Reproducibility and an introduction to R Markdown

Data Science option of BIO00058M Data Analysis.

Emma Rand
University of York, UK

Outline

The aim of this topic is to introduce to good programming practices such as styling and organising your code and writing your own functions as well as the use of R Markdown (Allaire Xie, et al., 2019a; Xie Allaire, et al., 2018a) for creating reproducible analyses.

Reproducibility is just good scientific practice

Good scientific practice

Analysis workflows should conform to the same standards as lab projects and lab books[1]: structured and documented pipelines.

- Important in research collaboration and mandatory in many industry settings.
- Will likely become mandatory for science publication and funding.
- Will ultimately make your life much easier.
- Requires time, diligence and practice.

Higher standards IMO, it can be 100% reproducible so it should be. Publication should be contingent on the supply of complete analytical pipelines because science is funded by taxes and donations.

Reproducibility is a continuum

Some is better than none!

- Organise your project See Topic 1 Project Organisation.
- Script **everything**.
- Organise code into sections.
- Code formatting and style.
- Code algorithmically / algebraically.
- Document your project - code commenting, licenses, READMEs
- Modularise your code by writing functions.
- Use literate programming e.g., R Markdown
- Version control, containerising

Code formatting and style

Code formatting and style

"Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. " The tidyverse style guide

We have all written code which is hard to read. We all improve over time.

Hadley Wickham
@hadleywickham

The only way to write good code is to write tons of shitty code first. Feeling shame about bad code stops you from getting to good code

1,142 2:11 PM - Apr 17, 2015

947 people are talking about this

Code formatting and style

Some keys points:

- be consistent, emulate experienced coders
- use snake_case for variable names (not CamelCase, dot.case or kebab-case)
- use `<-` not `=` for assignment
- use spacing around most operators and after commas
- use indentation
- avoid long lines, break up code blocks with new lines
- use `"` for quoting text (not `'`) unless the text contains double quotes
- do use kebab-case in rmarkdown code chunk names (see markdown later)

Ugly code

Ugly code

```
data<-read_csv('../data-raw/Y101_Y102_Y201_Y202_Y101-5.csv',skip=2)
library(janitor);sol<-clean_names(data)
data=data%>%filter(str_detect(description,"OS=Homo sapiens"))%>%filter(x1pep=='x')
data=data%>%
mutate(g=str_extract(description,
"GN=[^\\s]+")%>%str_replace("GN=",''))
data<-data%>%mutate(id=str_extract(accession,"1::[^;]+")%>%str_replace("1::",""))
```

- no spacing or indentation
- inconsistent splitting of code blocks over lines
- inconsistent use of quote characters
- no comments
- variable names convey no meaning
- use of `=` for assignment and inconsistently
- multiple commands on a line
- library statement in the middle of 'analysis'

😎 Cool code 😎

Cool code

```r
########################## Data import ##########################
# define file name
filesol <- "../data-raw/Y101_Y102_Y201_Y202_Y101-5.csv"

# import: column headers and data are from row 3
sol <- read_csv(filesol, skip = 2) %>%
  janitor::clean_names()


########################## Tidy Data ##########################
# filter out the bovine proteins and those proteins
# identified from fewer than 2 peptides
sol <- sol %>%
  filter(str_detect(description, "OS=Homo sapiens")) %>%
  filter(x1pep == "x")

# Extract the genename from description column to a column
# of its own
sol <- sol %>%
  mutate(genename =  str_extract(description,"GN=[^\\s]+") %>%
           str_replace("GN=", ""))

# Extract the top protein identifier from accession column (first
# Uniprot ID after "1::") to a column of its own
sol <- sol %>%
  mutate(protid =  str_extract(accession, "1::[^;]+") %>%
           str_replace("1::", ""))
```

