Organising Your Code

Data Science Workflows

Dr Zak Varty

Outline

Directories >> Files >> Contents

- Focus specifically on code
  - High level approaches to coding
  - Naming conventions
  - Style guides
  - Useful packages

1. Functional and Object Oriented Programming

Functional Programming

A functional programming style has two major properties:

- Object immutability,
- Complex programs written using function composition.

Mathematicians often find this way of operating quite intuitive:

$$y = g(x) = f_3 \circ f_2 \circ f_1(x).$$

The Pipe Operator

Function composition gets messy at ~3 functions

```r
log(exp(cos(sin(pi))))
```

The {magrittr} pipe operator facilitates this compositional style: %>%

```r
library(magrittr)
pi %>%
  sin() %>%
  cos() %>%
  exp() %>%
  log()
```

Recent addition of base R pipe |> (R >= 4.1), mostly behaves the same.

```r
iris |>
  head(n = 3)
```

When not to pipe !%>%

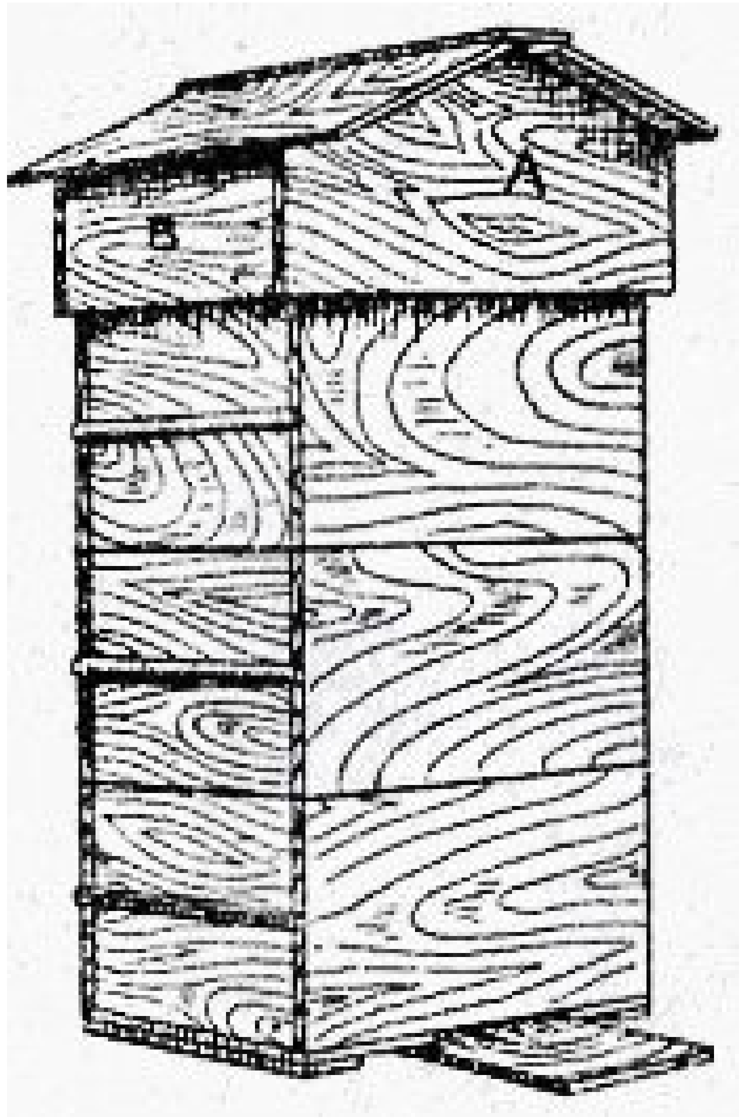
Avoid using the pipe when:

- Manipulating more than one object at a time. (Pipes are for a sequence of steps applied to a single primary object.)
- There are meaningful intermediate objects that could be given informative names.

Object Oriented Programming

The alternative to functional programming is object oriented programming.

- Solve problems by using lots of simple objects
- R has 3 OOP systems: S3, S4 and R6.
- Objects belong to a class, have methods and fields.
- Agent based simulation of beehive.



2. Structuring R scripts

Top of the Script

- Start script with a comment of 1-2 sentences explaining what it does.
- `setwd()` and `rm(ls())` are the devil's work.
  - "Session" > "Restart R" or Keyboard shortcut: crtl/cmd + shift + 0
- Polite to gather all `library()` and `source()` calls.
- Rude to mess with other people's set up using `install.packages()`.
- Portable scripts use paths relative to the root directory of the project.

Portable file paths with {here}

```r
# Bad - breaks if project moved
source("zaks-mbp/Desktop/exciting-new-project/src/helper_functions/rolling_mean.R")

# Better - breaks if Windows
source("../../src/helper_functions/rolling_mean.R")

# Best - but use here:here() to check root directory correctly identified
source(here::here("src","helper_functions","rolling_mean.R"))

# For more info on the here package:
vignette("here")
```

For even more on `{here}`, try: r-wtf chapter, r4ds chapter or project oriented workflow blog post.

