Coding Style and Good Computing Practices

Jonathan Nagler

January, 1995

This article appeared in *The Political Methodologist*, Spring, 1995 - Volume 6, No 2.

It was reprinted in *PS: Political Science & Politics*, September, 1995, Volume 28, No 3., Pgs 488-492.

I thank Charles Franklin, Bob Hanneman, Gary King, and Burt Kritzer for useful comments and suggestions.

Recently *The Political Methodologist* presented the case for replication. Replication depends upon individual researchers being able to explain exactly what they have done. And being able to explain exactly what one has done requires keeping good records of it. This article describes basic good computing practices. It contains advice for writing clear code, as well as advice on general computing practices to maintain. The goals are simple. First, the researcher should be able to replicate his or her own work six hours later, six months later, and even six years later. Second, others should be able to look at the code and understand what was being done (and preferably why it was being done). In addition, following good computing practices encourages the researcher to maintain a thorough grasp of what is being done with the data, and thus makes it easier to perform additional analyses. Good coding allows for more efficient research. One is not always re-reading one's own work and retracing one's own steps to perform the smallest bit of additional analysis.

The sequence this article is written in leads the reader from more general themes to more specific ones. I encourage readers who don't have the patience for the big picture to skip ahead rather than skip the whole article. Even learning basic conventions about variable names will put you ahead of most coders! And this article is *not* meant only for sophisticated statistical researchers. In fact the statistical procedures you will ultimately use have absolutely nothing to do with the topic of this article. These practices should be used even if you are doing nothing more complex than producing 2x2 tables with a particular data-set.

First, what do I mean by computing practices and `code'? Computing practices covers everything you do from the time you open the codebook to a dataset, or begin to enter your own data, to the time you produce the actual numbers that will be placed in a table of an article. By code I mean the actual computer syntax -- -- or computer program -- -- used to perform the computations. This most likely means the set of commands issued to a higher-level statistics package such as SAS or SPSS. I will refer to a given file of commands as a `program.' Most political scientists do not think of themselves as computer programmers. But when you write a line of syntax in SAS or SPSS, that is exactly what you are doing -- -- programming a computer. It is coincidental that the language you use is SAS instead of Fortran or C. The paradox is that most political scientists are not trained as computer programmers. And so they never learned basic elements of programming style, nor are they practiced in the art of writing clean, readable, maintainable code. The classic text on programming style remains Kernighan and Plauger *The Elements of Programing Style* (1974), and most books on programming include a chapter on style. I recommend including a section on coding style in every graduate methods sequence.

This article starts from the point when a raw data set exists somewhere on a computer. It breaks analysis down into two basic parts: coding the data to put it into a useable form, and computing with the data. The first part can be broken down into two component steps: reading the essential data from a larger data-set; and recoding it and computing new variables to be used in data analysis.

Here are the basic points that will be covered below, and at the end of the article I list a set of `rules' laid out.

- Labbooks: Essential.
- Command Files: they should be kept.
- Data-Manipulation vs. Data-Analysis: these should be in distinct files.
- Keep Tasks Compartmentalized (`modularity').
- Know what the code is supposed to do *before* you start.
- Don't be Too Clever.
- Variable Names should *mean* something.
- Use parentheses and white-space to make code readable.
- Documentation: All code should include comments meaningful to others.

Labbooks

Our peers in the laboratory sciences maintain lab-books. These books indicate what they did to perform their experiments. We are not generally performing experiments. But we have identical goals. We want to be able to retrieve a record of every step we have followed to produce a particular result. This means the lab-book should include the names of every file of commands you wrote, with a brief synopsis of what the file was intended to do. The labbook should indicate which produced results worth noting. For instance, the following labbook entry describes what the file of Gauss code `mnp152.g' did:

```
Date: Oct 11, 1994
File: mnp152.g
Author: JN
Purpose: Analysis - this file does multinomial probits on our basic model.
Results: The results were used in Table 1 of the Midwest paper.
Machine: Run on billandal (IBM/RS6000).
```

It is a good idea to have a template that you follow for each Labbook entry, this encourages you to avoid getting careless in your entires. The above template includes: Date, File, Author, Purpose, Results, and Machine. You might have a set of `Purposes' -- -- Re-Coding, Data-Extraction, Data-Analysis -- -- that you feel each file fits into. It may seem superfluous to indicate what machine the file was executed on. But should you develop the habit of computing on several machines, or should you move from one machine to another in the course of a project, this information becomes invaluable in making sure you can locate all off your files.

