How to structure analytical pipelines

Tad Dallas

Reading for this week:

> Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work reproducibly using R (and friends). The American Statistician, 72(1), 80-88.

What is an analytical pipeline?

The goal of this course is not only to teach `R` as a scientific programming language, but to help make your analyses *reproducible* and *portable*. We defined *reproducibility* as the ability to recreate an analysis and obtain the same output. Here, *portability* means that the analyses should be *reproducible* on any machine and any operating system (OS). As a step towards this, it is important to structure the analytical files in a way that they are entirely self-contained, and are clearly written so that anyone can take your code, run it, and reproduce the analyses.

There are a huge number of benefits to defining your project workflow or "analytical pipeline" as I refer to it. The biggest of these, in my opinion, is that the same template can be used for multiple projects, as your pipeline should be able to be carried over to a new project easily. It also allows collaborators to contribute to your code or manuscript files easily, since they will be able to see and understand the workflow.

Writing R scripts

The basic way to write an R script is to have file ending in `.R` which contains the code needed to run a specific analysis. I encourage you to consider using `R markdown`, which is an enhanced `R` script that allows you to include text directly into your `R` script, and can be compiled to html, pdf, and many other output formats. This is useful because it allows your code to not simply be code, but to be a narrative of what you did, why you did it, and what you found. Many researchers now write manuscripts in `R markdown`, as it allows the manuscript file and the analysis file to be the same file. `R markdown` scripts end in `.Rmd`. This lecture is written as an `.Rmd` file, and compiled to pdf for your viewing. Recall what we learned about markdown earlier. Now, to include `R` code, you will have to write code blocks, which start with three backtick symbols (‘).

Writing functions

We learned the basics of `R` coding previously, ending with how to write functions. Now we will go into a bit more detail as it pertains to your workflow, in that each function should be documented, such that it is portable across analyses and the input and output are clearly defined. We will start with a simple function structure

```
yourFunction <- function(arguments){

    # doing stuff
    ret <- NA

    # returning stuff
```

```
    return(ret)
}
```

Documenting functions

There are multiple forms of documentation. In this chapter, you'll learn about object documentation, as accessed by ? or help(). Object documentation is a type of reference documentation. It works like a dictionary: while a dictionary is helpful if you want to know what a word means, it won't help you find the right word for a new situation. Similarly, object documentation is helpful if you already know the name of the object, but it doesn't help you find the object you need to solve a given problem. That's one of the jobs of vignettes, which you'll learn about in the next chapter.

R provides a standard way of documenting the objects in a package: you write .Rd files in the man/ directory. These files use a custom syntax, loosely based on LaTeX, and are rendered to HTML, plain text and pdf for viewing. Instead of writing these files by hand, we're going to use roxygen2 which turns specially formatted comments into .Rd files. The goal of roxygen2 is to make documenting your code as easy as possible. It has a number of advantages over writing .Rd files by hand

Run `devtools::document()`

```
#' Title of function
#'
#' @param x a number
#' @param y a number
#'
#' @return sum of x and y
#'
#' @examples
#'   addXY(5,10)
#' @export

addXY <- function(x,y){
    return(sum(c(x,y)))
}
```

Structuring your functions as such will allow you to write clear code, but also to establish code that can be used across projects. For instance, I have written a simple `R` package containing functions I commonly use across projects. Calling in this package gives me access to all my common functions, which you will come to realize is a huge time-saver when you would otherwise be reinventing the wheel with each analysis. One important note here is to write functions in such a way that they can take general input. This means you should:

- **avoid hard-coding**: this refers to writing functions that will *only* work with a specific data structure. For instance, explicitly using a column index in a function is bad practice, while including an argument to the function that allows flexible indexing is better

```
# bad
getColumnMean <- function(mat){
    mean(mat[,3])
}

# better
getColumnMean <- function(mat, index=3){
    mean(mat[,index])
}
```

- **handle potential errors**: errors are inherent in programming, which can be frustrating, but can be managed. Ideally, the code you write should handle errors, NAs, and other things that affect function output. For instance, what happens if we try to take the mean of a character vector, or a numeric vector containing an NA value?

```
# bad
getColumnMean <- function(mat, index=3){
    mean(mat[,index])
}

# better
getColumnMean <- function(mat, index=3){
    if(any(is.na(mat[,index]))){
        warning('NA values present...ignoring them')
    }
    mean(na.omit(mat[,index]))
}
```

Here, we write a `warning` statement that tells the user that there is an NA value present, and then we ignore it to calculate the mean of the column.

