Writing Good Code

A Bag of Tips

Barum Park (b.park@cornell.edu)
Cornell University

CPC Proseminar, 2025

Bad News. Unfortunately...I realize that most of you probably use `Stata`

But most of my work uses other languages ...

So, today I'll be talking about **general tips** rather than any concrete code

But(!) at least for me, collection of tips were very helpful

And I wished someone had told me about them ...

What is a good code?

What is a good code?

I wish I knew... But we might agree on some criteria!

A good code should be ..

1. Working **as intended** / Replicable
2. Readable / Transparent
3. Efficient / Fast / Scale up to “large” data

For most social scientists, **1.** to **3.** would be roughly the order of importance (I believe...)

So, let us discuss them in order (as long as time allows) ...

Working (as intended)

Working (as intended)

A good thing to remember in coding is that

If your code throws an error at you, it's a good sign

You know that something is off, so you can fix it!

Far more worse are the errors that you don't even know about...

Working (as intended)

Here's a mistake that people often make:

```
use "data.dta", clear   // load data with vars x and y
gen z = (x > 10)        // create dummy var z
reg y z, robust         // run regression!
```

Problem here is: **`z`** will be `0` for rows in which **`x`** is missing

So, we are **imputing** (arbitrarily) the value `0` for cases, the true value of which might be `1`.

This is not `Stata`'s fault...**It's your fault**

You should be understanding the **default behavior** of `Stata`'s functions / commands (or any other programming language that you are using)

Working (as intended)

Many similar coding errors happen all the time ...

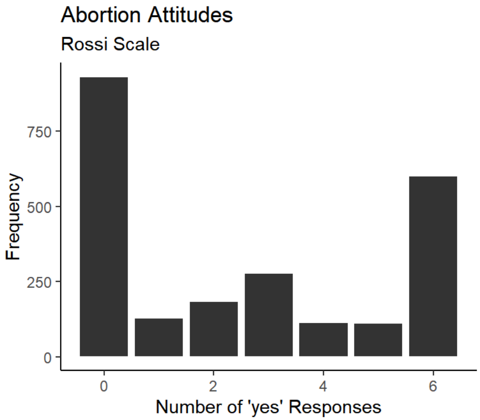
For example, when creating the abortion (Rossi) scale using the 2018 GSS, we might use the following R code :

```
# vector of abortion item names
items_names = c("abdefect", "abnomore", "abhlth",
                "abpoor", "abrape", "absingle")

# for each resp., count no. of "yes" responses
rossi = rowSums(gss[, abort_items] == "yes", na.rm = T)
```

Working (as intended)

When we plot the `rossi` variable, we get



which shows how polarized the US is on this issue. But is it really that polarized?

Working (as intended)

When "just" counting the number of "yes" responses, *all missing responses will be treated as "no" responses*

In the 2018 GSS, *a lot* of respondents weren't asked any of the abortion questions at all

Their response pattern would look like

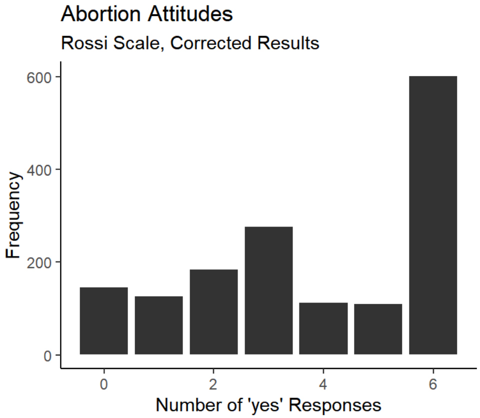
```
NA, NA, ..., NA
```

which contains zero `"yes"` responses.

For all these respondents, `rossi` would be equal to zero (meaning "abortion shouldn't be legal under any circumstances")

Working (as intended)

When dropping respondents whose ballot did not contain the abortion questions, we get



which tells a very different story

Working (as intended)

But how can we make sure that we don't make these mistakes?

There are basically three ways:

1. We give up on coding ...
2. We build in **checks** into our code
3. We hire RAs who build **checks** into their code

Testing

You might think “Of course, I’m checking my missing values. Why do I need a separate code for this?”

- your code will change multiple times over a project
- you’d need to check for errors manually **every time** something changes
- So, why not automatize it?

Here is a real-world example

Modularizing

A good way to reduce errors in your code and to make your code more manageable is to **modularize** it.

If your whole analysis is done in one `.do`, `.R`, or `.py` file, you're probably doing something wrong.

