Good Coding Practices for Data Analysts

Heather Turner
Research Software Engineering Fellow
University of Warwick

@HeathrTurnr

16 November 2022

heatherturner.net/talks/NHS-R2022

Goals

In theory, writing scripts for data analysis makes our work

- Transparent
- Reproducible/reusable
- Maintainable

In practice, need to adopt good coding and software engineering practices!

Transparency

Project Organization

Organize your project as you would like to find it!

- Organize files by type (data, code, etc) to make it easy to navigate.
- Name files to reflect the content/function.

```
example_project
│
└── data
│   │   patient_outcomes.csv
│
└── outputs
│   │   summarized_outcomes.csv
│
└── reports
│   │   study_report.Rmd
│   │   study_report.docx
│
└── scripts
│   │   functions.R
│   │   analysis.R
```

Documentation

- Put a README at the top level of your project folder
- Comment your code to describe its purpose

```
# Patient exposure and event rate
patient_summary <- patient_outcomes |>
    group_by(STUDYID, COUNTRY, CENTRE, PT) |>
    summarise(d_exposure = max(d_exposure, na.rm = TRUE),
              exposure = (d_exposure/30.4), # calculate exposure per month
              event_count = sum(!is.na(EVENT)),
              event_rate = event_count/exposure)
```

- In RStudio, use `Ctrl`/⌘ + `Shift` + R to insert a section

```
# Pre-processing ----------------------------------------------------------
```

Readable code

- Use meaningful names
- Keep line length <80 characters and use white space around operators
- Use one chunk of code per objective
- Prefer readability over maximum efficiency

Efficient but complex

```
df$lag_value <- c(NA, df$value[-nrow(df)])
df$lag_value[which(!duplicated(df$group))] <- NA
```

More readable, slightly less efficienct

```
df |>
  group_by(group) |>
  mutate(lag_value = dplyr::lag(value))
```

Going further on transparency

- Style guides
  - Naming conventions, e.g. `snake_case` vs `camelCase`
  - Indentation
  - See e.g. The Tidyverse Style Guide
- Code review
- Pair programming
- Function documentation using the **docstring** package

Reproducibility/Reusability

Project-oriented workflow

In addition to organizing files within a project directory...

1. Set the working directory to the project root
    - Use RStudio Projects
    - Use `here::set_here()` to tag the project root with a `.here` file
2. Use file paths relative to the project root, to make your project portable
    - The here package makes this easy, e.g.

      ```
      ggsave(here("figs", "mpg_hp.png"))
      ```

    - If you need to use paths from outside the project, set these once at the start

Parameterized R Markdown/Quarto

```
---
title: "`r params$data` Dataset"
output: html_document
params:
  data: sleep
---


Summary of the `r params$data` dataset:
```

```
```{r summary-data, echo = FALSE}
report_data <- get(params$data)
summary(report_data)
```
```

```
---
title: "`r params$data` Dataset"
format: html
params:
  data: sleep
---


Summary of the `r params$data` dataset:
```

```
```{r}
#| label: summary-data
#| echo: false
report_data <- get(params$data)
summary(report_data)
```
```

Render with custom parameters

```r
rmarkdown::render("rmarkdown.Rmd",
  params = list(data = "sleep"))
```

sleep Dataset

Summary of the sleep dataset:

```
##      extra          group        ID
##  Min.   :-1.600   1:10   1      :2
##  1st Qu.:-0.025   2:10   2      :2
##  Median : 0.950          3      :2
##  Mean   : 1.540          4      :2
##  3rd Qu.: 3.400          5      :2
##  Max.   : 5.500          6      :2
##                          (Other):8
```

```r
quarto::quarto_render("quarto.qmd",
  execute_params = list(data = "women"))
```

women Dataset

Summary of the women dataset:

```
     height         weight
 Min.   :58.0   Min.   :115.0
 1st Qu.:61.5   1st Qu.:124.5
 Median :65.0   Median :135.0
 Mean   :65.0   Mean   :136.7
 3rd Qu.:68.5   3rd Qu.:148.0
 Max.   :72.0   Max.   :164.0
```

Defensive programming

Validate inputs, e.g.

```
# check a Excel file exists at given path
xlsx <- normalizePath(xlsx, winslash = "/", mustWork = TRUE)
# check a threshold is valid
stopifnot(is.numeric(threshold) && threshold >= 0)
```

The **assertthat** and **validate** packages can be useful here.

Check results of filters and joins

```
tab1 <- patient_outcomes |>
    filter(as.Date(DATE) == report_date & PT == patient)
if (!nrow(tab1))
    warning("No records for ", patient, " on ", report_date)
```

Package management

Most basic:

1. Add a `requirements.txt` at the root of the project.
2. Put `library()` calls at the top of `.R` and `.Rmd` files.

More advanced tools to specify and restore working environment:

1. *One-off analysis*: use **groundhog** to specify R, packages & dependencies by a **date**.
2. *Repeated analysis*: use **automagic** to install package versions specified in **deps.yaml**.
3. *Production code*: use **renv** to specify version R, packages & dependencies.

Maintainability

Choose dependencies carefully

Using a (non-base) package is always a trade-off:

For (e.g.)	Against
Better readability	Package update can break code
Faster implementation	Dependent on maintainer to fix bugs
Better error handling	More setup to reproduce analysis

- How much of the functionality are you using?
- How mature/well-maintained is the package?
- Are you using it across multiple projects?

Don't Repeat Yourself

Copy-pasting is error-prone and leads to over-complex code.

Use custom functions instead, e.g.

```
# convert counts to percentages in 2-way table with row/column totals
make_perc_tab <- function(tab){
    nr <- nrow(tab)
    nc <- ncol(tab)
    tab/tab[nr, nc] * 100
}
```

Makes it easier to re-use or iterate, e.g.

```
tab_list <- list(tab1, tab2, tab3)
out <- lapply(tab_list, make_perc_tab)
```

Version control

Version control systems (e.g. git) allow us to record changes made to files in a directory.

Heather Turner	0804d6e	Pre-processed KHK data	2013-03-20
Heather Turner	ebcd49d	added data for KHK project	2013-03-20
Heather Turner	306524f	added README file to start off	2013-03-20

- Avoid saving multiple variants or commenting out old code
- Commits can be restored temporarily or permanently
- Syncing with a remote repository (e.g. on GitHub) provides a backup

Testing

Tests can be used to custom functions act as expected, e.g.

```r
log_2 <- function(x) log(x, 2)
```

```r
library(testthat)
test_that("log_2 returns log to base 2", {
  expect_equal(log_2(2^3), 3)
})
```

```
## Test passed 🌈
```

Can create a test suite and run as `test_file("tests.R")`.

Helps to detect issues introduced by changes to the code.

Going further on maintainability

- Package development
  - Functions, documentation and tests in a shareable format
  - Easier to use across projects
- Using a repository host, e.g. GitHub
  - Use issues: note and discuss changes to make
  - Teamwork: work asynchronously and merge changes
  - Publish your code
  - Encourage external contribution

Resources

Good enough practices in scientific computing, Wilson et al, PLOS Computat. Biol., 2017.
The Turing Way : A Handbook for Reproducible Data Science, Arnold et al, 2022.
What They Forgot to Teach You About R, Bryan and Hester, 2021.
Why should I use the here package when I'm already using projects?, Barrett, 2018.
How to use Quarto for Parameterized Reporting, Mahoney, 2022.
Managing R script dependencies: automagic and renv, Cámara-Menoyo, 2022.
How to Use Git/GitHub with R, Keyes, 2021.
Happy Git and GitHub for the useR Bryan et al, 2022.