
3. Copy and paste the code from the single-observation case into the "meat" part of the *for* loop sandwich

# For loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*
   1.2 Initialize an empty vector: *vec <- c()*
2. Use the for statement to tell the loop what to iterate through.
3. Copy and paste the code from the single-observation case into the "meat" part of the *for* loop sandwich
4. For step three, make sure to add indexing where appropriate

# For loops: example with sample means

We'll first create a new dataframe with just respondents with "high income".

```
addh3 <- filter(addh2, inclevel == "high")
```

```
## Error in `filter()`:
## ! Problem while computing `..1 = inclevel == "high"`.
## Caused by error in `mask$eval_all_filter()`:
## ! object 'inclevel' not found
```

```
nrow(addh3)
```

```
## [1] 3000
```

# For loops: example with sample means

Then, we'll sample once and take the mean of our sample.

```
set.seed(08540)
samp <- sample(addh3$money, size = 1000, replace = TRUE)
samp_mean <- mean(samp)
```

# For loops: example with sample means

Next, we'll stick our code into a (draft) for loop.

```
for (i in 1:1000) {
  samp <- sample(addh3$money, size = 1000, replace = TRUE)
  samp_mean <- mean(samp)
}
```

# For loops: example with sample means

Finally, we'll optimize our for loop and save the output.

```
sample_means <- numeric(length = 1000)
for (i in seq_along(sample_means)) {
  samp <- sample(addh3$money, size = 1000, replace = TRUE)
  samp_mean <- mean(samp)
  sample_means[i] <- samp_mean
}

mean(sample_means)
```

```
## [1] 5.569633
```

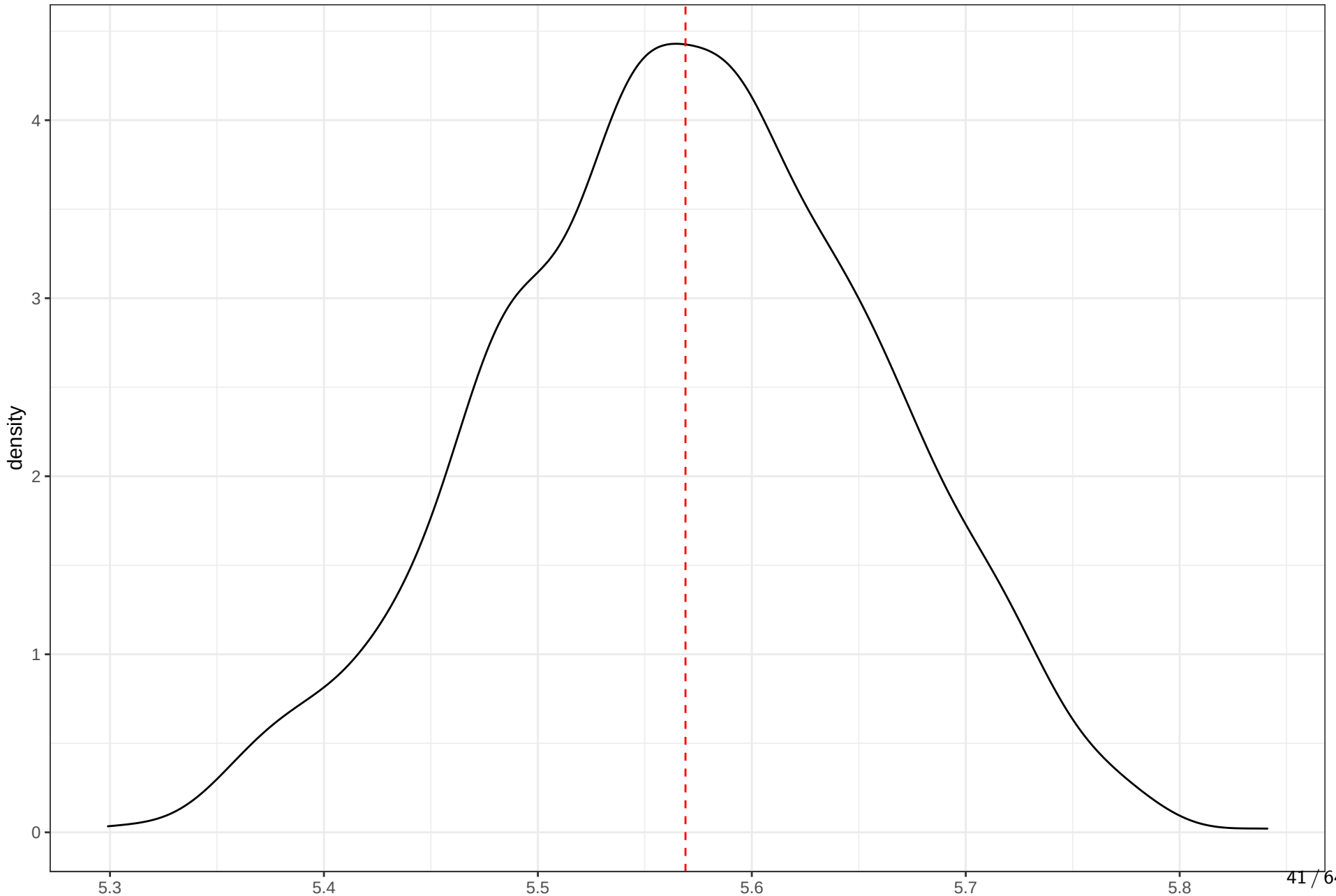
Changing small things about the for loop every time we want to change the sample size can be annoying, so tomorrow we'll learn about writing functions and purrr!

