Lesson 1: Adopting principles of reproducible research

Patrick Mathias

What is reproducible research?

In its simplest form, reproducible research is the principle that any research result can be reproduced by anybody. Or, per Wikipedia: "The term reproducible research refers to the idea that the ultimate product of academic research is the paper along with the laboratory notebooks and full computational environment used to produce the results in the paper such as the code, data, etc. that can be used to reproduce the results and create new work based on the research."

Reproducibility can be achieved when the following criteria are met (Marecelino 2016): - All methods are fully reported - All data and files used for the analysis are available - The process of analyzing raw data is well reported and preserved

But I'm not doing research for a publication, so why should I care about reproducible research?

- Someone else may need to run your analysis (or you may want someone else to do the analysis so it's less work for you)
- You may want to improve on that analysis
- You will probably want to run the same exact analysis or a very similar analysis on the same data set or a new data set in the future

"Everything you do, you will probably have to do over again." (Noble 2009)

There are three practices we will cover in this lesson to help get your code to be more reproducible and reusable:

- Develop a standardized but easy-to-use project structure
- Adopt a style convention for coding
- Enforce reproducibility when working with projects and packages

Develop a standard project structure

In their article "Good enough practices in scientific computing", Wilson et al. highlight useful recommendations for organizing projects (Wilson 2017):

- **Put each project in its own directory, which is named after the project**
- Put text documents associated with the project in the doc directory
- **Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory**
- Put project source code in the src directory
- Put compiled programs in the bin directory
- **Name all files to reflect their content or function**

Because we are focusing on using RMarkdown, notebooks, and less complex types of analyses, we are going to focus on the recommendations in bold in this course. All of these practices are recommended and we encourage everyone to read the original article to better understand motivations behind the recommendations.

Put each project in its own directory, which is named after the project

Putting projects into their own directories helps to ensure that everything you need to run an analysis is in one place. That helps you minimize manual navigation to try and tie everything together (assuming you create the directory as a first step in the project).

What is a project? Wilson et al. suggest dividing projects based on "overlap in data and code files." I tend to think about this question from the perspective of output, so a project is going to be the unit of work that creates an analysis document that will go on to wider consumption. If I am going to create multiple documents from the same data set, that will likely be included in the same project. It gets me to the same place that Wilson et al. suggest, but very often you start a project with a deliverable document in mind and then decide to branch out or not down the road.

Now that we're thinking about creating directories for projects and directory structure in general, let's take the opportunity to review some basic commands and configuration related to directories in R.

Exercise 1

1. Navigate to your **Preferences** for the RStudio application and note the *Default working directory (when not in a project)*
2. Navigate to your Console and get the working directory using `getwd()`
3. Review the contents of your current folder using `list.files()`
4. Now try to set your working directory using `setwd("test_dir")`. What happened?
5. Create a new test directory using `dir.create("test_dir")`
6. Review your current directory
7. Set your directory to the test directory you just created
8. Using the Files window (bottom right in RStudio, click on **Files** tab if on another tab), navigate to the test directory you just created and list the files. *Pro tip: The More menu here has shortcuts to set the currently displayed directory as your working directory and to navigate to the current working directory*
9. Navigate back to one level above the directory you created using `setwd("..")` and list the files
10. Delete the directory you created using the `unlink()` function. Learn more about how to use the function by reviewing the documentation: `?unlink`. Pay special attention to comments about deleting directories.

Optional Exercise (If you do not already have a project directory)

Now that you're warmed up with navigating through directories using R, let's use functionality that's built into RStudio to make our project-oriented lives easier. To enter this brave new world of project directories, let's make a home for our projects. (Alternately, if you already have a directory that's a home for your projects, set your working directory there.) 1. Using the Files navigation window (bottom right, Files tab), navigate to your home directory or any directory you'd like to place your future RStudio projects 2. Create a "Projects" directory 3. Set your directory to the "Projects" directory

```
dir.create("Projects")
setwd("/Projects")
```

Alternately, you can do the above steps within your operating system (eg. on a Mac, open Finder window and create a folder) or if you are comfortable working at the command line, you can make a directory there. In the newest version of RStudio (version 1.1), you have the option of opening up a command line prompt under the Terminal tab (on the left side, next to the Console tab).

Exercise 2

Let's start a new project : 1. Navigate to the **File** menu and select **New Project...** OR Select the **Create a project** button on the global toolbar (2nd from the left) 2. Select **New Directory** option 3. In the Project Type prompt, select **New Project** 4. In the Directory Name prompt under Create New Project, enter "intermediate-R-course" 5. In the Create Project as a Subdirectory of prompt under Create New Project, navigate to the Projects folder you just created (or another directory of your choosing). You can type in the path or hit the **Browse** button to find the directory. Check the option for "Open in a new session" and create your project.

So, what exactly does creating a Project in RStudio do for you? In a nutshell, using these Projects allows you to drop what you're doing, close RStudio, and then open the Project to pick up where you left off. Your data, history, settings, open tabs, etc. will be saved for you automatically.

