Coding style + Data types + Recoding

Dr. Maria Tackett

09.19.19

Click for PDF of slides

Announcements

- Writing Exercise #1 final revision **due today at 11:59p**
- HW 02 - due **Thursday, September 26 at 11:59p**

Coding style

Style guide

> "Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread."
>
> Hadley Wickham

- Style guide for this course is based on the Tidyverse style guide: http://style.tidyverse.org/
- There's more to it than what we'll cover today. We'll mention more as we introduce more functionality throughout the semester.

File names and code chunk labels

- Do not use spaces in file names, use - or _ to separate words
- Use all lowercase letters

```
# Good
ucb-admit.csv

# Bad
UCB Admit.csv
```

Object names

- Use _ to separate words in object names
- Use informative but short object names
- Do not reuse object names within an analysis

```
# Good
acs_employed

# Bad
acs.employed
acs2
acs_subset
acs_subsetted_for_males
```

Spacing

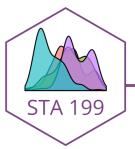
- Put a space before and after all infix operators (=, +, -, <-, etc.), and when naming arguments in function calls.
- Always put a space after a comma, and never before (just like in regular English).

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

ggplot

- Always end a line with **+**
- Always indent the next line

```
# Good
ggplot(diamonds, mapping = aes(x = price)) +
  geom_histogram()

# Bad
ggplot(diamonds,mapping=aes(x=price))+geom_histogram()
```



Long lines

- Try to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font.
- Take advantage of RStudio editor's auto formatting for indentation at line breaks.

Assignment

- Use **<-** not **=**

```
# Good
x <- 2

# Bad
x = 2
```

Quotes

Use **"**, not **'**, for quoting text. The only exception is when the text already contains double quotes and no single quotes.

```
ggplot(diamonds, mapping = aes(x = price)) +
  geom_histogram() +
  # Good
  labs(title = "`Shine bright like a diamond`",
  # Good
       x = "Diamond prices",
  # Bad
       y = 'Frequency')
```

Data types

Data types in R

- logical
- double
- integer
- character
- lists
- and some more, but we won't be focusing on those

Logical & character

logical - Boolean values **TRUE** and **FALSE**

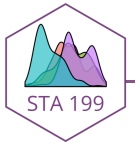
```
typeof(TRUE)
```

```
## [1] "logical"
```

character - character strings

```
typeof("hello")
```

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



Double & integer

double - floating point numerical values (default numerical type)

```
typeof(1.335)
```

```
## [1] "double"
```

```
typeof(7)
```

```
## [1] "double"
```

integer - integer numerical values (indicated with an **L**)

```
typeof(7L)
```

```
## [1] "integer"
```

```
typeof(1:3)
```

```
## [1] "integer"
```

Lists

- **Lists** are 1d objects that can contain any combination of R objects

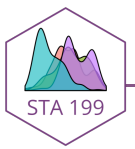
```
mylist <- list("A", 1:4, c(TRUE, FALSE), (1:4)/2)
mylist
```

```
## [[1]]
## [1] "A"
##
## [[2]]
## [1] 1 2 3 4
##
## [[3]]
## [1]  TRUE FALSE
##
## [[4]]
## [1] 0.5 1.0 1.5 2.0
```

Lists

```
str(mylist)
```

```
## List of 4
##  $ : chr "A"
##  $ : int [1:4] 1 2 3 4
##  $ : logi [1:2] TRUE FALSE
##  $ : num [1:4] 0.5 1 1.5 2
```



Named lists

Because of their more complex structure we often want to name the elements of a list (we can also do this with vectors). This can make reading and accessing the list more straight forward.

```
myotherlist <- list(A = "hello", B = 1:4, "knock knock" = "who's there?")
str(myotherlist)
```

```
## List of 3
##  $ A           : chr "hello"
##  $ B           : int [1:4] 1 2 3 4
##  $ knock knock: chr "who's there?"
```

```
names(myotherlist)
```

```
## [1] "A"           "B"           "knock knock"
```

```
myotherlist$B
```

```
## [1] 1 2 3 4
```