# For loops: example with sample means

We can visualize the distribution of these new means!

```
# plot it!
ggplot(as.data.frame(sample_means), aes(x = sample_means)) +
  geom_density() +
  geom_vline(xintercept = mean(addh3$money),
             col = "red", linetype = "dashed") +
  theme_bw()
```

# For loops: example with sample means



# Exercise:

In your groups, work on the homework Step 4.

# Summing up

In this lecture, we've reviewed:

- Recoding variables as a case study for programming concepts in R
  - Logical statements
  - Control structures
- For loops

# For next day of camp

- Topics:
  - Functions
  - Iteration beyond for loops (with purrr)
- Fill out feedback form
- Complete homework with your buddy

Section 4

# Appendix

# vectors: shortcuts to create a vector

- Depending on what you want in the vector, there are functions to help you create the vector more efficiently:

1. *rep*: repeat the same thing multiple times
2. *seq*: create a sequence of numbers
3. *paste*: stick together character and numeric info
4. *sample*: for vectors where we want to randomly sample from some larger pool

# vectors: shortcuts to create a vector

1. *rep*: repeat the same thing multiple times

```
rep(1000, 5)
```

```
## [1] 1000 1000 1000 1000 1000
```

```
rep("Wave 3", 5)
```

```
## [1] "Wave 3" "Wave 3" "Wave 3" "Wave 3" "Wave 3"
```

# vectors: shortcuts to create a vector

2. *seq* for sequence patterns in numeric vectors

```
##create a sequence of each decade (1900, 1910, 1920...)
decades <- seq(from = 1900, to = 2000, by = 10)
decades
```

```
##  [1] 1900 1910 1920 1930 1940 1950 1960 1970 1980 1990 2000
```

# vectors: shortcuts to create a vector

3. *paste*: for patterns in character vectors

```
##create names for decades spanning from 1900 to 2000 by 10
decadenames <- paste(c("decade", "birthday"),
                     seq(from = 1900, to = 2000, by = 10),
                     sep = "_")
decadenames
```

```
##  [1] "decade_1900"   "birthday_1910" "decade_1920"   "birthday_19
##  [5] "decade_1940"   "birthday_1950" "decade_1960"   "birthday_19
##  [9] "decade_1980"   "birthday_1990" "decade_2000"
```

# vectors: shortcuts to create a vector

4. *sample*: for vectors where we want to randomly sample from some larger pool

```
##set a seed so we sample same ids each time
set.seed(123)

##create a vector with three randomly sampled id's
sampids <- sample(addh$id, size = 3)
sampids
```

```
## [1] 2463 2511 2227
```

# Breaking down the sample command

sample(vector to sample from, size of sample, replace or not? (default = no replacement))

In our use, we sampled 3 random IDs from a vector of all respondent IDs, and we defaulted to sampling without replacement:

```
sample(addh$id, size = 3)
```

# Schematic to understand data structures in R

| | **Homogeneous elements** | **Heterogeneous elements** |
|---|---|---|
| 1-dimensional | Vector | List |
| 2-dimensional | Matrix | **Data.frame** |

*Source: Hadley Wickham's Advanced R*

# Creating binary variables (income example)

**Example**: create a new, recoded variable called `highinc` for respondents with a (relatively) "high income", defined as the 75th percentile or above

```
summary(addh$income)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1008    9372   15127   15231   20518   41700
```

```
# get the 75th percentile of this group to define "high income"
inc75 <- quantile(addh$income)[4]
quantile(addh$income)[4]
```

```
##      75%
## 20518.21
```

```
inc75
```

```
##      75%
## 20518.21
```

```
# make new binary variable for if respondent has high income
addh2 <- addh %>%
  mutate(highinc = ifelse(income > inc75, 1, 0))
```

# Creating categorical variables based on conditions (income example)

**Example**: create an income bucket variable called `income_buckets` for "low" (0-25%), "medium" (25-75%), and "high" (75%+) incomes.