- **be flexible**: Flexibility comes at a cost, so first you will need to decide the scope with which you want the function to be flexible. However, it is good practice to strive for flexibility in your code. What if we did not simply want the first moment (the mean) of a column, but wanted to decide arbitrarily what action was taken on the column?

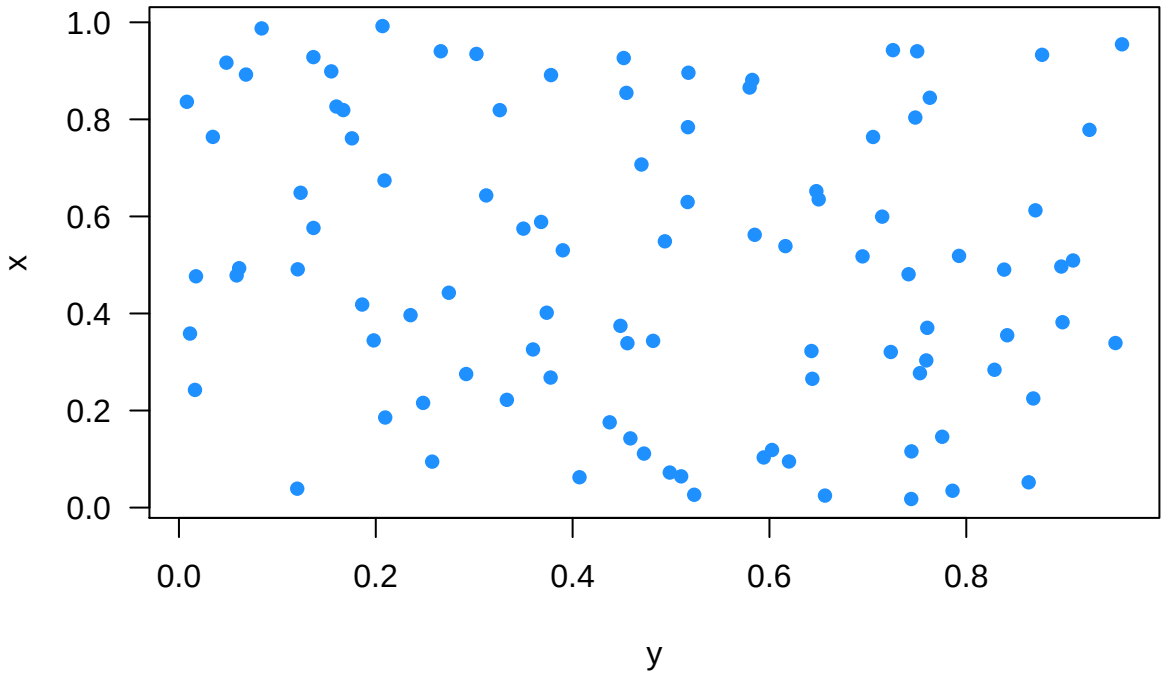
```
# bad
getColumnMean <- function(mat, index=3){
    if(any(is.na(mat[,index]))){
        warning('NA values present...ignoring them')
    }
    mean(na.omit(mat[,index]))
}

# better
getColumnInfo <- function(mat, index=3, func=mean){
    if(any(is.na(mat[,index]))){
        warning('NA values present...ignoring them')
    }
    func(na.omit(mat[,index]))
}
```

Here, we provide a new argument `func`, which defaults to taking the `mean`, but will accept any function argument. This is useful if we want to get information on the variance, skew, kurtosis, min, max, etc. of the column.

> side hint: writing plotting functions is incredibly useful, and you can enhance flexibility of your plotting functions by making many arguments (for color, point size, etc.) but having them default to your liking. For instance, I like `pch=16, las=1` for xy plots. On top of this, you can use the `...` operator so you can include an arbitrary number of other arguments to the plotting function. Note that we have not gone over visualization in `R` yet, as I wanted to first focus on An example below:

```
makePlot <- function(y,x, ...){
    plot(y,x, ...)
}
```

```
makePlot(y=runif(100), x=runif(100),
    pch=16, las=1, col='dodgerblue')
```