You should have at the very least two files

1. One code that takes the rawdata and creates datasets that on which your models are run
2. One code that runs the analysis on the created data and creates some output

This is the *minimum*...

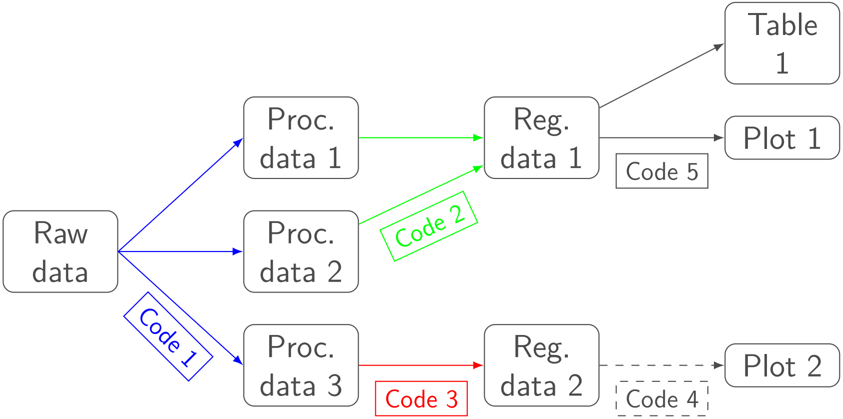
By the way, **never overwrite your raw data**

Modularizing

But why does this help with avoiding errors?

- ▷ You can build checks/tests for each code
- ▷ Whenever your data/code changes, you only have to check downstream codes
- ▷ Reviewing long code is tiring and error prone ...

Example Workflow



Modularizing

Once you have one script for each module, you can write one "main script" that will call each of the scripts in order to execute the whole workflow

But there are better alternatives

- ▷ **`snakemake`** (Python, R, Stata, bash, etc.)
- ▷ **`targets`** (R)

In any case, you should have one "main" code that

1. calls all other codes in your workflow; and
2. reproduces all the results of your analysis (inc. numbers mentioned in your text)

Here is a real-world example

Modularizing

Another way to modularize your code:

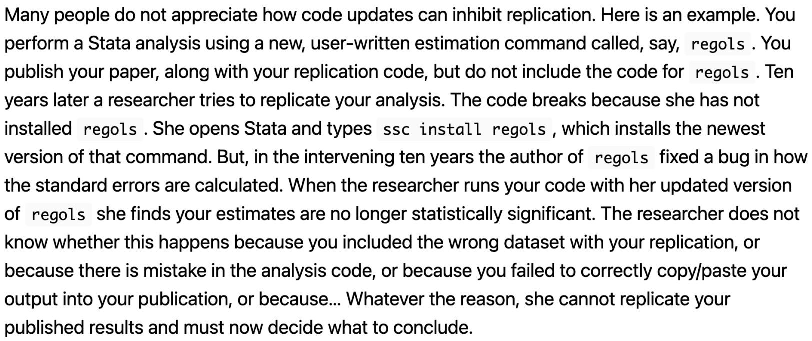
write your own functions/packages for repeated tasks !

- ▷ Cut-and-pasting is quite an error-prone task
- ▷ Once you tested your function thoroughly, you won't have to worry about it
- ▷ Portability (You can use them in other projects as well!)

Example

Package Versions

To make your analysis replicable by others, it is important to keep information about the *version of your packages/software*



Many people do not appreciate how code updates can inhibit replication. Here is an example. You perform a Stata analysis using a new, user-written estimation command called, say, `regols`. You publish your paper, along with your replication code, but do not include the code for `regols`. Ten years later a researcher tries to replicate your analysis. The code breaks because she has not installed `regols`. She opens Stata and types `ssc install regols`, which installs the newest version of that command. But, in the intervening ten years the author of `regols` fixed a bug in how the standard errors are calculated. When the researcher runs your code with her updated version of `regols` she finds your estimates are no longer statistically significant. The researcher does not know whether this happens because you included the wrong dataset with your replication, or because there is mistake in the analysis code, or because you failed to correctly copy/paste your output into your publication, or because... Whatever the reason, she cannot replicate your published results and must now decide what to conclude.

Source: https://julianreif.com/guide/#libraries

Package Versions

The lazy approach: Simply let others know what version each package you have used.

In `R`, printing out the information provided by `SessionInfo()` does this job (I am not sure about other languages)

Example

Better approach: Provide others with an replica of the environment in which your code was run (virtual environments, Docker, etc.)

If nothing works, you can always save the required packages/add-ons into the project directory and ship it with your replication package

Example

Using Other People's Code

Most statistical software allow you to import user-written functions (or commands) in the form of packages or macros.

