Announcements

1. Will give feedback on everyone’s workflow critiques before next class
2. Final workflow project change
   a. Week 10: ~~Presentation~~ -> “in-class hackathon”
   b. Come prepared with a few changes you’re stuck on, advice you want to ask
   c. Due date for project -> 3/17 (Week 11)
   d. Project guidelines will be posted shortly

Today

1. Workflow self-critique presentations
2. Break
3. Refactoring and technical debt
4. Advanced functions tutorial

PSYC 259: Principles of Data Science

Week 6: Technical debt

Technical debt, design smells, and refactoring

(Suryanarayana et al. 2015)

“Technical debt”

- Debt that we incur by writing code
  - “Interest” accrues the longer we fail to pay off the debt
- “…debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions”
  - For a software company, that debt can be time/money
  - For a lab, that debt is more likely time/fidelity

“Design smells”

- “…certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality”
- Hard coding, duplicated code, poor documentation, inflexibility, etc.

Example

- A grad student writes code to clean/analyze Exp 1
  - The details of the study are “hard-coded”
  - Cleaning/analysis in one long script
  - Variable names are hard to understand and the script is poorly documented
  - Used packages/functions that are now deprecated
- The code works…but now there’s an Exp 2 that adds IVs and additional measurements
  - The technical debt of those original design decisions must be repaid by re-coding Exp 2

Software always needs to be maintained

- Is the structure of the original code making it hard to write features?
- What are the effects of small changes/ extensions on other parts of the code base?
- Are inputs changing in quality/format?
- How does old code run on new software/ hardware?

How to pay off technical debt

- Refactoring
  - Rewriting code to change its design, style, or structure without changing its function
  - The opposite of “if it ain’t broke, don’t fix it”!
  - Just like revising is a part of writing, refactoring is a part of coding
    - We rephrase/refactor as we write code
    - We make larger changes to structure when needed

Design qualities to strive towards

- Understandability = able to reread your code in a week/month/year/decade, share it with others
- Reusability = general enough to be used again
- Changeability = can modify code easily without changing its function
- Reliability = code is resistant to breaking in the future

Examples using the built-in diamonds dataset

```
> head(diamonds)
# A tibble: 6 x 10
  carat cut       color clarity depth table price
  <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int>
1 0.23  Ideal     E     SI2      61.5    55   326
2 0.21  Premium   E     SI1      59.8    61   326
3 0.23  Good      E     VS1      56.9    65   327
4 0.290 Premium   I     VS2      62.4    58   334
5 0.31  Good      J     SI2      63.3    58   335
6 0.24  Very Good J     VVS2     62.8    57   336
```

- Goal: Filter data by cut and calculate the average price

Understandability & Reusability

```r
#BAD
library("tidyverse")
x <- diamonds
cs <- unique(diamonds[,2]) %>% pull
csmp <- map_dbl(cs,
                ~ summarize(filter(x, cut == .x), x = mean(price)) %>%
                pluck(1)) %>%
  set_names(cs)
```

```
> csmp
    Ideal   Premium      Good Very Good      Fair 
 3457.542  4584.258  3928.864  3981.760  4358.758 
```

Understandability & Reusability

```
#BAD
library("tidyverse")
x <- diamonds
cs <- unique(diamonds[,2]) %>% pull
csmp <- map_dbl(cs,
                ~ summarize(filter(x, cut == .x), x = mean(price)) %>%
                  pluck(1)) %>%
  set_names(cs)
```

```
#BETTER
library("tidyverse")
df <- diamonds

price_summary <- df %>% group_by(cut) %>% summarize(mean_price = mean(price))
cut_prices <- price_summary %>% pull(mean_price) %>% set_names(price_summary$cut)
```

Understandability & Reusability

```r
#BETTER
library("tidyverse")
df <- diamonds

price_by_cut <- function(data, cut_name) {
  data %>%
    filter(cut == cut_name) %>%
    summarize(mean_price = mean(price)) %>%
    as.double
}
cut_levels <- fct_unique(df$cut)
map_dbl(cut_levels, ~ price_by_cut(df, .x)) %>% set_names(cut_levels)
```

Non-standard evaluation can help with reusability

```r
library("tidyverse")
df <- diamonds

dv_by_iv <- function(data, iv_var, iv_level, dv) {
  data %>%
    filter({{iv_var}} == iv_level) %>%
    summarize(mean_dv = mean({{dv}})) %>%
    as.double
}

cut_levels <- fct_unique(df$cut)
map(cut_levels, ~ dv_by_iv(df, cut, .x, price)) %>% set_names(cut_levels)
map(cut_levels, ~ dv_by_iv(df, cut, .x, depth)) %>% set_names(cut_levels)
```

https://www.tidyverse.org/blog/2020/02/glue-strings-and-tidy-eval/

Default arguments can help make functions more reusable

```
price_by_cut <- function(data, cut_name, na.rm = T) {
  data %>%
    filter(cut == cut_name) %>%
    summarize(mean_price = mean(price, na.rm = na.rm)) %>%
    as.double
}
```

if() statements can also help with reusability

- In functions, you can use them to handle different options

```
if (condition == T) {
  #Do the stuff here
} else {
  #Do the stuff here
}
```

if() statements can also help with reusability

- In functions, you can use them to handle different options

```
price_by_cut <- function(data, cut_name = "Overall", na.rm = T) {
  if (cut_name != "Overall") {
    data <- filter(data, cut == cut_name)
  }
  data %>%
    summarize(mean_price = mean(price, na.rm = na.rm)) %>%
    as.double
}
```

Changeability & reliability

- Avoiding hard-coding filenames and working directories
- Avoiding repetitive code (easier to change if it needs to be changed in fewer places)
- Refactoring to avoid deprecated functions (or track package dates to recreate environment)

What leads to technical debt?

- Lack of awareness
- Schedule pressure
    - Priority is to get results, not to write good code
- When should you refactor?
    - After rushing to meet a deadline
    - Before starting an experiment 2 or adding a large analysis component
    - Preserve pre-refactored versions if published