Data Wrangling IV: A philosophy of data wrangling

EDH7916

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, is one part review and one part making explicit the good practices that we've implicitly followed.

Organization

From the organizing lesson[1], 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 — it's easier to get an idea of what's available by looking at the shelf than by scanning computer search output

[1] https:/equant.github.io/edh7916/lessons/organizing.html

— 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 lesson on organizing[2], 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 so 1 == variable name
  ```

- 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.

[2] https://equant.github.io/edh7916/lessons/organizing.html

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

Naming

For R, all the following objects are functionally 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 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 what it contains (or should contain) — even without running the code.

Quick question *Without seeing the initial object assignment, 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[3], 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[4], 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

In general, the goal for an analysis script — in particular, a replication script — is that it will run from top to bottom with no errors 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)

[3] https://en.wikipedia.org/wiki/Self-documenting_code

[4] https:/equant.github.io/edh7916/lessons/programming.html

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 have to clean up a paper eventually (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 since "no one will see it anyway, right…*right?*" Fight that urge!

Just like real debt, the technical debt[5] that you take out each time you settle for a hacky solution that's *good enough* can grow exponentially. At some point, the bill will come due in the form of obviously (or even scarier, not-so-obviously) incorrect results. Tracking down the source of errors for large projects or those that reuse code from past projects can be maddening. Take a little extra time to write good code now and it will pay dividends in saved hours down the road, not to mention fewer 2 a.m. cold-sweat awakenings, *Is that weird finding maybe because I constructed my main outcome variable wrong…wait, how* ***did*** *I code that…?*

Quick question *In a world in which not everyone knows how to use R, what's another benefit of push-button automation?*

Multiple scripts

We've only worked with one script so far, but you can use the `source()` function[6] 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:

```r
## To run all analyses, set the working directory to "scripts" and
## enter
##
## source("./run_all.R")
##
## into the console.

## download data
```

[5] https://en.wikipedia.org/wiki/Technical_debt

[6] https://stat.ethz.ch/R-manual/R-devel/library/base/html/source.html

```r
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[7], though using only R works pretty well.

Here are some replication examples for projects that I've done:

1. Skinner, B. T., & Doyle, W. R. (2021). Do civic returns to higher education differ across subpopulations? An analysis using propensity forests.[8] Journal of Education Finance, 46(4), 519–562. (Replication files)[9]
2. Skinner, B. T. (2019). Choosing College in the 2000s: An Updated Analysis Using the Conditional Logistic Choice Model.[10] Research in Higher Education, 60(2), 153–183. (Replication files)[11]
3. Skinner, B. T. (2019). Making the connection: Broadband access and online course enrollment at public open admissions institutions.[12] Research in Higher Education, 60(7), 960–999. (Replication files)[13]

You can find the rest on my website[14] as well as on my GitHub profile[15]. As you complete projects and create replication scripts, be sure to make them accessible. Not only is it good for the field, it also signals strong transparency and credibility on your part as a researcher.

Testing

As you wrangle your data, you should take advantage of your computer's power not only to perform the munging tasks you require, but also to test that inputs and outputs are as expected.

Expected data values

Let's say you are investigating how college students use their time. You have data from a time use study in which 3,000 students recorded how they spent every hour for a week. The data come to you at the student-week level and grouped by activity type. This means you see that Student A spent 15 hours working, 20 hours studying/doing homework, *etc.* You create a new column that adds the time for all categories together and take the summary. You find:

variable	mean	sd	min	max
total_hrs	167.99	1.93	152.48	183.46

[7] https://en.wikipedia.org/wiki/Makefile
[8] https://www.btskinner.io/repository/skinner2021civicii.pdf
[9] https://github.com/btskinner/civic_returns_pf_rep
[10] https://www.btskinner.io/repository/skinner2018clogit.pdf
[11] https://github.com/btskinner/colchoice_rep
[12] https://www.btskinner.io/repository/skinner2019oa_online_broadband.pdf
[13] https://github.com/btskinner/oa_online_broadband_rep
[14] https://www.btskinner.io/publications/
[15] https://github.com/btskinner

Something is wrong.

Quick question *What's wrong?*

Rather than going through an ad-hoc set of "ocular" checks, you can build in more formal checks. We've done a little of this when using `identical()` or `all.equal()` to compare data frames. You could build these into your scripts. For example:

- summing activity hours in a week, check if any exceed the total number of hours that are possible
- reading in data with student GPAs, check if they are negative
- combining data from 100 students at 5 schools, make sure your final data frame has 500 rows
- creating a new percentage from two values, check that the value between 0 and 100 (or between 0 and 1 if a proportion)

You can use simple tests to check, and include a `stop()` command inside an `if()` statement:

```r
## halt if max number of months is greater than 12 since data are only
## for one year
if (max(length(months)) > 12) {
    stop("Reported months greater than 12! Check!")
}
```

This only works if you know your data. Knowing your data requires both domain knowledge ("Why are summer enrollments so much higher than fall enrollments…that doesn't make sense.") and particular knowledge about your data ("All values of x between 50 and 100, except these two, which are -1 and 20000…something's up").

