Best Practices for
Computer Programming in Economics

Tal Gross

January 19, 2016

All empirical work requires computer code. And yet, economics departments almost never teach basic programming skills. Good coding in economics means first and foremost that the programmer avoid errors. There are several simple habits that can help. Here they are.

Rule 1 Use Sensible Names

A novice labor economist will create `wage1` and `wage2` when cleaning labor data. Perhaps `wage1` is the nominal wage and `wage2` is the real wage. Or perhaps `wage1` is measured in the first interview of a survey, and `wage2` is measured in the second interview. Such a naming convention—especially in the absence of documentation—creates a problem. If the economist were to hand the code over to someone else, or to come back to it after a lengthy journal-submission process, the variable names may lead to confusion.

The sensible thing to do would be to name the variables appropriately: `nominal_wage` and `real_wage` or `wage_interview1` and `wage_interview2`.

Another example comes from difference-in-difference papers. It is often necessary to create a variable that indicates an observation occurs after the policy change. Typically, economists call this the "`post`" variable. But a more effective name would be `post1999` or `post_min_wage_change`. The more concrete the name of the variable, the less likely you are to get confused later on.

Often, you see programmers create a `sex` variable, which is 1 for males and 2 for females. (Why do men get to be number one?) But, regardless, it is not obvious to someone just viewing the data for the first time how such a variable is coded. Instead, programmers ought to create a `female` variable, which is 1 for females and 0 for males.

Furthermore, take care in naming Stata programs. I've seen other economists create directories that are filled with programs like: `clean.do`, `main.do`, `regs.do`, and `regs_revised.do`. How do you know which program does what?

Instead, use long, descriptive names, that involve verbs and nouns. For instance, a Stata program called `clean-raw-census-data.do` is a program that is named sensibly. You already know what it does, even before looking at the code. Calling the program `cleaner.do` or, god forbid, `census.do` may lead to confusion.

In general, label variables, programs, and data sets with long, descriptive names. To save space, you can cut out the vowels. For instance, `run-regressions.do` can become `run-regressions-cntrlfrrace.do` once the new version of the program controls for race.

There's obviously a trade-off here between clarity and convenience. It does take time to type out long names, and it can be tedious. But, in research, clarity matters more than convenience. The goal is to avoid coding errors, and spending some extra time typing out long names is a burden worth enduring in the service of that goal.

Rule 2 Comment Everything

Code in economics should be written to be read by a bitter, angry graduate student who has been assigned to replicate your work. Whenever a program does anything at all out of the ordinary, one should explain what it is doing. For instance the code below makes clear exactly what it is going to do, and then does it.

```
** I know the year and quarter that the interview took place,
** along with the year and month that the respondent was born.
** So I can calculate year in quarters.
gen dob_q =0
if ‘year’ == 1998 | ‘year’ == 1999  | ‘year’ == 2000  | ‘year’ == 2001  \\\
| ‘year’ == 2002  | ‘year’ == 2003  | ‘year’ == 2004  | ‘year’ == 2005  \\\
| ‘year’ == 2006  | ‘year’ == 2007 {
        rename dob_m dob_m_orig
        gen dob_m = real(dob_m_orig)

        rename dob_y_p dob_y_p_orig
        gen dob_y_p = real(dob_y_p_orig)
}
```

Rule 3 Make Code Readable

Which of the following two lines of code are easier to read? This one:

```
gen seven_month_income=wage1+wage2+wage3+wage5+wage6+wage7
```

Or this one:

```
gen seven_month_income = wage1 + wage2 + wage3 + wage4 + wage5 + wage6 + wage7
```

The second line of code is much easier to read, simply because I added spaces in between each operator. (Did you catch the error in the first line?) After all, regular prose would bemuchmoredifficulttoreadwithout spaces.

In the same way, programmers can indent code to make it more readable. This can be especially useful for loops:

```
forvalues i = 1(1)5 {
        sum var‘i’
        gen log_var‘i’ = log(var‘i’)
}
```

Rule 4 Create Sections

Novels are composed of sentences, paragraphs, and chapters. Such divisions help the reader digest the work. Computer code should be similarly broken up into sections. The best tool for doing so is the “ascii box.” The box looks like so:

```
************************************************************
**
************************************************************
```

and can divide a program into sections. Well-labeled sections make the program much easier to understand.