😎 Cool code Tips

- to correct indentation

  CONTROL+i

- to reformat code

  CONTROL+SHIFT+A

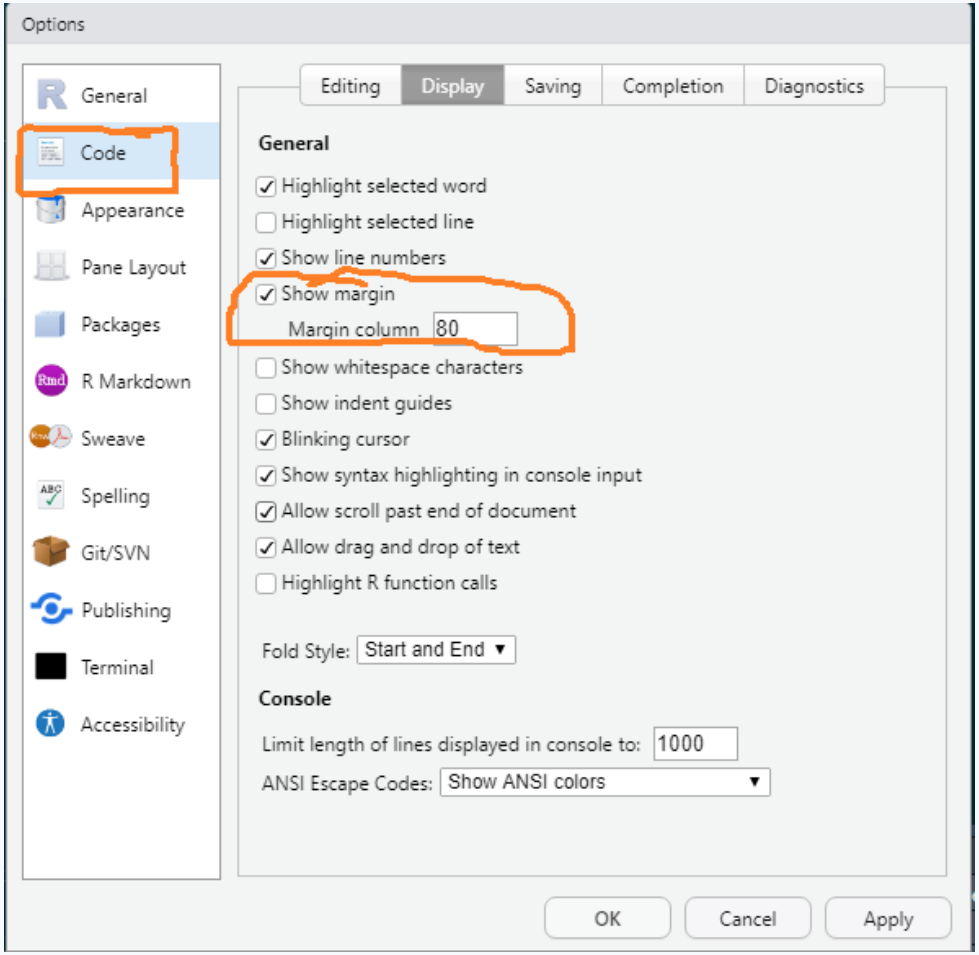
  Not perfect but corrects spacing, indentation, multiple commands on lines and assignment with `=`

- to comment and uncomment lines

  CONTROL+SHIFT+C

Cool code Tips

- show the margin Tools | Global Options | Code



Code 'algorithmically'

Code 'algorithmically.'

- Write code which expresses the structure of the problem/solution.
- Avoid hard coding numbers if at all possible - declare variables instead
- Declare frequently used values as variables at the start e.g., colour schemes, figure saving settings and chunk options (see R Markdown later)

Code 'algorithmically'

Hard coding numbers.