It makes a lot of sense to have the labbook online. It can be in either a Wordperfect file, or a plain ascii text file, or whatever you are most comfortable writing in. First, it is easy to search for particular events if you remember names of them. Second, it can be accessed by more than one researcher if you are doing joint-work.

> **Rule:** Maintain a labbook from the beginning of a project to the end.

Labbooks should provide an `audit-trail' of any of your results. The labbook should contain all the information necessary to take a given piece of data-analysis, and trace back where all of the data came-from - including its original source and all recode steps. So while there are many different sorts of entries you might want to keep in a labbook, and many different styles to keep them; the central point to keep in mind is that whatever style you choose meets this purpose.

Command Files

The notion of `command files' may appear as an anachronism to some. After all, can't we all just work interactively by pointing and clicking to do our analysis - not having to go through the tedium of typing separate lines for each regression we want to run and submitting our job for analysis? Yes, we can. But, we probably don't want to. And even if we do, modern software will keep a log for us of what we have done (a fact that seems to escape many users of the software). The reason I am not a fan of interactive work has to do with the nature of the analysis we do. The model we ultimately settle on was usually arrived at after several estimates testing many models. And this is appropriate, we ought to all be testing the robustness of our results to changes in model specification, changes in variable measurement, etc.. By writing command files to perform estimates, and keeping each version, one has a record of all of this. [One can go beyond individual command files and create batch-files to run the whole sequence of command files necessary to read the data, process it, and do analysis. There is alot to be said for this technique; but it is a bit beyond the scope of this article.]

You generally want a large set of comments at the beginning of each file indicating what the file is intended to do. And each file should not do too much. Comments on the top of a file should list the following:

- State the date it was written, and by whom.
- Include a description of what the file does.
- Note what file it was immediately derived from.
- Note all changes from its predecessors where appropriate.
- Indicate any datasets the file utilizes as input.
- Indicate any output files or dataset created.

If the file utilizes a data-set; there should be a comment indicating the source of the data-set. This could be either another file you have that produced the data; or it could be a description of a public-use data set. If it is a public-use dataset, include the date and version number. For example, the first nine lines of a command file might be:

```
    \*     File-Name:   mnp152.g
           Date:        Feb 2, 1994
           Author:      JN
           Purpose:     This file does multinomial probits on our basic model.
           Data Used:   nes921.dat (created by mkasc1c.cmd)
           Output File: mnp152.out
           Data Output: None
           Machine:     billandal (IBM/RS6000)
    *\
```

You could keep a template with the fields above left blank, and read in the template to start each new command file. You should treat the comments at the top of a file the way you would the notes on a table; they should allow the file to stand alone and be interpreted by anyone opening the file without access to other files.

Know the Goal of the Code

It makes no sense to start coding variables if you don't know what the point is. You end up with a bunch of variables that are all being recoded later on and confusing you to no end. Before you start manipulating the data, figure out what you will be testing, and how the variables need to be set up. An example: say you want to test the claim that people who voted in 1992, but did not vote in 1988 were more likely to support Perot than Bush in 1992. How do you code the independent variable indicated here? Well, you really want a `new voter' variable, since your substantive hypothesis is stated most directly in terms of `new voters.' Thus the variable should be coded so that:

```
vote in 1992, not voted in 1988 = 1 \\
voted in 1992, voted in 1988 = 0
```

> **Rule:** Code each variable so that it corresponds as closely as possible to a verbal description of the substantive hypothesis the variable will be used to test.

Fixing Mistakes

If you find errors, the errors should be corrected where they first appear in your code - not patched over later. This may mean re-running many files. But this is preferable to the alternative. If you patch the error further `downstream' in the code then you will need to remember to repeat the patch should you make any change in the early part of the data-preparation (i.e., you decide you need to pull additional variables from a data-set, etc). If the patch is downstream you are also likely to get confused as to which of your analysis runs are based on legitimate (patched) data, and which are based on incorrect data.

> **Rule:** Errors in code should be corrected where they occur and the code re-run.

Data-Manipulation Versus Data-Analysis

Most data goes through many manipulations from `raw-form' to the form we use in our analyses. It makes sense to isolate as much of this as possible in a separate file. For instance, say you are using the 1992 NES presidential election file. You have 100 variables in mind that you *might* use in your analysis. It makes sense to have a program that will pull those 100 variables from the NES dataset, give them the names you want, do any basic recoding that you know you will maintain for all of your analysis, and create a `system file' of these 100 named and recoded variable that can be read by your statistics package. There are at least two reasons for this. First, it saves a lot of time. You are going to estimate at least 50 models before settling one one. Do you really want to read the whole NES dataset off the disk 50 times when you could read a file 1/20th the size instead? Second, why do all that re-coding and naming 50 times? You might accidentally alter the recodes in one of your files.

