MONASH University

ETC5523: Communicating with Data

Clearly communicating with code

Lecturer: *Emi Tanaka*

Department of Econometrics and Business Statistics

emi.tanaka@monash.edu

Week 8

cwd.numbat.space

Aim

- Understand and formulate code for communication
- Understand R package structure
- Build R package with R code, data, and launching shiny apps

Why

- Sharing code makes your analysis **transparent** and **reproducible** to others.
- Writing code that is readable by others *make author intent* ***explainable***
- An R package makes functions, data or apps **accessible**, thereby increasing impact of your work

Thanks to Stuart Lee for developing the initial content in this slide, which has been subsequently modified a fair amount by me.

Example data

Plant growth

An experiment to compare yields under control and two different treatment conditions on plants

```
str(PlantGrowth)
```

```
'data.frame':   30 obs. of  2 variables:
 $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
 $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Tooth growth

An experiment to study the effect of vitamin C on tooth growth in guinea pigs

```
str(ToothGrowth)
```

```
'data.frame':   60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

Which one do you prefer?

Code #1

```
 1 with(PlantGrowth, tapply(weight, group, mean))
```

```
 ctrl  trt1  trt2
5.032 4.661 5.526
```

Code #2

```
 1 library(tidyverse)
 2 PlantGrowth %>%
 3   group_by(group) %>%
 4   summarise(weight_avg = mean(weight))
```

```
# A tibble: 3 × 2
  group weight_avg
  <fct>      <dbl>
1 ctrl        5.03
2 trt1        4.66
3 trt2        5.53
```

What do you expect the output is?

Code #1

```
  1 with(ToothGrowth, tapply(len, list(supp, dose), mean))
```

```
     0.5     1     2
OJ 13.23 22.70 26.06
VC  7.98 16.77 26.14
```

Code #2

```
  1 library(tidyverse)
  2 ToothGrowth %>%
  3   group_by(supp, dose) %>%
  4   summarise(length_avg = mean(len))
```

```
# A tibble: 6 × 3
# Groups:   supp [2]
  supp   dose length_avg
  <fct> <dbl>      <dbl>
1 OJ      0.5       13.2
2 OJ      1         22.7
3 OJ      2         26.1
4 VC      0.5        7.98
5 VC      1         16.8
6 VC      2         26.1
```

Naming matters

Syntactic sugar

> **Syntactic sugar** means using function name or syntax in a programming language that is designed to make things ***easier to read or to express for humans***.

```
my_function(x)
```

What do you think this function is doing?

```
compute_average(x)
```

💡 A human reads your code, so write your function in a way that reads and works well for humans

Naming cases

camelCase

- Capitalise all words after first word.
- **Common in R Shiny** and JavaScript.

PascalCase

- Capitalise all words.
- Preferred by C programmers.

snake_case

- All words are lower case and separated by an underscore.
- **Preferred by R programmers** in general (except Shiny).

kebab-case

- All words are lower case and separated by a dash.
- Common in HTML attribute names and CSS property names.

Stick with the style convention of the language as much as possible!

Syntactically valid names in R

- In R, a **syntactically valid name** consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number.
- It also cannot be a **reserved word**, e.g. `if`, `else`, `TRUE`, `FALSE`, `while`, `function`. For full list see `?Reserved`.
- Anything else is a **non-syntactic name** in R and you can still use any name in R by surrounding it with backticks:

```
`4` <- 3
```

- You can use `make.names()` to make syntatically valid names in R.

Source: R Manual

Object names

- Variable and function names should **use only snake case**.
- Strive for names that are **concise** and **meaningful**.

Good ✔

```
day_one
day_1
```

Bad ✖

```
DayOne
dayone
first_day_of_the_month
djm1
```

Source: The tidyverse style guide

Function names

- Function names should be **verbs** (with exceptions).
- Avoid using . in the names…
- …unless writing a function method for S3 object system.

Good ✔

```
add_row()
permute()
add_column()
```

Bad ✖

```
row_adder()
permutation()
add.column()
```

Source: The tidyverse style guide

Variable names

- Variable names should be **nouns**.
- Consider using a list or data.frame to group variables in a similar context instead of assigning it as separate objects.

Good ✔

```
origin
fit
```

where

- `fit[[1]] = fit1`,
- `fit[[2]] = fit2`,
- and so on.

Bad ✖

```
originate
fit1
fit2
fit3
fit4
fit5
```

Readable code

Consistency is key!

Consistency spacing

- Always put a space after a comma, never before.

Good ✓

```
x[, 1]
```

Bad ✗

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

- Do not put spaces inside or outside parentheses for regular function calls.

Good ✓

```
mean(x, na.rm = TRUE)
```

Bad ✗

```
mean (x, na.rm = TRUE)
mean( x, na.rm = TRUE )
```

Source: The tidyverse style guide

Consistency spacing

- Place a space before and after `( )` when used with `if`, `for`, or `while`.

Good ✔

```
if (debug) {
  show(x)
}
```

Bad ✖

```
if(debug){
  show(x)
}
```

- Place a space after `( )` (but not before) used for function arguments:

Good ✔

```
function(x) {}
```

Bad ✖

```
function (x) {}
function(x){}
```

Source: The tidyverse style guide

Consistency spacing

- Most infix operators (+, –, <–, etc.) should be surrounded by spaces…

Good ✔

```
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)
```

Bad ✖

```
height<-feet*12+inches
mean(x, na.rm=TRUE)
```

- …with exceptions of operators with high precedence (::, :::, $, @, [, [[, ^, unary –, unary +, and :).

Good ✔

```
sqrt(x^2 + y^2)
df$z
x <- 1:10
```

Bad ✖

```
sqrt(x ^ 2 + y ^ 2)
df $ z
x <- 1 : 10
```

Source: The tidyverse style guide

Consistency spacing

- … with exceptions of single-sided formulas when the right-hand side is a single identifier.