Suppose we want to calculate $SS(x)$ for the number of chicks in five nests. The formula is given by: $\sum(x_i - \bar{x})^2$

```
# calculating sum of squares for the number of
# chicks in nests
sum(3, 5, 6, 7, 8) / 5
```

```
## [1] 5.8
```

```
(3 - 5.8)^2 + (5 - 5.8)^2 + (6 - 5.8)^2 + (7 - 5.8)^2 + (8
```

```
## [1] 14.8
```

Code 'algorithmically'

Hard coding numbers

- if any of the sample numbers must be altered, all the code needs changing
- it is hard to tell that the output of the first line is a mean
- its hard to tell that the second number in each bracket comes from the first command
- it is hard to tell that 5 is just the number of values
- no way of know if numbers are the same by coincidence or they refer to the same thing

Code 'algorithmically'

😎 Better

```r
# calculating sum of squares for the number of
# chicks in nests
offspring <- c(3, 5, 6, 7, 8)
mean_offspring <- sum(offspring) / length(offspring)
sum((offspring - mean_offspring)^2)
```

```
## [1] 14.8
```

Code 'algorithmically'

😎 Better

- the commenting is the same but it is easy to follow
- if any of the sample numbers must be altered, only that number needs changing
- assigning a value you will later use to a variable with a meaningful name allows us to understand the first and second calculations.

Code 'algorithmically'

😎 Even better - use existing functions where you can

```
# calculating sum of squares for the number of
# chicks in nests
offspring <- c(3, 5, 6, 7, 8)
mean_offspring <- mean(offspring)
sum((offspring - mean_offspring)^2)
```

```
## [1] 14.8
```

Writing functions

Writing functions

Putting code that you will use more than once in a function is efficient and makes your code easier to read.

Suppose we had two samples of the number of chicks in nests. We could do this:

```
# calculating sum of squares for the number of
# chicks in nests
offspring_1 <- c(3, 5, 6, 7, 8)
mean_offspring_1 <- mean(offspring_1)
ss_offspring_1 <- sum((offspring - mean_offspring_1)^2)
offspring_2 <- c(4, 1, 6, 6, 3)
mean_offspring_2 <- mean(offspring_2)
ss_offspring_2 <- sum((offspring - mean_offspring_2)^2)
```

Writing functions

Instead, we can create a function for repeated use. Functions are useful because they generalise a process thus making it reproducible without copying and pasting.

A function is defined by an assignment of the form:

```
functionname <- function(arg1, arg2, ...) {
  expression
  }
```

The `expression` is any R code that uses the arguments (arg1 etc) to calculate a value. In our case it will be the formula:
$SS(x) = \sum(x_i - \bar{x})^2$

Writing functions

We define our $SS(x)$ function like this:

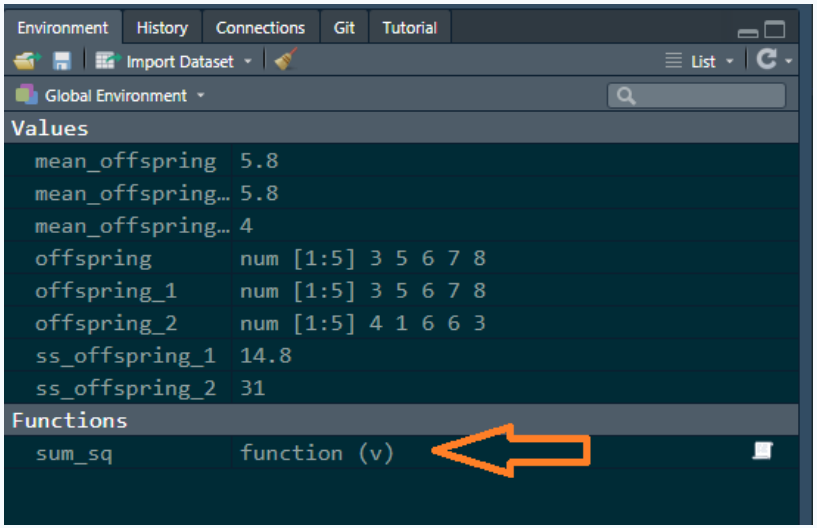
```
sum_sq <- function(v) {
  sum((v - mean(v))^2)
  }
```

I chose `v`, as a name, arbitrarily. It doesn't matter what you call it. It only exists inside the function when the function is called. It acts as a place holder for the thing that you pass in.

