Lecture 3.8 – Refactoring

Specific Learning Objectives:

1.2.1 – Understand the way computers execute commands.

1.2.11 – Use a conditional statement to add exception handling to a function or script.

1.3.9 – Learn basic skills in debugging and troubleshooting error messages.

1.3.10 – Search for effective solutions and tools using online resources.

3.5 – Think and work independently with code.

What is ‘Refactoring’?

- **Refactoring** is the process by which code is *restructured* without changing what the code *produces.*
    - Improves readability and reduces complexity
    - Cleaner code that is easier to understand
    - Can help fix bugs and other problems
    - Often makes code more compact and versatile

- Refactoring code is like editing a paper. You’re not really changing what the paper says, but making in clearer and easier to understand!

What are the goals of refactoring?

Good goals 👍🏻

- ✅ Improve readability & understandability
- ✅ Breaking up code into logical pieces
- ✅ Reducing repetition & duplication
- ✅ Improving naming & location of code
- ✅ Improving documentation
- ✅ Bringing code up to best practices standards

NOT good goals 🙅🏼‍♀️

- ❌ “Rewriting” code
- ❌ Code golfing
- ❌ Optimization (different step!)
- ❌ Improving looks (interface)

When should you refactor code?

- ***After*** your code works and does the thing you want it to do.

- ***"Rule of 3"*** or ***"Three strikes and refactor"*** third time you have to use/copy the code, you refactor.

- ***Around*** the time you optimize the code.

- ***Around*** the time you have a code review.

- ***Before*** you give it to others who will use it.

- ***Before*** doing production runs of data analysis.

- ***Before*** publishing/leaving a lab/putting it down for several months.

Code Smells – Areas where refactoring can help

Smelly code 	**Nicer code** 
Too long or too short variable names	Rename during refactor
Commenting/uncommenting to alter behavior	Use conditionals for flow control
Using `attach()`	Try `with()` instead
Using `setwd()`	Use R Projects, standard organization
Excessive use of `if` and `else`	Try switch or early exits
Lots and lots of indentation	

R Style Guide http://adv-r.had.co.nz/Style.html

- **File names**
  - be long enough to be informative
  - end with .R or .Rmd
  - if need to be run in a sequence, prefixed with numbers

```
# Good
fit-models.R
utility-functions.R

# Bad
foo.r
stuff.r
```

```
0-download.R
1-parse.R
2-explore.R
```

- **Object names**
  - lower case, words separated by underscore _
  - variables are nouns, functions are verbs
  - concise and meaningful
  - avoid using previously defined functions and special values

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

```
# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

R Style Guide

http://adv-r.had.co.nz/Style.html

- **Spacing**
  - spaces should be used around operators (<-, +, -, /, *, =, etc)
  - put a space after commas
  - do not put spaces around : or ::

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

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

```r
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

- **Indentation**
  - use 80 characters per line or fewer
  - use 2 spaces to indent except for using a new line within a function, then indent to opening of function

```r
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

R Style Guide http://adv-r.had.co.nz/Style.html

- **Curly Braces**

  - opening curly braces should never go on their own line
  - closing braces should go on their own line unless followed by else or if else
  - always indent code inside braces
    - ok to leave short things on one line

```
# Good

if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

```
# Bad

if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

- **Assignments**

  - Use <- and not = for assignments

Examples of Refactoring

Example 1: variable names

Goals of Refactoring:

1. Making variable names more meaningful.
2. Conform variable names to style guide.

Example 2: improve readability

Goals of Refactoring:

1. Improve syntax by adding spaces.
2. Conform braces and brackets to style guide.

Example 3: commenting/uncommenting to alter behavior

Goals of Refactoring:

1. Add conditional to handle behavior switch

Check Your Understanding

Refactor the code in the lecture notebook so that it conforms to the style guide.

Standard Organization for R Projects

- **Standard organization** refers to the accepted way to organize parts of a project that helps orient users to your code.

MyProject

- ALL PARTS of the project belong in the main directory!

RProject file

- Established automatically when opening a new project

README file

- README files give instructions on how to run code to user and info about the project

data/

- directory containing all raw data (csv files)

src/

- directory containing all scripts (you might not have this)

results/

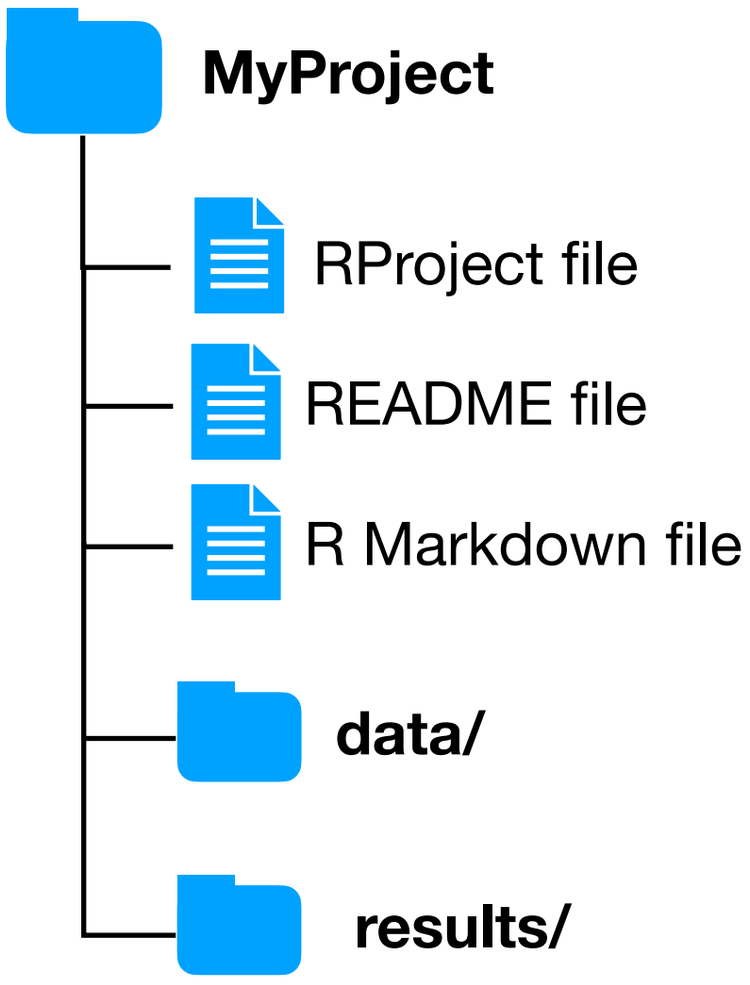
- directory containing all results produced by your code

doc/

- directory containing all documentation (including RMD files)

Standard Organization for R Projects

- **Modified standard organization** is acceptable for projects in this course!



- If you only have one R Markdown file, you can put it in the main directory. But data and results must still be separated!

In Class Exercises

1. **Take this time to reorganize your Project 2. Make it conform to standard organization (or modified standard organization).**

2. **Pick one section of your Project 2 code and refactor it. First, pick one or two goals for the refactor, then refactor. Rerun the code to make sure it still works!**

Action Items

1. **Complete previous assignments.**

2. **Read Davies Ch. 11 for next time.**