Expected output from functions

Programmers are big on unit testing[16] their functions. This simply means giving their functions or applications very simple input that *should* correspond to straightforward output and confirming that that is the case.

For example, if their function could have three types of output depending on the input, they run at least three tests, one for each type of output they expect to see. In practice, they might run many more tests for all kinds of conditions — even those that should "break" the code to make sure the proper errors or warnings are returned.

As data analysts, you are unlikely to need to conduct a full testing regime for each project. But if you write your own functions, you should definitely stress test them a bit. This testing shouldn't be in your primary script but rather in another script, perhaps called `tests.R`. These can be informal tests such as

```r
## should return TRUE
(function_to_return_1() == 1)
```

or more sophisticated tests using R's testthat[17] library.

Again, the point isn't that you spend all your time writing unit tests. But spending just a bit of time checking that your functions do what you think they do will go a long way.

[16] https://en.wikipedia.org/wiki/Unit_testing

[17] https://testthat.r-lib.org

Imagine you've written your own version of `scale()`, *which lets you normalize a set of numbers to N(0,1):*

```
my_scale <- function(x) {
    ## get mean
    mean_x <- mean(x)
    ## set sd
    sd_x <- sd(x)
    ## return scaled values
    ##             x - mean_x
    ## scaled_x = ------------
    ##                sd_x
    return((x - mean_x) / sd_x)
}
```

What kind of tests do you think you should run, i.e., *what kind of inputs should you give it and what should you expect?*

Fail loudly and awkwardly

There is a notion in software development that a program should fail gracefully[18], which means that when a problem occurs, the program should at the very least give the user a helpful error message rather than just stopping without note or looping forever until the machine halts and catches fire[19].

As data analysts, I want your scripts to do the opposite: I want your code to exit loudly and awkwardly if there's a problem. You aren't writing a program but rather an analysis. Therefore, the UX[20] I care about is the reader of your final results and your future replicators. Rather than moving forward silently when something strange happens[21] — which is lovely on the front end because everything keeps running but deadly on the back end — your code should stop and yell at you when something unexpected happens. It's annoying, but that's the point.

Four stages of data: a taxonomy

In the lessons so far, we've generally read in raw data, done some processing that we printed to the console, and then finished. This is fine for our small lessons and assignments, but in your own work, you will often want to save your changes to the data along the way — that is, save updated data files at discrete points in your data analysis. This is particularly true for large projects in which wrangling your data is distinct from its analysis and you don't want to run each part every time.

Broadly, there are four stages of data:

1. Raw data
2. Built data (objective changes)
3. Clean data (subjective changes)
4. Analysis data

[18] https://en.wikipedia.org/wiki/Graceful_exit
[19] https://en.wikipedia.org/wiki/Halt_and_Catch_Fire_(computing)
[20] https://en.wikipedia.org/wiki/User_experience
[21] https://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-1044-7

For small projects, you may not end up with data files for each stage. That said, your workflow should be organized in terms of moving from one stage to the next.

We'll talk about each below.

ACKNOWLEDGMENT I want to give a quick hat tip to two people who've really helped me clarify my thinking about these stages of data, Will Doyle[22] and Sally Hudson[23]: Will for first showing the importance of keeping raw data untouched; and Sally for showing how useful it is to make a distinction between objective and subjective data processing, particularly in large projects. Thanks and credit to both.

Raw data

Raw data, that is, the data you scrape, download, receive from an advisor or research partner, *etc*, are sacrosanct. Do not change them! They are to be read into R (or whatever software you use) and that is all.

You will be tempted from time to time to *just fix this one thing* that is wonky about the data. Maybe you have a `*.csv` file that has a bunch of junk notes written above the actual data table (not uncommon). The temptation will be to delete those rows so you have a clean file. NO! There are ways to skip rows in a file like this, for example. R is particularly good at reading in a large number of data types with a large number of special situations.

If you are at all concerned that you or a collaborator may accidentally save over your raw files, your operating system will allow you to change the file permissions such that they are **read-only files** (Dropbox will let you protect files in this way, too). Particularly if working with a number of collaborators, protecting your raw data files in this way is a good idea.

Built data (objective changes)

As a first step in processing your data, you should only make *objective modifications*. What do I mean by objective? They are those modifications that:

1. Are required to get your data in good working order
2. Are modifications that will always be needed across projects using the same data (in the context of multiple projects)

To the first point, let's say you receive student-level administrative data that includes a column, `state`, that is the student's home state. However, values in the column include:

- `"Kentucky"`
- `"KY"`
- `"Ken"`
- `"21"` (state FIPS code)

Another example would be a date column, `month`, in which you see:

- `"February"`
- `"Feb"`
- `"2"`

[22] https://twitter.com/wdoyle42
[23] https://twitter.com/SallyLHudson

In both cases, it's clear that you want a consistent format for the data. Whatever you pick, these are objective changes that you would make at this time.