Does using a RStudio Project allow someone else to pick up your code and just use it? Or let you come back to a Project 1 year later and have everything work magically? Not by itself, but with a few more tricks you will be able to more easily re-run or share your code.

Put raw data and metadata in a data directory and files generated during cleanup and analysis in a results directory

Before we broke up with Excel, it was standard operating procedure to perform our calculations and data manipulations in the same place that our data lived. This is not necessarily incompatible with reproducibility, if we have very careful workflows and make creative use of macros. However, once you have modified your original input file, it may be non-trivial to review what you actually did to your original raw data (particularly if you did not save it as a separate file). Morever, Excel generally lends itself to non-repeatable manual data manipulation that can take extensive detective work to piece together.

Using R alone will not necessarily save you from these patterns but they take a different form. Instead of clicking around, dragging, and entering formulas, you might find yourself throwing different functions at your data in a different order each time you open up R. While it takes some effort to overwrite your original data file in R, other non-ideal patterns of file management that are common in Excel-land can creep up on you if you're not careful.

One solution to help avoid these issues in maintaining the separation of church and state (if I may use a poor analogy) is to explicitly organize your analysis so that raw data lives in one directory (the *data* directory) and the results of running your R code are placed in another directory (eg. *results* or *output*). You can take this concept a little further and include other directories within your project folder to better organize work such as *figures*, *documents* (for manuscripts), or *processed_data/munge* (if you want to create intermediate data sets). You have a lot of flexibility and there are multiple resources that provide some guidance (Parzakonis 2017), (Muller 2017), (Software Carpentry 2016).

> Exercise 3
>
> Let's go ahead and create a minimal project structure:

```
dir.create("data") # raw data
dir.create("output") # output from analysis
dir.create("cache") # intermediate data (after processing raw data)
dir.create("src") # code goes into this folder
```

This is a bare bones structure that should work for our purposes.

Further exploration/tools for creating projects: There is also a dedicated Project Template package that has a nice "minimal project layout" that can be a good starting point if you want R to do more of the work for you: Project Template. This package duplicates some functionality that the RStudio Project does for you, so you probably want to run it outside of an RStudio Project but it is a good tool to be aware of.

Name all files (and variables) to reflect their content or function

This concept is pretty straightforward: assume someone else will be working with your code and analysis and won't intuitively understand cryptic names. Rather than output such as results.csv, a file name of morphine_precision_results.csv offers more insight. Wilson et al. make the good point that using sequential numbers will come back to bite you as your project evolves: for example, "figure_2.txt" for a manuscript may eventually become "figure_3.txt". We'll get into it in the next section but the final guidance with regards to file names is to using a style convention for file naming to make it easier to read names an manipulate files in

R. One common issue is dealing with whitespace in file names: this can be annoying when writing out the file names in scripts so underscores are preferrable. Another issue is the use of capital letters: all lowercase names is easier to write out. As an example, rather than "Opiate Analysis.csv", the preferred name might be "opiate_analysis.csv".

Adopt a style convention for coding

Reading other people's code can be extremely difficult. Actually, reading your own code is often difficult, particularly if you haven't laid eyes on it long time and are trying to reconstruct what you did. One thing that can help is to adopt certain conventions around how your code looks, and style guides are handy resources to help with this. Google has published an R Style Guide that has been a long-standing resource and nice to refer to, but since we are immersing ourselves in the tidyverse, we will recommend the Tidyverse style guide.

Some highlights: - Use underscores to separate words in a name (see above comments for file names) - Put a space before and after operators (such as `==`, `+`, `<-`), but there are a few exceptions such as `^` or `:` - Use `<-` rather than `=` for assignment - Try to limit code to 80 characters per line & if a function call is too long, separate arguments to use one line each for function, arguements, and closing parenthesis.

```
# 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"
                              )
```