Concatenation

Vectors can be constructed using the **`c()`** function.

```
c(1, 2, 3)
```

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

```
c("Hello", "World!")
```

```
## [1] "Hello"  "World!"
```

```
c(1, c(2, c(3)))
```

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

Coercion

R is a dynamically typed language -- it will easily convert between various types

```
c(1, "Hello")
```

```
## [1] "1"     "Hello"
```

```
c(FALSE, 3L)
```

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

```
c(1.2, 3L)
```

```
## [1] 1.2 3.0
```

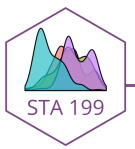
In general, R will convert all values to the simplest type needed to represent all the information

Missing Values

R uses **NA** to represent missing values in its data structures.

```
typeof(NA)
```

```
## [1] "logical"
```



Other Special Values

`NaN` - Not a number

`Inf` - Positive infinity

`-Inf` - Negative infinity

```
pi / 0
```

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

```
0 / 0
```

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

```
1/0 + 1/0
```

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

```
1/0 - 1/0
```

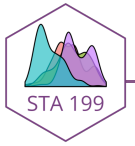
```
## [1] NaN
```

```
NaN / NA
```

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

```
NaN * NA
```

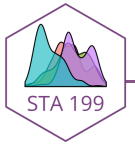
```
## [1] NaN
```



Activity

What is the type of the following vectors? Explain why they have that type.

1. `c(1, NA+1L, "C")`
2. `c(1L / 0, NA)`
3. `c(1:3, 5)`
4. `c(3L, NaN+1L)`
5. `c(NA, TRUE)`



Example: Cat lovers

A survey asked respondents their name and number of cats. The instructions said to enter the number of cats as a numerical value. See the full table here.

```
cat_lovers <- read_csv("data/cat-lovers.csv")
```

Show 10 entries Search:

	name	number_of_cats	handedness
1	Bernice Warren	0	left
2	Woodrow Stone	0	left
3	Willie Bass	1	left
4	Tyrone Estrada	3	left
5	Alex Daniels	3	left
6	Jane Bates	2	left

Why isn't this working?!

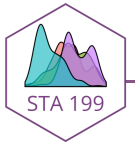
```
cat_lovers %>%
  summarise(mean = mean(number_of_cats))
```

```
## # A tibble: 1 x 1
##    mean
##   <dbl>
## 1    NA
```

Why is this still not working??!!

```
cat_lovers %>%
  summarise(mean_cats = mean(number_of_cats, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   mean_cats
##       <dbl>
## 1        NA
```



Let's look at the data...

What is the type of the **`number_of_cats`** variable?

```
glimpse(cat_lovers)
```

```
## Observations: 60
## Variables: 3
## $ name           <chr> "Bernice Warren", "Woodrow Stone", "Willie Bass",…
## $ number_of_cats <chr> "0", "0", "1", "3", "3", "2", "1", "1", "0", "0",…
## $ handedness     <chr> "left", "left", "left", "left", "left", "left", "…
```

Let's take another look

Cat Lovers Data

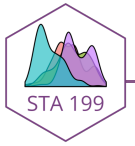
Show 10 entries Search:

	name	number_of_cats	handedness
1	Bernice Warren	0	left
2	Woodrow Stone	0	left
3	Willie Bass	1	left
4	Tyrone Estrada	3	left
5	Alex Daniels	3	left
6	Jane Bates	2	left
7	Latoya Simpson	1	left
8	Darin Woods	1	left

Sometimes you have to fix data entry errors

```
cat_lovers %>%
  mutate(number_of_cats = case_when(
    name == "Ginger Clark" ~ 2,
    name == "Doug Bass"    ~ 3,
    TRUE                   ~ as.numeric(number_of_cats)
    )) %>%
  summarise(mean_cats = mean(number_of_cats))
```