As an example, see the code below, which is part of a larger program that cleans data from the National Health Interview Survey. Notice that each section is labeled, indicating the section’s purpose.

```
************************************************************
**  Bring in Alternate Schooling Variables                **
************************************************************

** The NHIS asks all respondents who are >= 18 what they were
** doing last week (DOINGLW) and then asks why they were not
** working (WHYNOWRK)

if ‘year’ == 2001 | ‘year’ == 2002 | ‘year’ == 2003 {
        rename whynowk1 whynowrk
}
if ‘year’ == 2004 | ‘year’ == 2005 | ‘year’ == 2006 | ‘year’ == 2007 {
        rename whynowkp whynowrk
}

disp "Now getting the alternate schooling variable for year ‘year’"
tab whynowrk, miss
gen byte in_school_lastwk = (whynowrk == 2)
sum in_school_lastwk

************************************************************
** Clean Weights                                          **
************************************************************

sum wtfa
rename wtfa sample_weight
egen sample_weight_sum = sum(sample_weight)
list sample_weight_sum in 1/1

************************************************************
** Bring in interview quarter, just 2004                  **
************************************************************

** In 2004 alone, intv_qrt existed only in the household file
if ‘year’==2004 {
        preserve
                clear
                use ../src/‘year’/househld.dta
                keep hhx intv_qrt assignwk
                sort hhx
                tempfile household_file_intv_‘year’
                save "‘household_file_intv_‘year’’"
        restore

        sort hhx
        merge hhx using "‘household_file_intv_‘year’’" , uniqusing nokeep
        tab _merge
        assert _merge==3
        drop _merge

}
```

The section headings make it easier to digest the program. If you're focused on the interview-quarter issue, the section headings help you find the part you care about quickly.

Rule 5 Make Code Portable

Imagine this line of code in a Stata program:

```
merge state year using ’C:\\My Documents\Get Tenure\Project Dir\x.dta’
```

What happens when this researcher switches from a "C" drive to a "D" drive? Or—god forbid—gives this Stata program to a Linux user?

There is an easier way. When creating a project directory, I construct the following folders: `do`, `log`, `dta`, `src`, and `gph`. Then I put all Stata programs in the `do` directory, all original data in the `src` directory, all the new data sets created in the `dta` directory, and all figures in the `gph` directory. When I run programs, I manually change Stata's working directory to the `do` directory, and change the merge statement above to:

```
merge state year using ../dta/x.dta
```

Notice that this new command only refers to two levels of directories. Thus the Stata code can now be moved from one file system to another without changing *any* code. The code is even portable across operating systems. It works equally well on a Unix server as on a Windows desktop machine.

Rule 6 Check Your Work

Programmers make mistakes when they do not understand how a variable is coded. For instance, a programmer might think that a variable is binary when it is actually categorical, or a programmer might think that a variable is always positive when it is often negative. One way to prevent such errors is to constantly summarize the variables that one creates.

Here's one example:

```
gen byte female = (sex < 1)
sum female
```

Here, I created a female indicator, but I summarized the variable immediately afterward. This way, I can then read the log file to ensure that roughly half of the sample is female. This may seem like overkill, but the extra summary command barely slows the program down. And every once in a while, this habit will help you catch errors.

Rule 7 Use a Template

An entire research project in economics will involve numerous computer programs. It is easier to interpret the research project if each program is formatted in a similar manner, and if each program states its purpose very clearly. A template for programs can help. The code sample below is one such template I use for all of my Stata programs.

```
#delim cr
set more off
pause on

capture log close
set linesize 200
set logtype text
log using ../log/XXX.log , replace

/* -------------------------------------

AUTHOR:

PURPOSE:

DATE CREATED:

NOTES:

--------------------------------------- */

clear

** Code begins here...

log close
exit
```

Rule 8 Preserve Source Data

In the end, you should be able to re-create a project from scratch. To do so, you need to isolate "source" data from data sets that are created. Never ever modify source data.

Rule 9 Don't Repeat Yourself

Professional programmers tell each other "don't repeat yourself" so often that they abbreviate the advice: DRY. Here's the idea behind DRY.

Suppose that you have a large project involving survey data. Certain commands might appear in multiple programs. For instance, you might have to inflation-adjust wages at several different points in the process.

Suppose that inflation adjusting wages looks something like this:

```
replace wage = 1.8 * wage if year == 1998
replace wage = 1.6 * wage if year == 1997
...
```

What happens if you discover a mistake in these lines of code? You have to remember to correct the mistake in all of the programs in which the commands appear. And what if you introduce subtle differences in the commands across programs? How do you harmonize the commands, when it is not clear which version is correct?

Instead, just create a short program called `inflation-adjust-wages.do`, and then have other programs call that program. That way, these commands only appear once and you wont be repeating yourself.