While we're talking about style conventions, let's take a little diversion to discuss a common element of code in the tidyverse that you may not be familiar with: the almighty pipe `%>%`. The pipe allows you to chain together functions sequentially so that you can be much more efficient with your code and make it readable. Here is an example (with imaginary functions) adapted from the tidyverse style guide:

```
# one way to represent a hop, scoop, and a bop, without pipes
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
# another way to represent the same sequence with less code but in a less readable way
foo_foo <- bop(scoop(hop(foo_foo, through = forest), up = field_mice), on = head)

# a hop, scoop, and a bop with the almight pipes
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

Pipes are not compatible with all functions but should work with all of the tidyverse package functions (the magrittr package that defines the pipe is included in the tidyverse). In general, functions expect data as the primary argument and you can think of the pipe as feeding the data to the function. From the perspective of coding style, the most useful suggestion for using pipes is arguably to write the code so that each function is on its own line. The tidyverse style guide section on pipes is pretty helpful.

You're not alone in your efforts to write readable code: there's an app for that! Actually, there are packages for that, and multiple packages at that. We will not discuss in too much depth here but it is good to be

aware of them: - styler is a package that allows you to interactively reformat a chunk of code, a file, or a directory - lintr checks code in an automated fashion while you are writing it

So, if you have some old scripts you want to make more readable, you can unleash styler on the file(s) and it will reformat it. Functionality for lintr has been built into more recent versions of RStudio.

Enforce reproducibility of the directories and packages

Scenario 1: Sharing your project with a colleague

Let's think about a happy time a couple months from now. You've completed this R course, have learned some new tricks, and you have written an analysis of your mass spec data, bundled as a nice project in a directory named "mass_spec_analysis". You're very proud of the analysis you've written and your colleague wants to run the analysis on similar data. You send them your analysis project (the whole directory) and when they run it they immediately get the following error when trying to load the data file with the `read.csv("file.csv")` command: Error in file(file, "rt") : cannot open the connection In addition: Warning message: In file(file, "rt") : cannot open file 'file.csv': No such file or directory

Hmmm, R can't find the file, even though you set the working directory for your folder using `setwd("/Users/username/path/to/mass_spec_analysis")`.

What is the problem? Setting your working directory is actually the problem here, because it is almost guaranteed that the path to a directory on your computer does not match the path to the directory on another computer. That path may not even work on your own computer a couple years from now!

Fear not, there is a package for that! Let's install the here package to help solve this problem and explore how it works.

Exercise 4

1. Install the package
2. Load the package. What does the output tell you?
3. Run the command `here()`. What does the output tell you?

```
install.packages("here")
library(here)
here()
```

The here package uses a pretty straightforward syntax to help you point to the file you want. In the example above, where file.csv is a data file in the root directory (I know, not ideal practice per our discussion on project structure above), then you can reference the file using `here("file.csv")`, where `here()` indicates the current directory. So reading the file could be accomplished with `read.csv(here("file.csv"))` and it could be run by any who you share the project with.

The here package couples well with an RStudio Project because there is an algorithm that determines which directory is the top-level directory by looking for specific files - creating an RStudio Project creates an .Rproj file that tells here which is the project top-level directory - if you don't create a Project in RStudio, you can create an empty file named .here in the top-level directory to tell here where to go - there are a variety of other file types the package looks for (including a .git file which is generated if you have a project on Github)

I encourage you to read the following post by Jenny Bryan that includes her strong opinions about setting your working directory: Project-oriented workflow.

Moral of the story: avoid using `setwd()` and complicated paths to your file - use `here()` instead!

Scenario 2: Running your 2018 code in 2019

Now imagine you've written a nice analysis for your mass spec data but let it sit on the shelf for 6 months or a year. In the meantime, you've updated R and your packages multiple times. You rerun your analysis on the same old data set and either (a) one or more lines of code longer works or (b) the output of your analysis is different than the first time you ran it. Very often these problems arise because one or more of the packages you use in your code have been updated since the first time you ran your analysis. Sometimes package updates change the input or output specific functions expect or produce or alter the behavior of packages in unexpected ways. These problems also arise when sharing code with colleagues because different users may have different versions of packages loaded.

Don't worry, there is a package for that! Probably the most lightweight solution to this problem is the checkpoint package. The basic premise behind checkpoint is that it allows you use the package as it existed at a specific date. There is a snapshot for all packages in CRAN (the R package repository) each day, dating back to 2017-09-17. By using checkpoint you can be confident that the version of the package you reference in your code is the same version that anyone else running your code will be using.

Let's run through some basics of using the package.

> Exercise 5

1. Install the checkpoint package

```
install.packages("checkpoint")
```

2. Load the checkpoint package
3. Read the help for the checkpoint function

```
library(checkpoint)
?checkpoint

# below is sample code but we're not quite ready to run it (yet)
# library(checkpoint)
# checkpoint("2018-01-01", checkpointLocation = tempdir())
# checkpoint function takes the date of the package you want as the primary argument
# and a location for the directory to load the packages into
#
# library(dplyr)
# library(lubridate)
```

The behavior of checkpoint makes it complicated to test out in this section: the package is tied to a project and by default searches for every package called within your project (via `library()` or `require()`). However, if you refer to the setup code chunks for this course you will see how checkpoint works in the wild.

The checkpoint package is very helpful in writing reproducible analyses, but there are some limitations/considerations with using it: - retrieving and installing packages adds to the amount of time it takes to run your analysis - package updates over time may fix bugs so changes in output may be more accurate - checkpoint is tied to projects, so alternate structures that don't use projects may not able to utilize the package

In the next lesson we will dive into more detail about how to use checkpoint in the context of an R Markdown document, so this is just a brief introduction to the concept of using checkpoint to make your code reproducible.

Summary

- Reproducible research is the principle that any research result can be reproduced by anybody

- Practices in reproducible research also offer benefits for to the code author in producing clearer, easier to understand code and being able to easily repeat past work
- Important practices in reproducible research include:
  - Developing a standardized but easy-to-use project structure
  - Adopting a style convention for coding
  - Enforcing reproducibility when working with projects and packages