> **Rule:** Separate tasks related to data-manipulation vs data-analysis into separate files.

Modularity

Separating data-manipulation and data-analysis is an example of modularity. Modularity refers to the concept that tasks should be split up. If two tasks can be performed sequentially, rather than two-at-a-time, then perform them sequentially. The logic for this is simple. Lots of things can go wrong. You want to be able to isolate what went wrong. You also want to be able to isolate what went right. After what specific change did things improve? Also, this makes for much more readable code.

Thus if you will be engaging in producing some tables before multivariate analysis, you might have a series of programs: descrip1.cmd, descript2.cmd, ..., descrip9.cmd. Following this, you might produce: reg1.cmd, reg2.cmd,..., reg99.cmd. You need not constrain yourself to one regression per file. But the regressions in each file should constitute a coherent set. For instance, one file might contain your three most likely models of vote-choice - each dissaggregated by sex. This does tend to lead to proliferation of files. One can start with reg1.cmd and finish with reg243.cmd. But disk-space is cheap these days; and the files can easily be compressed and stored on floppies if things are getting tight.

> **Rule:** Each program should perform only one task.

KISS

Keep it simple and don't get too clever. You may think of a very clever way to code something this week. Unfortunately you may not be as clever next week and you might not be able to figure out what you did. Also, the next person to read your code might not be so clever. Finally, you might not be as clever as you think - the clever way you think of to do 3 steps at once might only work for 5 out of 6 possible cases you are implementing it on. Or it might create nonsense values out of what should be missing data. Why take the chance? Computers are very fast. Any gains you make thru efficiency will be dwarfed by the confusion caused later on in trying to figure out what exactly your code is doing so efficiently.

> **Rule:** Do not try to be as clever as possible when coding. Try to write code that is as simple as possible.

Variable Names

There is basically no place for variables named **X1** other than in simulated data. Our data is real, it should have names that impart as much meaning as we can. Unfortunately many statistical packages still limit us to 8-character names (and for portability's sake, we are forced to stick with 8 character names even in packages that don't impose the limit). However, your keyboard has 84 keys, and the alphabet has 52 letters: 26 lower-case and 26 upper-case. Indulge yourself and make liberal use of them. There are also several additional useful characters -- -- such as the underscore and the digits 0-9 -- -- at your disposal. It is a general convention in programming to use UPPER CASE characters to indicate constants, and lower case characters to indicate variables. This might not be as useful in statistical programming. You might adopt the convention that capitals refer to computed quantities (such as **PROBCHC1** : the estimated probability of choosing choice 1). And if you are trying to have your code closely follow the notation of a particular econometrics article, you might use a capital U for utility, or a capital V for the systemic component of utility. Obviously in such a case comments would be in order! Some people like variable names such as **NatlEcR** because the use of capitals allows for clearly indicating where one word stops and another starts. **NatlEcR** makes it easier to think of `National Economic - Retrospective' than **natlecr** might. You will need to make some tradeoffs in the conventions you choose. The important thing is to adopt a convention on the use of capitals and stick with it.

Rule: Use a consistent style regarding lower and upper case letters.

Rule: Use variable names that have substantive meaning.

When possible a variable name should reveal subject and direction. The simplest case is probably a dummy variable for a respondent's gender; imagine it is coded so that 0 = men, 1 = women. We could either call the variable `SEX', or `WOMEN.' It is pretty clear that `WOMEN' is the better name since it indicates the direction of the variable. When we see our coefficients in the output we won't have to guess whether we coded men=1 or women=1.

Rule: Use variable names that indicate direction where possible.

Similarly, value labels are very useful things for packages that permit them. The examples of computer syntax I use in this article are written in SST (Dubin/Rivers 1992), but they can be easily translated into SAS, SPSS, or most statistical packages. Here is a simple example. The variable {\bf natlecr} indicates the respondent's retrospective view of performance of the national economy. Notice that the variable name can only indicate so much information in 8 characters. But the label of it and the values tell us what we need. And the fact that the label tells us where to look the variable up in the code-book is further protection.

```
label   var[natlecr]     lab[v3531:national economy - retro]         \
                         val[1 gotbet 3 same 5 gotworse]
```

Some people using NES data -- -- or any data produced by someone else accompanied by a codebook -- -- follow the convention of naming the variable by its codebook number (i.e., V3531), and using labels for substantive meaning. I think this is a poor practice. Consider which of the following statements is easier to read:

```
logit      dep[preschc]    ind[one educ women partyid]
```

or:

```
logit      dep[V5609]      ind[one V3908 V4201 V3634]
```

The codebook name for the variable should *definitely* be retained; but it can be retained in the label statement. Without the codebook name one would not know which of the several party-identification variables the variable **partyid** refers to.