Our function take one argument. The expression describes what the function should do with the argument passed.

Writing functions

When you run the code that defines the function, it will appear in your environment window.



Writing functions

To call the function:

```
sum_sq(offspring_1)
```

```
## [1] 14.8
```

```
sum_sq(offspring_2)
```

```
## [1] 18
```

Writing functions

You can put your functions in one of two places:

1. at the beginning of your analysis in a section for defining functions.
2. better: put the function in a file of its own then `source()` it before running it:

```
source("functions/sum_sq.R")
sum_sq(offspring_1)
```

```
## [1] 14.8
```

Writing functions

Save the function in a file with the same name as the function.

Then put the `source()` command at the top of the script (like the package loading)

R Markdown

Reproducible Reports: R Markdown

How do you work?

- What program do you analyse your data in?
- What program do you plot your data in?
- What program do you use to write up the results to submit for assessment or to a journal?
- What is your process for getting your summary data, statistical results, tables and figures in to your report?
- What do you do when you get additional data that increases your sample sizes?
- What do you do if you wrote in Word formatted for one journal and now have to submit in PDF formatted for another?

Reproducible Reports: R Markdown

Typically people analyse, plot and write up in different programs.

Graphs are saved to files and copied and pasted into the final report.

This process relies on manual labour.

If the data changes, or you want to add a table or figure, you must repeat the entire process to update the report and renumber all your figures and all the references to figures.

If you have ever had do this, you'll know it is time consuming but also very error prone.

Reproducible Reports: R Markdown

The brilliance of R Markdown (Allaire Xie, et al., 2019a) is that you can use a **single R Markdown file** to:

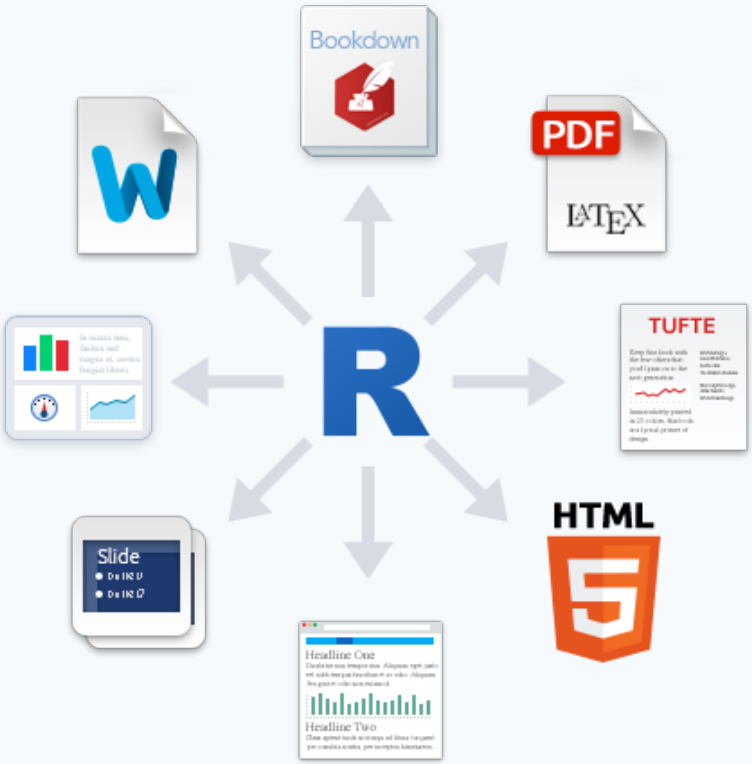
- save and execute code
- do all your data processing, analysis and plotting
- generate high quality reports that can be shared with an audience

This is known as literate programming (Knuth, 1984).

😎

Reproducible Reports: R Markdown

Many output formats are supported!