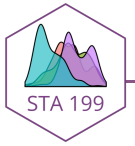
```
## # A tibble: 1 x 1
##   mean_cats
##       <dbl>
## 1     0.817
```



You always need to respect data types

```
cat_lovers %>%
  mutate(
    number_of_cats = case_when(
      name == "Ginger Clark" ~ "2",
      name == "Doug Bass"    ~ "3",
      TRUE                   ~ number_of_cats
      ),
    number_of_cats = as.numeric(number_of_cats)
    ) %>%
  summarise(mean_cats = mean(number_of_cats))
```

```
## # A tibble: 1 x 1
##   mean_cats
##       <dbl>
## 1     0.817
```



Now that we know what we're doing...

```
cat_lovers <- cat_lovers %>%
  mutate(
    number_of_cats = case_when(
      name == "Ginger Clark" ~ "2",
      name == "Doug Bass"    ~ "3",
      TRUE                   ~ number_of_cats
      ),
    number_of_cats = as.numeric(number_of_cats)
    )
```

Moral of the story

- If your data does not behave how you expect it to, type coercion upon reading in the data might be the reason.
- Go in and investigate your data, apply the fix, **save your data**, live happily ever after.

Vectors vs. lists

Vectors

```
x <- c(8,4,7)
```

```
x[1]
```

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

```
x[[1]]
```

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

Lists

```
y <- list(8,4,7)
```

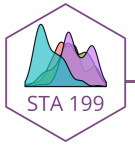
```
y[2]
```

```
## [[1]]
## [1] 4
```

```
y[[2]]
```

```
## [1] 4
```

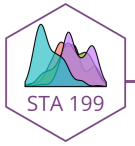
Note: When using tidyverse code you'll rarely need to refer to elements using square brackets, but it's good to be aware of this syntax, especially since you might encounter it when searching for help online.



Data "set"

Data "sets" in R

- "set" is in quotation marks because it is not a formal data class
- A tidy data "set" can be one of the following types:
  - `tibble`
  - `data.frame`



Data frames & tibbles

- A **`data.frame`** is the most commonly used data structure in R, they are just a list of equal length vectors. Each vector is treated as a column and elements of the vectors as rows.
- A tibble is a type of data frame that ... makes the data analysis easier.
- Most often a data frame will be constructed by reading in from a file, but we can also create them from scratch.
  - **`readr`** package (e.g. **`read_csv`** function) loads data as a **`tibble`** by default
  - **`tibble`**s are part of the tidyverse, so they work well with other packages we are using
  - they make minimal assumptions about your data, so are less likely to cause hard to track bugs in your code

Creating data frames

```
df <- tibble(x = 1:3, y = c("a", "b", "c"))
class(df)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
glimpse(df)
```

```
## Observations: 3
## Variables: 2
## $ x <int> 1, 2, 3
## $ y <chr> "a", "b", "c"
```

Features of data frames

```
attributes(df)
```

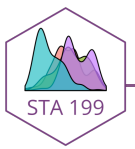
```
## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
class(df$x)
```

```
## [1] "integer"
```

```
class(df$y)
```

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



Working with tibbles in pipelines

How many respondents have below average number of cats?

```
mean_cats <- cat_lovers %>%
  summarise(mean_cats = mean(number_of_cats))

cat_lovers %>%
  filter(number_of_cats < mean_cats) %>%
  nrow()
```

```
## [1] 60
```

Do you believe this number? Why, why not?

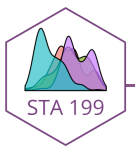
A result of a pipeline is always a tibble

```
mean_cats
```

```
## # A tibble: 1 x 1
##   mean_cats
##       <dbl>
## 1     0.817
```

```
class(mean_cats)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```



`pull()` can be great

But use it sparingly!

