Writing Reproducible Code

Literate, linear programming and modularity

Daniela Palleschi
Leibniz-Zentrum Allgemeine Sprachwissenschaft

Thu Oct 17, 2024

Topics

- modular analyses and literate programming
- create and render a dynamic report with Quarto
- documenting your dependencies

Modular analyses

- recall our `scripts` folder (which you might've named `analysis` or something else)
- ideally, this would also contain subfolders, one for each stage of your analysis
  - or at least, multiple scripts
- this is the concept of *modularity* (Bowers & Voors, 2016; Nagler, 1995)
  - separating data cleaning, pre-processing, recoding, merging, analyses, etc. into files/scripts

Reproducible code

- how you write your code is the first step in making it reproducible
- the first principle is that your code must be *linear*
  - this means code must be written in a linear fashion
  - this is because we typically run a script from top-to-bottom

▶ Example

Writing linear code

- you need to load a package *before* you call a function from it
  - if we're just working in an R session, *before* means temporally prior
  - with linear code, *before* means higher up in the script
- such pre-requisite code must

  a. be present in the script

  b. appear above the first line of code that uses a function from this package
- missing pre-requisite code might not throw an error message
  - but might produce output we aren't expecting
  - e.g., forgetting to filter out certain observations
  - or forgetting *that* some observations have been filtered out

Literate programming

> Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.

— Knuth (1984), p. 97

- refers to writing and documenting our code so that humans can understand it
  - important for us: we are (generally) not professional programmers, nor are our peers
- we need to not only know what our code is doing when we look back at it in the future/share it
- the easiest way: informative comments
  - the length and frequency of these comments is your choice

Example R script

```
#| eval: false
#| echo: true
#| code-fold: false
#| code-summary: "Example"

# Analysis script for phoneme paper
# author: Joe DiMaggio
# date: Feb. 29, 2024
# purpose: analyse cleaned dataset

# Set-up ###

# load required packages
library(dplyr) # data wrangling
library(readr) # loding data
library(here) # project-relative file path

# Load-in data
df_phon <- read_csv(here("data", "phoneme_tidy_data.csv"))

# Explore data ###
summary(df_phone)
```

- begins with some meta-information about the document, including its purpose
  - aids in knowing which scripts to run in which sequence
- there are three hashtags after some headings (###)
  - this is helpful because it structures the outline of the document in RStudio
- the purpose of chunks of code are written above
  - description of specific lines of code are also given

Dynamic reports

- R scripts are useful, but don't show the code output
  - and commenting can get clunky
- dynamic reports combine prose, code, and code output
  - R markdown (`.Rmd` file extension) and Quarto (`.qmd`) are extensions of markdown
    - can embed R code 'chunks' in a script, thus producing 'dynamic' reports
  - produce a variety of output files which contain text, R code chunks, and the code chunk outputs all in one

Task: New Quarto document

1. Navigate to `File > New file > Quarto document`
2. Write some title, your name (Author), make sure 'Visual markdown Editor' is *unchecked*
3. Click 'Create'
4. A new tab will open in R Studio. Press the 'Render' button above the top of the document, you will be prompted to save the document. Store it in a folder called `scripts` and save it as `01-literate-programming.qmd`.
5. What happens?

R v. Rmarkdown v. Quarto

- `.R` files contain (R) source code only
- `.Rmd` files are *dynamic reports* that support
  - R-Code (and R-packages)
- `.qmd` files are *dynamic reports* (RStudio v2022.07 or later
  - R-Code (and R-packages)
  - native support for Python (and Jupyter-Notebooks)
  - native support for Julia

Check your RStudio version

Run the following in the Console: `RStudio.Version()$version`

- if the output is `2022.07` or higher you can use Quarto
- if not: update RStudio: `Help > Check for updates`

YAML

- the section at the very top fenced by ---
- contains all the meta information about your document
  - e.g. title, author name, date
  - also formatting information, e.g. type of output file
- there are many document formatting and customisation options that we won't cover in this course
- but for example I have many YAML formatting options in the source code of my slides

```
---
title: "My title"
---
```