Structuring your repository

README file

A README file is a user-facing file which should detail what your repo is for and how to use it (e.g., reproduce an analysis from data to figures and results). It is normally a .txt or .md extension, meaning it is plain text (.txt) or markdown (.md). README files have been around for well over 50 years, and are essential in designing your workflow. The README should contain the following information:

- **Name**: the name of the project and the names of the contributors (or at least the main author)
- **Background/summary**: what is your project for. In the context of science, you would highlight your main research question
- **Prerequisites**: things necessary for the code to run effectively. Even if you were to write R code that was entirely reproducible, you are assuming that the user has R installed. Information on necessary programs, libraries, etc. should be listed
- **Installation**: what commands need to be run to reproduce the analyses? This is a description of how to understand the structure of the repo and how to actually reproduce (or install) the analyses.
- **Contributors**: Credit existing contributors and state any conflict of interests.
- **Acknowledgments**: Acknowledge financial support or help from organizations
- **Code of conduct**: A clear statement describing acceptable behavior of contributors as they interact. This is important for larger, public-facing products, as it sets up the expectations of community members in terms of standards (e.g., use of inclusive language) and things that are entirely unacceptable (e.g., sexualized language, trolling, political attacks, etc.)
- **License info**: Describe what license the code is released under. That is, you are the owner of your intellectual property, and that gives you the right to license it in different ways.

LICENSE

The license file contains the license under which you would like to develop and distribute your science under. In traditional copyright, you maintain all rights to your intellectual property (except that sweet gif you made using footage from the last LSU Tigers football game, as they retain the copyright for the original material). Copyrights are often prohibitively strict in their interpretation, and are almost entirely ignored since the development of the internet. But we still need a way to get credit for our work. *Open source licenses* allow us to do this in a flexible way. In essence, open source licenses are approved by the Open Source Initiative, and spell out the conditions for the re-use of the product. It is an important distinction, as copyright protects all rights (the default is closed), while open source licenses have things added to them to make them less open (the default is open).

There are a number of open source licenses. For the sake of this course, we will focus only on those in the Creative Commons (https://creativecommons.org/licenses/). The most liberal Creative Commons (CC for short) license is CC0, which waives all rights and places the work in the public domain (anyone can remix, reuse, profit, share, etc. your product). After this, there are variations that modify this open base. The first modification is concerning attribution.

BY: BY stands for "by attribution", meaning that you can use freely use, distribute, and profit from the product, but you have to acknowledge the source of the product.

The other licenses are variants on this license, as attribution is a fairly non-restrictive bar. The other restrictions include:

- **SA**: Standing for "share-alike", this modification means that you can build on the existing work, and use it for profit, but you will have to adopt this same license. This is often referred to as a "parasitic" license, since it sort of requires that the person using the licensed product has to license their product in a certain way (i.e., they have to have a CC license).
- **ND**: Standing for "no derivatives", this modification means that others can freely use the existing work, but cannot modify, transform, or build upon the original product. This is most useful for situations where you want to share your product, but you do not want the content of copies to change across projects. For instance, an internet security company may make a security product, but place this limitation on it so they know that others are building on their existing codebase and not modifying it or degrading it (e.g., for purposes of hacking rather than information security).
- **NC**: Standing for "non-commericial", this modification means that you cannot use the product for commercial gain. This would especially useful for photographers that produce striking images, which people may want to print and sell. This makes sure that if people print the photos, they cannot profit from it.

The end product is a basic license (either "CC0" or "CC BY") with some extra letters defining the scope of the license (e.g., "CC BY NC" stands for 'by attribution' and 'non-commercial', so you have to attribute the source of the material and cannot use it for commercial purposes). Now that you know about Creative Commons, you will likely see these everywhere.

.gitignore

The .gitignore file is a plain text file that you contains paths to files you wish to not version-control. For instance, I version control my lectures and exams for "Principles of Ecology", but I develop it in a private repo and have the answer keys to exams in my .gitignore, so that I do not version the answer key (just in case). This is also useful for dealing with large files, as Github can only accept files that are less than 100 Mb. This will not automatically be created when you initialize a git repo, so you will have to use your command line skills to create the file (i.e., `touch .gitignore` to create the file; `nano`, `vim`, `emacs`, `gedit` or the like to edit it). Files beginning with a "." are normally not even shown in file managers, but you can see them by hitting "Ctrl + h" (in Ubuntu).