- Word, PDF, journal article formats for many journals
- Webpages - many styles and themes
- webslides
- powerpoint
- books
- blogs
- posters
- web applications including interactivity

Reproducible Reports: R Markdown

🎬 This might be a good time to watch the demo

Reproducible Reports: R Markdown

Key points from the demo. R Markdown.....

- mixes text and code
- is human readable
- has a YAML header, containing "metadata", between the ---
- code chunk options control whether the code and its output end up in your 'knitted' document
- comments
  - in a code chunk the `#` is used for comments as normal
  - in text comments are written like this: `<!-- a comment -->`
  - but Ctrl+Shift+C will do context-specific comments
- # in the text indicate headings
- Formatting `**bold**`, `*italics*`

Reproducible Reports: R Markdown

Create your own R markdown file

- File | New File | R Markdown
- Delete everything except the YAML header the first code chunk
- Add your name, and a title

Reproducible Reports: R Markdown

🎬 Edit the YAML. I recommend using the `bookdown` package (Xie, 2016) for output types which handle cross referencing well. We will do cross referencing in the next Topic.

```
---
title: "My awesome title"
author: "Emma Rand"
output:
  bookdown::pdf_document2: default
  bookdown::word_document2: default
  bookdown::html_document2: default
---
```

Reproducible Reports: R Markdown

🎬 Set some **default** code chunk options. I recommend these:

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE,
                      warning = FALSE,
                      message = FALSE,
                      fig.retina = 3)
```

`echo = FALSE` code will not be included in the output - normally what you want in a report.

`warning = FALSE` and `message = FALSE` R messages and warnings will not be included

`fig.retina = 3` for improving the appearance of R figures in HTML documents

Organise code into sections.

Have separate named code chunks for each process: set up, package loading, data import, data tidying (maybe several chunks), different analyses, figures etc.

The first two code chunks are usually for the default code chunk options (which I tend to call `setup`) and for package loading.

Reproducible Reports: R Markdown

🎬 Use Insert | R to add a code chunk:

```
```{r packages}
library(tidyverse)
```
```

- `r` indicates it is an R code chunk
- `packages` is just a name for the chunk. Naming chunks makes debugging easier.

Summary

- R Markdown interweaves analysis code and reporting and is human readable
- metadata about the document is given YAML header
- there are many formats for outputs and several packages available
- code chunk behaviour can be set
- `#` is a heading in the text
- Use Ctrl+Shift+C for do context-specific comments
- Formatting: `**bold**`, `*italics*`
- Organise code into sections
- Use a consistent style particularly in terms of indentation, spacing and variable names
- Modularise your code by writing functions
- Code algorithmically

Reading

Strongly recommended

- Good enough practices in scientific computing (Wilson Bryan, et al., 2017)
- The tidyverse style guide (Wickham,) Chapter 2 Syntax

References

Allaire, J, Y. Xie, et al. (2019a). *rmarkdown: Dynamic Documents for R*. R package version 1.16. URL: https://github.com/rstudio/rmarkdown.

Knuth, D. E. (1984). "Literate Programming". In: *Comput. J.* 27.2, pp. 97-111.

Wickham, H. "The tidyverse style guide". . Accessed: 2020-10-30.

Wilson, G, J. Bryan, et al. (2017). "Good enough practices in scientific computing". En. In: *PLoS Comput. Biol.* 13.6, p. e1005510.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. ISBN 978-1138700109. Boca Raton, Florida: Chapman and Hall/CRC. URL: https://github.com/rstudio/bookdown.

References

Xie, Y, J. Allaire, et al. (2018a). *R Markdown: The Definitive Guide*. ISBN 9781138359338. Boca Raton, Florida: Chapman and Hall/CRC. URL: https://bookdown.org/yihui/rmarkdown.

Slides made with with xaringan (Xie, 2019) and xaringanExtra (Aden-Buie, 2020)

Emma Rand

emma.rand@york.ac.uk

Twitter: @er13_r

GitHub: 3mmaRand

blog: https://buzzrbeeline.blog/