3. Style Gudie Summary

The Body of the Code - Comments

Well commented and organised code easier to read and understand.

```r
# This is an example script showing good use of comments and sectioning

library(here)
source(here("src","helper_functions","rolling_mean.R"))

#===============================================================================  <- 80 characters max for readability
# Major Section on Comments ----
#===============================================================================

#-------------------------------------------------------------------------------
##  Minor Section on inline comments ----
#-------------------------------------------------------------------------------
x <- 1:10 # this is an inline comment

#-------------------------------------------------------------------------------
##  Minor Section on full line comments ----
#-------------------------------------------------------------------------------
rolling_mean(x)
# This is an full line comment
```

Naming Things Revisited: Objects = Nouns

Object names should use only lowercase letters, numbers, and `_`.

Use underscores (`_`) to separate words within a name. (`snake_case`)

Use singular over plural names.

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

Naming Things Revisited: Functions = Verbs

Function names should use only lowercase letters, numbers, and `_`.

Use underscores (`_`) to separate words within a name. (`snake_case`)

Suggest imperative mood, as in a recipe.

Break long functions over multiple lines. 4 vs 2 spaces.

```r
# Good
add_row()
permute()

# Bad
row_adder()
permutation()

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

Casing Consistently

Many options for separating words within names:

- `CamelCase`
- `pascalCase`
- `snakecase`
- `underscore_separated` ❤️
- `hyphen-separated`
- `point.separated` 💀

Style guide in brief

1. Use comments to structure your code
2. Objects = Nouns
3. Functions = Verbs
4. Use snake case and friendly grammar

Further tips for friendly coding

- Write your code to be easily understood by humans.

- Use informative names, typing is cheap.

```
# Bad
for(i in dmt){
  print(i)
}

# Good
for(temperature in daily_max_temperature){
  print(temperature)
}
```

- Divide your work into logical stages, human memory is expensive.

Tidyverse Style Guide

For further details on writing clean, readable code see the Tidyverse Style Guide.

4. Reduce, Reuse, Recylce

Writing reusable code (DRY coding)

- **If you do something twice, write a function.**
  - when you write a function, document it.
  - when you write a function, test it.
- **If your function is used in two scripts, it gets it's own script.**
  - name that script after the function & save it in the `src/` directory of your project
- **If your function is used across projects, add it to a package.**

Remembering how to use your own code

When you write a function, document it. What should the documentation contain?

- Inputs
- Outputs
- Example use cases
- Author (if not obvious or working in a team)

Roxygen for code documentation

The R package `{Roxygen}` can help with this.

1. `install.packages("Roxygen")`

2. With cursor inside function: Code > Insert Roxygen Skeleton

3. Keyboard shortcut: **cmd + option + shift + r** or **crtl + option + shift + r**

4. Fill out relevant fields

Roxygen Example

```r
#' Title
#'
#' @param x
#' @param remove_NA
#'
#' @return
#' @export
#'
#' @examples
geometric_mean <- function(x, remove_NA = FALSE){
  # Function body goes here
}
```

Roxygen Example

```r
#' Calculate the geometric mean of a numeric vector
#'
#' @param x numeric vector
#' @param remove_NA logical scalar, indicating whether NA values should be stripped be
#'
#' @return the geometric mean of the values in `x`, a numeric scalar value.
#'
#' @examples
#' geometric_mean(x = 1:10)
#' geometric_mean(x = c(1:10, NA), remove_NA = TRUE)
#'
geometric_mean <- function(x, remove_NA = FALSE){
  # Function body goes here
}
```

For more on Roxygen, see the package documentation or the chapter of R packages on function documentation.

Checking Your Code

If you write a function, test it. Testing code has two main purposes:

- To warn or prevent user misuse (e.g. strange inputs),
- To catch edge cases.

Testing is fun but painful - you are trying as hard as you can to break your beautiful new creation!

An Informal Testing Workflow

1. Write a function
2. Experiment with the function in the console, try to break it
3. Fix the break and repeat.

Problems: Time consuming and not reproducible.

Formalising Our Testing Workflow

- Test in a script, named after the function and stored in the `tests/` directory.
- The R package `{testthat}` provides useful functions for unit testing your code. Check out the {testthat} function reference for more examples.
- We'll explore this in more detail during the live session.

```r
testthat::expect_equal(
  object = geometric_mean(x = c(1, NA), remove_NA = FALSE),
  expected = NA)

# Error: geometric_mean(x = c(1, NA), remove_NA = FALSE) not equal to NA.
# Types not compatible: double is not logical
```

Summary

- Functional and Object Oriented Programming
- Structuring your scripts
- Styling your code
- Reduce, reuse, recycle
- Documenting and testing