# Data Wrangling IV: A philosophy of data wrangling

EDH7916 | Spring 2020

Benjamin Skinner

Now that we've learned a variety of data wrangling tasks, we'll take some time to discuss good workflow, that is, moving from raw data to data that are ready for analysis. In the lessons and assignments we've had so far, we've actually already covered many of the major points that are discussed below. This lesson, therefore, mostly just makes explicit the good practices that we've implicitly followed so far.

## Organization

From the second lesson, we've been very concerned about how our project is organized. This includes both the directory of folders and files on our machine and the scripts that run our analyses.

### Project directory

While it is certainly *easier* on the front end to store all project files in a single folder

```
./project
|
|--+ analysis.R
|--+ clean.R
|--+ clean_data.rds
|--+ data1.csv
|--+ data2.csv
|--+ histogram_x.png
|--+ density_z.png
```

it's better to separate files by type/purpose in your project directory ("a place for everything and everything in its place"):

```
./project
|
|__/data
|   |--+ clean_data.rds
|   |--+ data1.csv
|   |--+ data2.csv
|__/figures
|   |--+ histogram_x.png
|   |--+ density_z.png
|__/scripts
    |--+ analysis.R
    |--+ clean.R
```

Just like there's something to be said for visiting the library in order to scan titles in a particular section — maybe you'll find something you weren't expecting — it's useful to be able to quickly scan only the relevant files in a project. With well-named files (discussed below) and an organized directory structure, a replicator may be able to guess the purpose, flow, and output of your project without even running your code.

**Quick question** When using a project directory instead of just a single folder, we need to use file paths. In particular, we use *relative paths* rather than *fixed paths*. Why?

### Script

In the second lesson on organizing, I also shared a template R script. While you do not need to use my particular template — in fact, you should modify it to meet your needs — it does follow a few organizational rules that you should follow as well.

1. Clear header with information about the project, this file, and you.
2. Load libraries, set paths and macros, and write functions towards the top of the file.
3. Clear sections for reading in data, process, and output (ingredients, recipe, prepared dish).

**Quick question** Why put things like file paths or macros at the top of the file?

## Clarity

Remember: even though our scripts are instructions for the computer, they are also meant to be read by us humans. How well you comment your code and how well you name your objects, scripts, data, output, *etc*, determine the clarity of your intent.

### Commenting

Don't assume that your intent is clear. Particularly because so much working code comes as the result of many (…many…many…) revisions to non-working code, it's very important that you comment liberally.

- What are you doing?

```
## Creating a dummy variable for each state from current categorical variable
```

- Why are you doing it this way?

```
## Converting 1/2 indicator to 0/1 for clarity
```

- Links to supporting documents/websites

```
## see <url> for data codebook
```

- Hat-tip (h/t) for borrowed code

```
## h/t <url> for general code that I slightly modified
```

- Formula / logic behind method

```
## log(xy) = log(x) + log(y)
## using logarithms for more numerically stable calculations
```

All of these items are good uses for comments.

**Quick question** In which situation are comments not useful?

### Naming

For R, all the following objects are equivalent:

```
## v.1
x <- c("Alabama","Alaska","Arkansas")

## v.2
stuff <- c("Alabama","Alaska","Arkansas")

## v.3
cities <- c("Alabama","Alaska","Arkansas")

## v.4
states <- c("Alabama","Alaska","Arkansas")
```

However, only the last object, `states`, makes logical sense to us based on its known contents. So while R will work just as happily with `x`, `stuff`, `cities`, or `states`, a collaborator will appreciate `states` since it gives an idea of, for example, expected output — even without running the code.

> **Quick question** Without seeing the initial object assigment, what might you expect to see as output from the following code coming from a fellow higher education researcher? Why?
>
> ```
> for (i in flagship_names) {
>     print(i)
> }
> ```

When objects are well named, your code may become largely self-documenting, which is particularly nice since you don't have to worry about drift between your comments and code over time.

## Automation

In the functional programming lesson, we discussed the difference between **D**on't **R**epeat **Y**ourself programming and **W**rite **E**very **T**ime programming. As much as possible and within the dictates of organizing and clarity, it's a good idea to automate as much as you can.

### Push-button replication

If I've said it once, I've said it 1,000 times: I want an analysis script — in particular, a replication script — to run by *just pushing a button*. What I mean by this is that I want to:

1. Download the project directory
2. Open the script in R Studio
3. (If necessary, make sure I'm in the correct working directory)
4. Push the "Run" button and have all the analyses run with all estimates/figures/tables being produced and sent to their proper project folders.

The dream!

When projects are complicated or data isn't easily shared (or can't be shared), the push button replication may not be feasible. But your goal should always be to get as close to that as possible. Make your life a little harder so your collaborators, replicators — and future you! — have it a little easier.

**ONE NOTE** Like writing a paper, the process going from a blank script to a final product typically isn't linear. You'll have starts and stops, dead-ends and rewrites. You shouldn't expect a polished paper from the first draft, and you shouldn't expect to code up a fully replicable script from the get-go.

There's something to be said for just getting your ideas down on paper, no matter how messy. Similarly, it may be better to *just get your code to work first* and then clean it up later. My only warning here is that while you are forced to eventually clean up a paper (someone's going to read it), code still remains largely hidden from public view. There will be a temptation to get it to work and then move on. Fight that urge!

### Multiple scripts

We've only worked with one script so far, but you can use the `source()` function in R to call other R scripts. For a very large project, you may decide to have multiple scripts. For example, you may have one script that downloads data, one that cleans the data, one that performs the analyses, and one that makes the figures.

Rather than telling a would-be replicator to "run the following files in the following order"

1. `get_data.R`
2. `clean_data.R`
3. `do_analysis.R`
4. `make_figures.R`

you could instead include a `run_all.R` script that only sources each script in order and tell the replicator to run that:

```
## download data
source("./get_data.R")

## clean data
source("./clean_data.R")

## run analyses
source("./do_analysis.R")

## make figures
source("./make_figures.R")
```

Not only is that easier for others to run, you now have a single script that makes your analytic workflow much clearer. Note that there are other ways to do something like this, such as using makefiles (here's an example for one of my projects), but using only R will work pretty well.

## Testing

### Expected data values

### Expected output

### Warn or stop appropriately

## Data munging process

### Raw data

### Built data (objective changes)

### Clean data (subjective changes)

### Analysis data

## Recommended texts