Data

Data is typically a folder containing the assorted data used in the analyses. For the sake of this course, there are two types of data.

- **Raw data**: These files are never changed. Treat them like they are protected files. Scripts that process raw data into the form used in the analyses are placed in the "Analysis" folder. Raw data are not edited directly.
- **Processed data**: These files are outputs from the initial pre-processing scripts. These files are what your analyses and plots will use. Treat them like they are wrong and may change. If it is not too time-intensive to re-process your raw data, do it on occasion to ensure that nothing changes in your processed data.

Analysis

This folder contains your analytical scripts. For the sake of this class, we will assume that these are `R` or `R markdown` scripts. This code can output figures directly into this directory, or can create a "figures" directory and output there. However, it is best practice to not have your code create directories (easier to break).

It is incredibly important here to use relative paths. It is not uncommon when you see code that has calls like `read.csv('/home/user/myProject/Data/myData.csv')`. This is bad practice, because it assumes I have a home folder, a user folder, and the entire directory structure as the user did. For many, the path I used above will almost never be the case, as it is more typical in the linux OS. For Windows users, it would be something like 'C:/My Documents/Data.csv'.

> Another side 'best practice' point: file names should not have spaces. Spaces are evil, and should be avoided at all costs.

Manuscript

This folder contains all the files associated with your manuscript or project write-up (if it has one). I like to separate my manuscript writing from my analysis files, but like to have this as the same set of files. I find that when dealing with large data or complex simulation exercises, it is just not worth it to re-build the analysis each time I want to see the manuscript output. There are ways around this, but they are largely hacky.

Makefile

Makefiles are incredibly useful, and fairly simple to write. We will not go into too much detail on them, but they are essentially recipes that can be used to build your analytical pipeline and reproduce your analyses by issuing one simple command (`make`) from the command line. The makefile is named "Makefile" and is placed in the repository you wish to act on (typically) but can also exist at a higher-level.

```
TEXFILE = manuscript

paper:
    xelatex $(TEXFILE).tex
    bibtex *.aux
    xelatex $(TEXFILE).tex
    xelatex $(TEXFILE).tex
    rm -fv *.aux *.log *.toc *.blg *.bbl *.synctex.gz
    rm -fv *.out *.bcf *blx.bib *.run.xml
    rm -fv *.fdb_latexmk *.fls
```

```
view:
    evince $(TEXFILE).pdf &

clean:
    rm -fv *.aux *.log *.toc *.blg *.bbl *.synctex.gz
    rm -fv *.out *.bcf *blx.bib *.run.xml
    rm -fv *.fdb_latexmk *.fls
    rm -fv *.pdf
```

This makefile is used to compile a manuscript written in latex. The file is named 'manuscript'. The default is the first bit (`paper`). When I want to run it, I issue the command `make paper` from the terminal when my working directory is where the makefile is. This issues terminal commands that run `xelatex` and `bibtex` to compile my manuscript into pdf, and then remove `rm` a bunch of unncessary files to keep my directory tidy. If I wanted simply to clean a repo of these files without re-compiling the manuscript, I could use `make clean`.

`R` scripts can be run from the command line using the `Rscript` or `R -e` flags. This means that your makefile could be used to compile all your analyses, without the end user needing to know how to compile your code... meaning that they would not have to know `R` or `R markdown`, but they would need to have `make` installed and know how to use that. This means this is not a silver bullet for reproducibility, but it will save you some time and help keep your analytical pipeline clean and structured.

Continuous integration

Continuous integration is the practice of building and checking your project iteratively with each change made, rather than trying to ensure that everything runs only at the end of the development cycle. In the context of project development and pipelines, having a `travis.yml` file in your repo ensures that every time you make a change to your directory and push it to Github, tests on your analysis can be run and compared to expectations. We will go over this in more detail when we talk about testing in more detail. But I wanted to introduce Travis CI as a continuous integration service that is useful to developing your `analytical pipeline`. More on this later.

Other ways

There are plenty of ways to structure your analytical pipeline! I give the example I use above, but it is personal preference. One alternative is to structure your analytical pipeline as an `R` package (https://www.carlboettiger.info/2012/05/06/research-workflow.html), while others promote the use of `bookdown`, `rticles`,and other packages. This starts to get away from reproducibility concepts a bit, as there is no guarantee that these packages will be available in t years, meaning that your reproducible analytical workflow could be easily broken as a result.