`ifelse()` only allows for two conditions. So we nest `ifelse()` statements. But you can see how this might get complicated.

```
inc25 <- quantile(addh$income)[2]

addh2 <- addh %>%
  mutate(inclevel = ifelse(income <= inc25,
                           "low",
                           ifelse(income >= inc75,
                                  "high", "medium")))
```

# Creating categorical variables based on conditions (income example)

Instead, use `case_when()` if there are 3 or more conditions for creating a variable.

The syntax for this is:

*casewhen(logical condition ~ value assigned, logical condition 2 ~ value assigned... .default = value if does not fit other logical conditions)*

```
addh2 <- addh %>%
  mutate(inclevel = case_when(income <= inc25 ~ "low",
                              income >= inc75 ~ "high",
                              .default = "medium"))
```

```
## Error in `mutate()`:
## ! Problem while computing `inclevel = case_when(...)`.
## Caused by error in `case_when()`:
## ! Case 3 (`income <= inc25 ~ "low"`) must be a two-sided formula, not a
##   character vector.
```

# If / else: perform different operations based on specific conditions

**If Else: common control structures**:

ifelse(logical test, what to do if true, what to do if false)

if(logical test, what to do if true), else(what to do if false)

if(logical test, what to do if true), else(logical test, what to do if true), else(...)...

# If / else: perform different operations based on specific conditions

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?

# If / else: perform different operations based on specific conditions

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

# If / else: perform different operations based on specific conditions

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

```
if(logical statement){
   what to do
}
```

# If / else: perform different operations based on specific conditions

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

*what to do*

} else if (logical statement){

*what to do*

}

# If / else: perform different operations based on specific conditions

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

*what to do*

} else if (logical statement){

*what to do*

} else if (logical statement){

*what to do*

}

# If / else: perform different operations based on specific conditions

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

*what to do*

} else if (logical statement){

*what to do*

} else if (logical statement){

*what to do*

} else {

*what to do with values that didn't meet any of the above conditions*

}

# matrices: using for loops to populate a matrix

**Example**: want to calculate distribution around mean rating of money's importance. To do that, we want to:

1. Draw 1000 samples of size 3050 (number of observations in data) with replacement and store the samples in a matrix where each row is a draw and each column is an element of that draw (so the matrix will be 1000 x 3050) inside the loop

# matrices: using for loops to populate a matrix

**Example**: want to calculate distribution around mean rating of money's importance. To do that, we want to:

1. Draw 1000 samples of size 3050 (number of observations in data) with replacement and store the samples in a matrix where each row is a draw and each column is an element of that draw (so the matrix will be 1000 x 3050) inside the loop
2. Find mean of each of the 1000 samples outside the loop

# matrices: using for loops to populate a matrix

**Example**: want to calculate distribution around mean rating of money's importance. To do that, we want to:

1. Draw 1000 samples of size 3050 (number of observations in data) with replacement and store the samples in a matrix where each row is a draw and each column is an element of that draw (so the matrix will be 1000 x 3050) inside the loop
2. Find mean of each of the 1000 samples outside the loop
3. Plot the distribution of that mean outside the loop

# matrices: using for loops to populate a matrix

```
# initialize empty matrix, good to preallocate space
set.seed(1234)
sampmat <- matrix(NA, nrow = 1000, ncol = 3050)
# iterate through each row of the matrix
for(i in 2:nrow(sampmat)){
  # and fill it with a sample of size 10 from the data
  draws <- sample(addh$money, size = 3050, replace= TRUE)
  # note that because each i-th sample is filling a row,
  # we add that sample to the matrix by indexing the i-th row
  sampmat[i, ] <- draws
}

# this is basically bootstrapping!
# check to make sure the for loop properly populated the matrix
sampmat[1:2, 1:10]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
## [2,]    9    2    9    3    3    2    5    4    6     6
```

```
# find mean of each 1000 samples
samplemeans <- rowMeans(sampmat)
```

# matrices: using for loops to populate a matrix

We can now visualize using ggplot2!

```
# plot distribution of mean ratings
# adding a vertical line for observed mean
ggplot(as.data.frame(samplemeans), aes(x = samplemeans)) +
  geom_density() +
  geom_vline(xintercept = mean(addh$money),
             col = "red", linetype = "dashed") +
  theme_bw()
```

```
## Warning: Removed 1 rows containing non-finite values (stat_density).
```