Good ✔

```
~foo
tribble(
  ~col1, ~col2,
  "a",   "b"
)
```

Bad ✖

```
~ foo
tribble(
  ~ col1, ~ col2,
  "a", "b"
)
```

- Single-sided formulas with a complex right-hand side do need a space.

Good ✔

```
~ x + y
```

Bad ✖

```
~x + y
```

Source: The tidyverse style guide

Avoid long lines

- Limit your code to 80 characters per line.
- If the arguments to a function don't all fit on one line, put each argument on its own line and indent.

Good ✔

```
do_something_very_complicated(
  something = "that",
  requires = many,
  arguments = "some of which may be long"
)
```

Bad ✖

```
do_something_very_complicated("that", requires,
many, arguments, "some of which may be long")
```

Source: The tidyverse style guide

Sequence of functions

- Avoid deeply nesting functions in one line.
- %>% should always have a space before it, and should usually be followed by a new line.
- After the first step, each line should be indented by two spaces.

Good ✓

```
shopping_list %>%
  buy() %>%
  prepare() %>%
  cook()
```

Bad ✗

```
cook(prepare(buy(shopping_list)))

shopping_list %>% buy() %>%
  prepare() %>% cook()
```

Source: The tidyverse style guide

R packages for styling code

- `styler` allows you to interactively restyle selected text, files, or entire projects.
- `styler` includes an RStudio add-in, the easiest way to re-style existing code.
- `lintr` performs automated checks to confirm that you conform to the style guide.

What exactly are R packages?

R packages can be many things...

A container:

- for a set of R functions,
- to share data,
- to share an app,
- and more, e.g. Rmd templates (out of scope for this unit).

The anatomy of an R package

- `DESCRIPTION` file
- `R/` directory for R files that contain your functions
- `NAMESPACE` file (manual creation is out of scope for this unit)

Optionally,

- `data/`: for binary data available to the user
- `data-raw/`: for raw data
- `inst/`: for arbitrary additional files that you want include in your package.
- and others.

The `DESCRIPTION` file

Metadata for the package

- Package name
- Title and description of what the package does
- Authors
- Dependencies (depends, imports and suggests)
- Licencing
- Version number
- Where to report bugs and so on

Example: `dplyr` `DESCRIPTION` file

```
Type: Package
Package: dplyr
Title: A Grammar of Data Manipulation
Version: 1.0.99.9000
Authors@R:
    c(person(given = "Hadley",
             family = "Wickham",
             role = c("aut", "cre"),
             email = "hadley@rstudio.com",
             comment = c(ORCID = "0000-0003-4757-117X")),
      person(given = "Romain",
             family = "François",
             role = "aut",
             comment = c(ORCID = "0000-0002-2444-4226")),
      person(given = "Lionel",
```

The R/ directory

- The functions you create are stored as R scripts that live in the R/ directory.
- Functions can be internal to the package or exported so other users have access to them.
- See for example the `dplyr` R directory.

The NAMESPACE file

- The file contains a directive that describes whether an R object is exported from this package or imported from others.
- These directives can be automatically created by `roxygen2` (covered next lecture).

```
# Generated by roxygen2: do not edit by hand

S3method("$<-",grouped_df)
S3method("[",fun_list)
S3method("[",grouped_df)
S3method("[",rowwise_df)
S3method("[<-",grouped_df)
S3method("[<-",rowwise_df)
S3method("[[<-",grouped_df)
S3method("names<-",grouped_df)
S3method("names<-",rowwise_df)
S3method(add_count,data.frame)
S3method(add_count,default)
S3method(anti_join,data.frame)
```

Creating an R package

- `usethis::create_package("mypkg")` for creating a skeleton R package
- Add things to your R package
- `devtools::load_all()` for loading the functions in the R/ directory to the current environment

Adding functions to an R package

- `usethis::use_r("new-r-file")` for creating a new R file in the R/ directory

```
- mypackage
  |- R
     |- new-r-file.R
     |- ...
  |- DESCRIPTION
  |- ...
```

Distribute data via an R package

- `usethis::use_data_raw("filename")` for adding a file to data-raw/ directory to include code to reproduce data.
- `usethis::use_data(mydata)` for creating a binary data file in data/ directory.
- More information on this at R Packages (2e) Chapter 8 Data.

Launcing shiny app via an R package

```
- mypackage
  |- inst
     |- myapp
        |- app.R
  |- R
     |- run-app.R
     |- ...
  |- DESCRIPTION
  |- ...
```

run-app.R

```r
#' @export
run_app <- function() {
  app_dir <- system.file("myapp", package = "mypackage")
  shiny::runApp(app_dir, display.mode = "normal")
}
```

Note: for this to work, you need to first install the package!

Installing the package

- First run `devtools::document()` (we will cover this more next lecture).
- Then run `devtools::install()`.
- Now you can call `library(mypackage)` to use exported functions or data in your package!

Master the keyboard shortcuts

- **Cmd/Ctrl** + **Shift** + **L**: Load all
- **Cmd/Ctrl** + **Shift** + **D**: Document
- **Cmd/Ctrl** + **Shift** + **B**: Build and Reload
- plus more… see **RStudio IDE** > **Tools** > **Keyboard Shortcuts Help**

Week 8 Lesson

Summary

- How to enhance communication in code by adopting a consistent styling.
- R packages are flexible containers to share R code, data, and app amongst other things.
- An R package can be set up, documented, and tried out with `usethis` and `devtools`.

Resources

- Filazzola & Lortie (2022) A call for clean code to effectively communicate science
- R Packages 2nd edition by Hadley Wickham and Jenny Bryan