To the second point, let's say this same administrative data are going to be used across multiple projects. In each project, you need to compute the proportion of in-state vs. out-of-state students. The data column that indicates this is called `residency` and is coded (with no data labels) as

- `1` := in-state
- `2` := out-of-state

Coming back to the data later and taking a look at the first few rows in `residency`, will it be clear

- what `residency` means?
- what the values of `1` and `2` represent?

No.

An objective change that you can use across analyses will be to make a new column called `instate` in which

- `0` := out-of-state
- `1` := in-state

The benefits are three-fold:

- `0` and `1` are logically coded as `FALSE` and `TRUE`
- `instate` gives a direction to the `0`s and `1`s: `1` means in-state
- taking the mean of `instate` will give you the proportion of in-state students in your data

If your data or project (or number of projects) is large, you many want to save data files after your objective wrangling. These can be reused across projects, which will make sure that objective data wrangling decisions are consistent.

Clean data (subjective changes)

Whatever changes are left over after you've make objective changes are *subjective* changes. These changes are project and analysis specific.

For example, let's say your student-level administrative data have information about the student's expected family contribution or EFC[24]. In your data, you have the actual values. However, you want to compare your data to other published research in which EFC is binned into discrete categories. This means that in order to make a comparison, you have to bin your data the same way. This is a subjective task for two reasons:

- there's not an *objective* reason to bin your data in this manner, *i.e.* to make the data consistent or clearer
- it's unlikely you'll need this new column of data for other projects

Some other subjective changes you might make to a data set:

[24] https://fafsa.ed.gov/help/fftoc01g.htm

- create an indicator variable that equals `1` when a student's family income is below $35,000/year (why $30k? why not $25k or $35k?)
- convert institutional enrollments to the log scale (do you always need to do this?)
- join in data on the unemployment rate in the student's county (do you always need this information at the county level? at all?)

But wait! Maybe you *do* want to use the binned value of EFC in multiple projects — perhaps the binned categories are standard across the literature — or the other examples appear always useful and otherwise standard. Isn't that *objective*?

Maybe!

The thing about determining the difference between *objective* vs *subjective* cleaning is that on the margins, well, it's subjective! The point is not to have hard and fast rules, but rather to do your best to clearly separate those data wrangling tasks that must be done for the sake of consistency and clarity versus those that may change depending on your research question(s).

> **Quick question** *What might be some other* subjective *data wrangling tasks? Think about other research you've conducted or seen. Or think about what our lessons and assignments so far: what have been subjective tasks?*

Analysis data

Finally, analysis data are data that are properly structured for the analysis you want to run (or the table or figure you want to make). In many cases, your analysis data set will be the same as your clean data set:

- it's in the shape you want (rectangular, with observations in rows and variables in columns)
- it has all the variables you need to compute summary statistics or run regression models

Most R statistical routines (*e.g.*, `mean()`, `sd()`, `lm()`) do a fair amount of data wrangling under the hood to make sure it can do what you ask. For example, you may have missing values (`NA`) in your clean data set, but R's linear model function, `lm()`, will just drop observations with missing values by default. If you are okay with that in your analysis, R can handle it.

However, for some special models or figures, you may need to reshape or pre-process your clean data in one last step before running the specific analysis you want. For example, your clean data frame may have a column of parental occupational field (`parocc`) as a categorical variable in which the first few categories are

- `11` := Management Occupations
- `13` := Business and Financial Operations Occupations
- `15` := Computer and Mathematical Occupations
- `17` := Architecture and Engineering Occupations

…and so on. You want to run an analysis in which you can't use this categorical variable but instead need a vector of dummy variables, that is, converting this single column into a bunch of columns that only take on `0/1` values if `parocc` equals that occupation:

- `manage_occ == 1` if `parocc == 11` else `manage_occ == 0`
- `busfin_occ == 1` if `parocc == 13` else `busfin_occ == 0`
- `commat_occ == 1` if `parocc == 15` else `commat_occ == 0`
- `arceng_occ == 1` if `parocc == 17` else `arceng_occ == 0`

…and so on. Particularly if

- your data are large
- the analysis takes a while to run
- you want to use the same data set up for multiple (but not all) models or analytic tasks

you may want to save this particular analysis data set.

As with the difference between *objective* and *subjective* data, the difference between *subjective* and *analysis* data does not align to a set of hard and fast rules. That's okay! The main point is that as a data analyst, you make data wrangling decisions thinking about how you will go from *raw* to *objective* to *subjective* to *analysis* data in a logical and clear manner. Collaborators, replicators, and future you will appreciate it!

Recommended texts

There are a number of great texts that talk about the process of data wrangling. Two that have particularly informed this lecture and are worth looking at more fully:

- The Pragmatic Programmer: From Journeyman to Master[25] by Andrew Hunt and David Thomas
- Code and Data for the Social Sciences: A Practitioner's Guide[26] by Matthew Gentzkow and Jesse Shapiro

[25] https://pragprog.com/book/tpp20/the-pragmatic-programmer-20th-anniversary-edition

[26] https://web.stanford.edu/~gentzkow/research/CodeAndData.pdf