```
mean_cats <- cat_lovers %>%
  summarise(mean_cats = mean(number_of_cats)) %>%
  pull()

cat_lovers %>%
  filter(number_of_cats < mean_cats) %>%
  nrow()
```

```
## [1] 33
```

```
mean_cats
```

```
## [1] 0.8166667
```

```
class(mean_cats)
```

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

Factors

Factors

Factor objects are what R uses to store data for categorical variables (fixed numbers of discrete values).

```
(x = factor(c("BS", "MS", "PhD", "MS")))
```

```
## [1] BS  MS  PhD MS
## Levels: BS MS PhD
```

```
glimpse(x)
```

```
##  Factor w/ 3 levels "BS","MS","PhD": 1 2 3 2
```

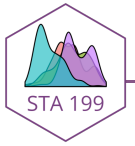
```
typeof(x)
```

```
## [1] "integer"
```

Read data in as character strings

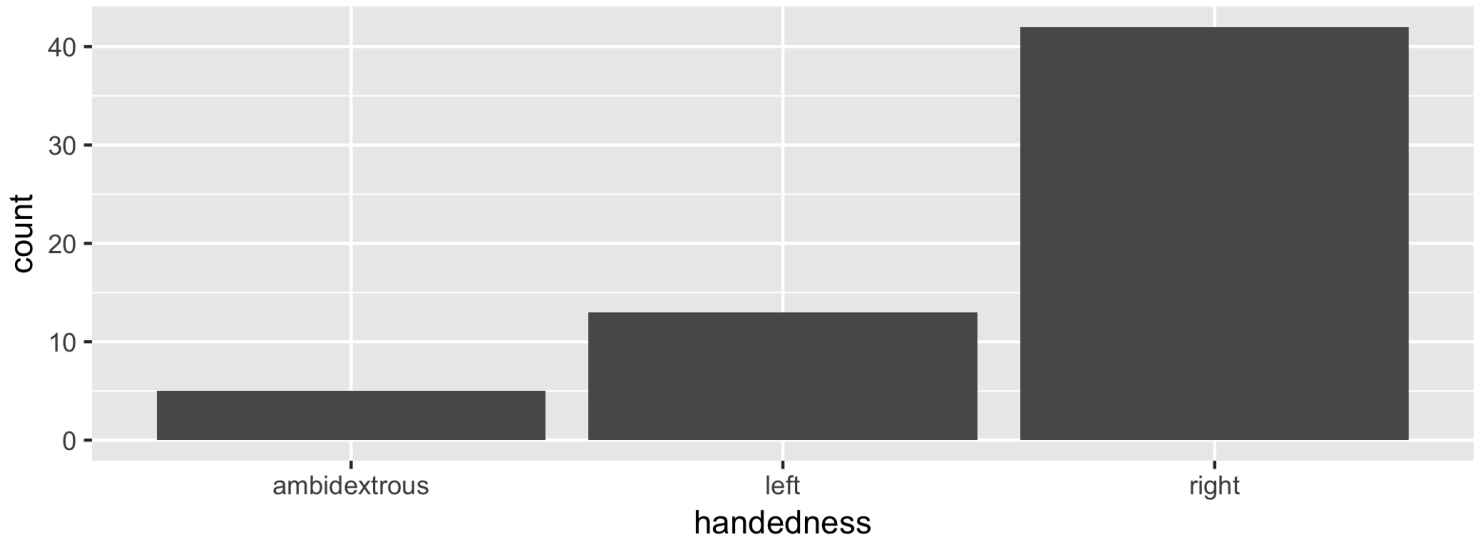
```
glimpse(cat_lovers)
```

