Refactoring Based on Clean Code

Slide 0: Why Functions?

- Make your code easier to understand
- As things change, only need to update the code in one place
- Reduce the chance of making a mistake
- Have less overall code to edit
- Let other people use the functionality without having to understand everything

Slide 0: Why Functions?

- Make your code easier to understand
- As things change, only need to update the code in one place
- Reduce the chance of making a mistake
- Have less overall code to edit
- Let other people use the functionality without having to understand everything *Makes Your Code Cleaner*

When to Function?

- Whenever you repeat yourself more than twice (or maybe even once); DRY
- When you have several complex steps going and you want to make your code easier to understand from a holistic point of view -When your code won't functionally be changing, but maybe it's inputs will (you starting out by reading a csv, but you want to read from a database eventually)

DRY Example

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$b, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

DRY Example

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

DRYer Example

```
df %<>% mutate(across(c(a, b, c, d), rescale01))
```

Meaningful Names

- Names should reveal intent.
- Use descriptive, unambiguous names for variables, functions, classes, etc.
- Clear naming reduces the need for comments.

Meaningful Names

- Your code will run the same whether you use functions or not, so maximize their value for human consumption of your code
- Make your functions names:
  - Short but clear what they do $\rightarrow$ longer is better if it makes it more clear
  - Function names should be verbs and arguments should be nouns (generally)
  - Use snake_case (I guess you could use camelCase, but don't)

What's Wrong with this Function?

```
calc <- function(d, x) {
  d * x
}
```

Here’s a Better Version

```
calculate_total_price <- function(quantity, price_per_unit) {
  quantity * price_per_unit
}
```

Here's a Better Version

```
calculate_total_price <- function(quantity, price_per_unit) {
  quantity * price_per_unit
}
```

“You should name a variable using the same care with which you name a first-born child.” — Robert C. Martin

Single Responsibilty Principle

- Functions should do one thing (SRP).
- Small functions that have limited scope are easier to test, debug, and understand.

Too Complex

```
process_data <- function(file_path, column_name, threshold, new_col_name) {

  data <- read_csv(file_path)

  data <- data %>%
    drop_na()

  data <- data %>%
    filter(!!sym(column_name) > threshold)

  data <- data %>%
    mutate(!!sym(new_col_name) := ifelse(
    !!sym(column_name) > mean(!!sym(column_name)),
    "Above Average",
    "Below Average")
    )

  summary_data <- data %>%
    summarize(
      Mean = mean(!!sym(column_name), na.rm = TRUE),
      Median = median(!!sym(column_name), na.rm = TRUE),
      Count = n()
    )

  write_csv(data, paste0("cleaned_", basename(file_path)))
  write_csv(summary_data, paste0("summary_", basename(file_path)))
```

Better

```
process_data <- function(file_path, column_name, threshold, new_col_name) {
  data <- read_and_clean_data(file_path)
  filtered_data <- filter_data(data, column_name, threshold)
  enriched_data <- add_new_column(filtered_data, column_name, new_col_name)
  summary_data <- summarize_data(enriched_data, column_name)
  write_data_to_csv(enriched_data, summary_data, file_path)
  list(cleaned_data = enriched_data, summary_data = summary_data)
}
```

Better

```
process_data <- function(file_path, column_name, threshold, new_col_name) {

  data <- read_and_clean_data(file_path)

  filtered_data <- filter_data(data, column_name, threshold)

  enriched_data <- add_new_column(filtered_data, column_name, new_col_name)

  summary_data <- summarize_data(enriched_data, column_name)

  write_data_to_csv(enriched_data, summary_data, file_path)

  list(cleaned_data = enriched_data, summary_data = summary_data)
}
```

- This is a **general contractor** function

General Contractor

```r
build_me_a_house <- function() {
  electric_system <- build_electricity()
  hvac_system <- build_hvac()
  plumbing_system <- build_plumbing()
  framing <- build_framing()

  house <- electric_system + hvac_system + plumbing_system + framing
  house
}
```

Avoid Side Effects

- Functions should not have unexpected side effects.
- Side effects make code unpredictable and difficult to debug.

What's Wrong with This?

```r
add_flag_column <- function(data, column_name, threshold) {
  data[[paste0(column_name, "_flag")]] <- ifelse(
                                            data[[column_name]]
                                            "high",
                                            "low")
}
```

What's Wrong with This?

```
add_flag_column <- function(data, column_name, threshold) {
  data[[paste0(column_name, "_flag")]] <- ifelse(
                                              data[[column_name]]
                                              "high",
                                              "low")
}
```

The function modifies the input data frame *directly*. If the original data frame is used elsewhere in the code, it now has the extra column and possibly unwanted changes. This can cause *unexpected results* in other parts of the code that use the same data frame.

How to Avoid Side Effects

```
add_flag_column <- function(data, column_name, threshold) {
  data_copy <- data
  data_copy[[paste0(column_name, "_flag")]] <- ifelse(data_copy[[column_name]]
  data_copy
}
```

Now, when you call this function, it will return a new data frame without modifying the original one.

Comments Should Be Rare and Meaningful

- Comments should explain *why*, not *what* the code is doing.
- Clean code is self-explanatory; comments should be used sparingly.

Comments Should Be Rare and Meaningful

- Comments should explain *why*, not *what* the code is doing.
- Clean code is self-explanatory; comments should be used sparingly. **Let's go back to the `process_data` function**

Lots of Comments is a Sign Something is Wrong

- *Excessive Comments*: When you see a lot of comments in a codebase, it's often a sign that the code is too complex or unclear. This means the code itself isn't doing a good job of communicating its purpose, and the comments are trying to compensate for poor readability or structure.
- *Outdated Comments*: Comments can easily become outdated or inaccurate as the code evolves, especially if developers forget to update them. This can lead to confusion and bugs.
- *Comments as a Crutch*: Relying too heavily on comments can act as a crutch for bad code. Instead of writing clean, self-explanatory code, developers might add comments to explain what the code is doing.

"A comment is a failure to express yourself in code. If you fail, then write a comment; but try not to fail."–Uncle Bob

Code Formatting Matters

- Consistent formatting improves readability.
- Use automated tools to enforce consistency.

Code Formatting Matters

```
data<-read.csv(file_path)
data<-na.omit(data)
filtered=data[data[[column_name]]>threshold,]
filtered[[new_col_name]]<-ifelse(filtered[[column_name]]>mean(filtered[[column_
"Above Avg","Below Avg")
summary_data=data.frame(Mean=mean(filtered[[column_name]])
,Median=median(filtered[[column_name]]),Count=nrow(filtered))
write.csv(filtered,paste0("cleaned_",basename(file_path)))
write.csv(summary_data,paste0("summary_",basename(file_path)))
```

Code Formatting Matters

```
data <- read.csv(file_path)

data <- na.omit(data)

filtered <- data[data[[column_name]] > threshold, ]

filtered[[new_col_name]] <- ifelse(filtered[[column_name]] > mean(filtered[[col
                                   "Above Avg",
                                   "Below Avg")

summary_data <- data.frame(Mean = mean(filtered[[column_name]]),
                           Median = median(filtered[[column_name]]),
                           Count = nrow(filtered))

write.csv(filtered,paste0("cleaned_", basename(file_path)))

write.csv(summary_data, paste0("summary_", basename(file_path)))
```

Recap

- Functions
- DRY
- Naming
- SRP and General Contractors
- Avoiding Side Effects
- Comments
- Formatting *MORE*