How do you know which one to trust?

1. Check the user base (how many people are using it?)
2. Browse the source code (esp. testing)
3. Check whether the package is in active development

Examples of :

Package no longer in development
Package in active development

Readability / Transparency

Readability / Transparency

Modularization helps a lot with readability, already.

So, let's talk about how to make your code readable *within* each specific code file.

Readability / Transparency

But first, why is readability important?

1. You'll make less errors
2. You won't be able to collaborate with others on a code if your code is not readable (nor will others be able to review your code)
3. You often want to reuse your old code; so you need to understand what you did when you come back after years...
4. You want to make your code public (always!) and help others to use it and/or learn from it

Comments

The first rule is to **write A LOT of comments**
Tell readers (including yourself) what you are doing

This become *extremely important* when you collaborate with others on the same code

Without comments, your code is often unintelligible to others (including your future self)

Here is a bad example

Coding Style

In general, it is a good idea to have a *reasonable coding style* and stick to it (i.e,. **be consistent**)

What is "reasonable"?

1. it should be easily readable
2. it should not be too far from the convention in your field

As with other points about readability, this becomes *important* when you collaborate with others

So, it is often a good idea to have a "style sheet" to which all collaborators stick to

Here is a real-world example

Version Control

Use a *version control system* (e.g., `Git`)

If your code has names such as

```
regV1.R
regV2.R
   ⋮
regFinal.R
regFinalV1.R
   ⋮
regFinalFinalV1.R
```

you are not taking advantage of civilization

Version Control

What you want is

- work on only *one file*
- while saving the *entire history* of the file
- and being able to *revert* your file to any point in that history

This is what version control systems help you to do (and there are *a lot* of other benefits as well)

Here is a real-world example

Efficiency

Efficiency

By “efficiency” we mean how fast your code gives you the desired results

Efficiency is probably the least important aspect of coding for most of you

If your data set is small (e.g., $N < 1,000,000$ and $K < 1,000$), it really doesn’t matter “how” you compute your results, as long as you get the right answer

If you work with large(r) data, an efficient code can save you days, weeks, or even months

Benchmarking

I think there are no "general" rules to make your code efficient. Things will depend on the language you are using (e.g., loops are efficient in `C/C++` but slow in `R` or `Python`)

But, here are some tips:

1. Unless you know what you are doing, use the functions that are provided by default or by well-known packages (and try hard to find them before you write your own functions!)

2. If you are not sure what function to use, *benchmark your functions* (example)

Scaling Up

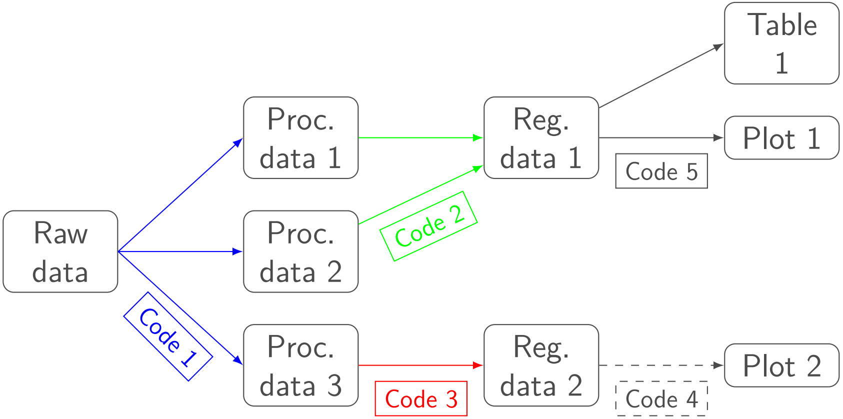
To scale up your code to large(r) datasets

1. Parallelize independent parts of your

2. If you have a modularized workflow, think about whether some codes can be run independently
   (programs such as `snakemake` will do this for you automatically and can be used with `R, python, Stata, etc.`)

3. Then, run your code on a HPC cluster.

Example Workflow (Again)



HPC

If you plan to work on complicated projects, it pays off to learn how to interact with high-performance computing clusters (HPCs).

You don't want to face a situation like this:

When it's time to go home but your code isn't done running. #RStats

when there are other ways around ...

HPC

To use these clusters:

▷ you'll have to become familiar with the *command line*, since most of them run on Linux

▷ Using version control systems (Git/Github) makes things much easier as well

It *really pays off to become familiar with these tools*

Thank you!

Barum Park
b.park@cornell.edu