Writing Cleanly

Anything that reduces ambiguity is good. Parentheses do so and so parentheses are good. A reader should not need to remember the precedence of operators. But in most cases parentheses are more valuable as visual cues to grouping expressions than to actual precedence of operators. Almost as useful as parentheses is white-space.

Proper spacing in your code can make it much easier to read. This means both vertical space between sections of a program that do different tasks, and indenting to make things easier to follow.

Not knowing the syntax of SST, it might be completely opaque that the following code saves all observations of the variables **preschc ... deficit** for which all of the variables contain a valid response. However, the spacing can be a big help towards figuring this out.

```
rem    *******************************************************************
rem    *******************************************************************

recode  var[preschc educ east south west                                \
            women respfinp natlec resplib bclibdis gblibdis rplibdis    \
            dem rep respgjob resphlth respblk respab                    \
            age1829 age3044 age4559 newvoter termlim deficit] map[md=-9]


write   var[preschc educ east south west                                \
            women respfinp natlec bclibdis gblibdis rplibdis            \
            dem rep respgjob resphlth respblk respab                    \
            termlim age1829 age3044 age4559 newvoter deficit]           \
            file[nes9212r.asc]                                          \
        if[(preschc!=-9)&(educ!=-9)&(east!=-9)&(south!=-9)&(west!=-9)&  \
            (women!=-9)&(respfinp!=-9)&(natlec!=-9)&(bclibdis!=-9)&     \
            (gblibdis!=-9)&(rplibdis!=-9)&                              \
            (dem!=-9)&(rep!=-9)&(respgjob!=-9)&(resphlth!=-9)&          \
            (respblk!=-9)&(respab!=-9)&(termlim!=-9)&                   \
            (age1829!=-9)&(age3044!=-9)&(age4559!=-9)&(newvoter!=-9)&   \
            (deficit!=-9)]

rem    *******************************************************************
rem    *******************************************************************
```

> **Rule:** Use appropriate white-space in your programs, and do so in a consistent fashion to make them easy to read.

Kernighan and Plauger (1972) suggest the telephone test, perhaps a bit anachronistic since you will more likely email the code than read it to someone, but useful nonetheless. Read your code to someone over the phone. If they can't understand it, try writing the code again.

Comments

There is probably *nothing* more important than having adequate comments in your code. Basically one should have comments before any distinct task that the code is about to perform. Beyond this, one can have a comment to describe what a single line does. The basic rule of thumb is this: is the line of code absolutely, positively self-explanatory *to someone other than yourself* without the comment? If there is any ambiguity, go ahead and put the comment in. Notice that in the second sample of code below, the section titled `Creating Liberal-Distance Variables' does exactly that, it creates a set of variables measuring the ideological distance between respondents and candidates. This would not be so clear from the 8 character variable names. Also notice that the code is a distinct part of the total program and it is easy to see where it begins and ends.

Remember though, the comments should *add* to the clarity of the code. Don't put a comment before each line repeating the content of the line. Put comments in before specific blocks of code. Only add a comment for a line where the individual line might not be clear. And remember, if the individual line is not clear without a comment - maybe you should rewrite it.

> **Rule:** Include comments before each block of code describing the purpose of the code.

Rule: Include comments for any line of code if the meaning of the line will not be unambiguous to someone other than yourself.

Rule: Rewrite any code that is not clear.

Following is a case where a single comment lets us know what is going on:

```
rem    *******************************************************************
rem     Create party-id dummy variables
rem    *******************************************************************

rem     Missing values are handled correctly here by SST.
rem     In other statistics packages these three variables might
rem     have to be initialized as missing first.

set     dem = (pid < 3)
set     ind = (pid == 3)
set     rep = (pid > 3)

rem    *******************************************************************
rem    *******************************************************************
```

Now here is a case where a longer comment is essential:

```
rem    *******************************************************************
rem     Create ideological-distance variables
rem    *******************************************************************

rem     Ideological distance is computed as the distance between the
rem     respondent, and the mean placement of the candidate by all
rem     respondents used in our multivariate analysis who could place
rem     the candidate.

rem     The mean-values used here are generated by making a first pass
rem     at the data set for the subsample we will use here. See the code
rem     at the end of this file for that pass.

rem     This hard-wiring of numbers is poor style; but it is very
rem     difficult to automate this given the way in which SST handles
rem     missing data.

set     gblibdis = (resplib - 5.32)^2
set     bclibdis = (resplib - 2.98)^2
set     rplibdis = (resplib - 4.49)^2

rem    *******************************************************************
rem    *******************************************************************
```

Most programmers think that well-written code should be self-documenting. This is partly true. But no matter how well-written your code is some comments can make it much clearer.