```
## Observations: 60
## Variables: 3
## $ name           <chr> "Bernice Warren", "Woodrow Stone", "Willie Bass",…
## $ number_of_cats <dbl> 0, 0, 1, 3, 3, 2, 1, 1, 0, 0, 0, 0, 1, 3, 3, 2, 1…
## $ handedness     <chr> "left", "left", "left", "left", "left", "left", "…
```



But coerce when plotting

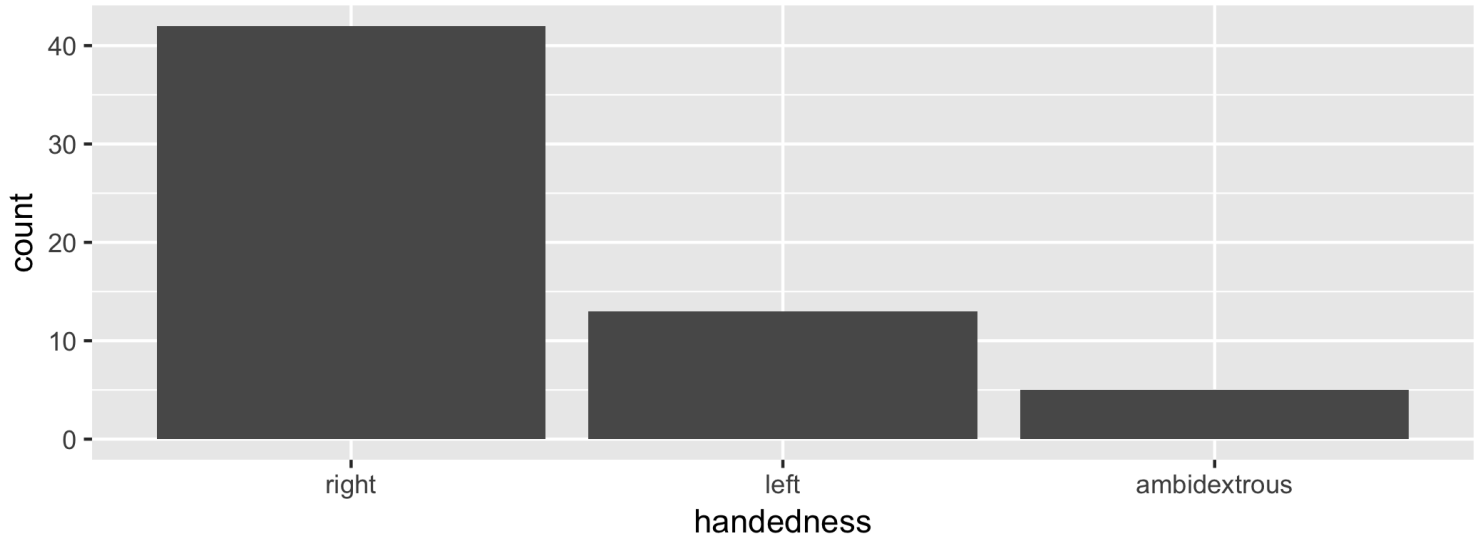
```
ggplot(cat_lovers, mapping = aes(x = handedness)) +
  geom_bar()
```



Use forcats to manipulate factors

```
cat_lovers <- cat_lovers %>%
  mutate(handedness = fct_relevel(handedness,
                                  "right", "left", "ambidextrous"))
```

```
ggplot(cat_lovers, mapping = aes(x = handedness)) +
  geom_bar()
```



forcats

- R uses factors to handle categorical variables, variables that have a fixed and known set of possible values. Historically, factors were much easier to work with than character vectors, so many base R functions automatically convert character vectors to factors.

- However, factors are still useful when you have true categorical data, and when you want to override the ordering of character vectors to improve display. The goal of the forcats package is to provide a suite of useful tools that solve common problems with factors.

Source: forcats.tidyverse.org

Recap

- Always best to think of data as part of a tibble
  - This works nicely with the **`tidyverse`** as well
  - Rows are observations, columns are variables
- Be careful about data types / classes
  - Sometimes **`R`** makes silly assumptions about your data class
    - Using **`tibble`**s help, but it might not solve all issues
    - Think about your data in context, e.g. 0/1 variable is most likely a **`factor`**
  - If a plot/output is not behaving the way you expect, first investigate the data class
  - If you are absolutely sure of a data class, over-write it in your tibble so that you don't need to keep having to keep track of it
    - **`mutate`** the variable with the correct class