YAML

YAML

1. change the title if you want to do so.
2. guess how to add a subtitle (hint: it is similar to adding a `title`)
3. add an author, `author: 'firstname lastname'` (see example below)
4. add a table of contents (Table of Contents = `toc`) by changing `format` so that it looks like this:

```
---
title: "Dynamic reports"
author: "Daniela Palleschi"
format:
  pdf:
    toc: true
---
```

5. Render the document. Do you see the changes?

Structure your reports

- remember to use (sub-)headings (e.g., `# Set-up`)
- describe the function/purpose at the beginning of the script
- document your train of thought and findings throughout the script
  - e.g., why are you producing this plot, what does it tell you?
- give an overview of the findings/end result at the end
- it's wise to avoid very long, multi-purpose scripts
  - rule of thumb: one script per product or purpose
  - e.g., data cleaning, exploration, analysis, publication figures, etc.

Code chunks

- the main benefit of dynamic reports: combining text with code (and code output)
- R code goes in code chunks:

```
1 ```{r}
2 2+2
3 ```
```

```
[1] 4
```

- to add a code chunk: `Code > Insert Chunk`
  - or use the keyboard shortcut: `Cmd+Opt+I` (Mac) / `Ctrl+Alt+I` (Windows)

Adding content

Adding structure and code chunks

1. Use the example R script above to create a structured document
   - use headings (#) and subheadings (##) accordingly
2. Load in our dataset in a code chunk
3. Render the document. Do you see the changes?

Documenting package dependencies

- R and R package versions are both open source, and are frequently updated
  - you might've run your code using `dplyr` version `1.1.0` or later, which introduced the `.by` per-operation grouping argument
  - what happens when somebody who has an older version of `dplyr` tries to run your code?
    - They won't be able to!
  - the reverse of this situation is more common:
    - a newer version of a package no longer supports a deprecated function or argument

Session info

- so, print your session info at the end of every script
  - this will print your R version, package versions, and more

```
sessionInfo()
```

Tips and tricks

- when you start a new script make sure you *always* start with a clean R environment: `Session > Restart R` or `Cmd/Ctrl+Shift+0`
  - this means no packages, data, functions, or any other dependencies are loaded
- at the top of your script, always load packages required below
  - you can always add more packages to the list as you add to your script
- **Render often**: when you make changes to your script make sure you re-render your document
  - checks you haven't introduced any errors
  - easier to troubleshoot if smaller changes have been made
- if you can run your script manually from source but it won't render, restart your R session and see if you can still run it from source
  - often the problem is some dependency in your environment that is not linearly introduced in the script

Hands-on: working with Quarto

Follow the instructions on the workshop website: Hands-on: working with Quarto

Topics 🏁

- modular analyses and literate programming ✅
- create and render a dynamic report with Quarto ✅
- documenting your dependencies ✅

Session Info

My session info.

```r
sessionInfo()
```

```
R version 4.4.1 (2024-06-14)
Platform: aarch64-apple-darwin20
Running under: macOS Sonoma 14.6

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.12.0

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: Europe/Berlin
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] digest_0.6.35     utf8_1.2.4        R6_2.5.1          fastmap_1.2.0
 [5] xfun_0.45         tzdb_0.4.0        magrittr_2.0.3    glue_1.7.0
 [9] tibble_3.2.1      knitr_1.47        pkgconfig_2.0.3   htmltools_0.5.8.1
[13] rmarkdown_2.27    lifecycle_1.0.4   readr_2.1.5       cli_3.6.2
```

Bowers, J., & Voors, M. (2016). How to improve your relationship with your future self. *Revista de Ciencia Política (Santiago)*, *36*(3), 829–848. https://doi.org/10.4067/S0718-090X2016000300011

Knuth, D. (1984). Literate programming. *The Computer Journal*, *27*(2), 97–111.

Nagler, J. (1995). Coding Style and Good Computing Practices. *PS: Political Science & Politics*, *28*(3), 488–492. https://doi.org/10.2307/420315