Recodes and Creating New Variables

Probably the most important thing to keep track of both when recoding variables and creating new variables is missing data. There is no general rule that can specify exactly how to do this, because treatment of missing data can vary across statistics packages. Thus the best rule is:

Rule: Verify that missing data is handled correctly on any recode or creation of a new variable.

In some statistics packages you may be best served by initializing all new variables as missing data, and allowing them to become legitimate values only when they are assigned a legitimate value. The best advice is to recode and create new variables defensively.

> **Rule:** After creating each new variable or recoding any variable, produce frequencies or descriptive statistics of the new variable and examine them to be sure that you achieved what you intended.

Generally it is poor style to `hard-wire' values into your code. Any specific values are likely to change when some related piece of code somewhere else is altered or when the data-set changes. The example of ideological-distance variables in the Comment section above is an example where values *are* hard-wired into the code. This is a case where a choice had to be made. The values 5.32, 2.98, and 4.49 represent means of candidate placement by a selected set of respondents. Computing this with the **means** command over the appropriate set of respondents, and doing the assignment was complicated enough that rather than write code that could not be simple - it was decided to hard-wire the values. If the following code produced the desired results (it does not), it would be far preferable:

```
rem     ******************************************************************
rem      Create ideological-distance variables
rem     ******************************************************************


set      gblibdis = (resplib - mean(bushlib))^2
set      bclibdis = (resplib - mean(clinlib))^2
set      rplibdis = (resplib - mean(perolib))^2

rem     ******************************************************************
rem     ******************************************************************
```

> **Rule:** When possible, automate things and avoid placing hard-wired values (those computed `by-hand') in code.

Finally, after you have done recodes and created new variables it is a good idea to list all of the variables. This way you can confirm that you and your statistics package agree on what data is available, and how many observations are available for each variable. In SST this would be done with a list command, in SAS, PROC CONTENTS will produce a clean list of variables. Most statistics packages offer similar commands.

Procedures or Macros

Most political scientists don't ever have to write a `macro' or `procedure', but maybe that's why they do so little secondary analysis once they generate some estimates. The purpose of a well-defined procedure is to automate a particular sequence of steps. They are very useful both in making your code more readable, and in allowing you to perform the same operation multiple times on different values or on different variables. The use of Procedures and Macros is really a topic for a separate article; but political scientists should realize that they are available in most statistics packages.

Summing Up

The rules presented here are merely *one way* of accomplishing the goal you should have in mind. That goal is to write clear code that will function reliably and that can be read and understood by you and others and can serve as a road-map for replicating and extending your research.

Most people are in a huge hurry when they write their code. They are either excited about getting the results and want them as fast as possible, or they figure the code will be run once and then thrown out. **If your program is not worth documenting, it probably isn't worth running.** The time you save by writing clean code and commenting it carefully may be your own.

Rules

1. Maintain a labbook from the beginning of a project to the end.
2. Code each variable so that it corresponds as closely as possible to a verbal description of the substantive hypothesis the variable will be used to test.
3. Errors in code should be corrected where they occur and the code re-run.
4. Separate tasks related to data-manipulation vs data-analysis into separate files.
5. Each program should perform only one task.
6. Do not try to be as clever as possible when coding. Try to write code that is as simple as possible.
7. Each section of a program should perform only one task.
8. Use a consistent style regarding lower and upper case letters.
9. Use variable names that have substantive meaning.
10. Use variable names that indicate direction where possible.
11. Use appropriate white-space in your programs, and do so in a consistent fashion to make them easy to read.
12. Include comments before each block of code describing the purpose of the code.
13. Include comments for any line of code if the meaning of the line will not be unambiguous to someone other than yourself.
14. Rewrite any code that is not clear.
15. Verify that missing data is handled correctly on any recode or creation of a new variable.
16. After creating each new variable or recoding any variable, produce frequencies or descriptive statistics of the new variable and examine them to be sure that you achieved what you intended.
17. When possible, automate things and avoid placing hard-wired values (those computed `by-hand') in code.

Return to: Jonathan Nagler's Home Page

Please send suggestions and comments to: jonathan.nagler@nyu.edu .