The Stata Journal

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press, and Stata is a registered trademark of StataCorp LP.

The Stata Journal (2005)
5, Number 4, pp. 560–566

Suggestions on Stata programming style

Nicholas J. Cox
Durham University, UK
n.j.cox@durham.ac.uk

Abstract. Various suggestions are made on Stata programming style, under the headings of presentation, helpful Stata features, respect for datasets, speed and efficiency, reminders, and style in the large.

Keywords: pr0018, Stata language, programming style

1 Introduction

Programming in Stata, like programming in any other computer language, is partly a matter of syntax, as Stata has rules that must be obeyed. It is also partly a matter of style. Good style includes, but is not limited to, writing programs that are, above all else, clear. They are clear to the programmer, who may revisit them repeatedly, and they are clear to other programmers, who may wish to understand them, to debug them, to extend them, to speed them up, to imitate them, or to borrow from them.

People who program a great deal know this: setting rules for yourself and then obeying them ultimately yields better programs and saves time. They also know that programmers may differ in style and even argue passionately about many matters of style, both large and small. In this morass of varying standards and tastes, I suggest one overriding rule: Set and obey programming style rules for yourself. Moreover, obey each of the rules I suggest unless you can make a case that your own rule is as good or better.

Enough pious generalities: The devil in programming is in the details. Many of these details reflect longstanding and general advice (e.g., Kernighan and Plauger 1978).

2 Presentation

In this section, I give a list of basic guidelines for programs.

1. Always include a comment containing the version number of your program, your name or initials, and the date the program was last modified above the `program` line, for example,

```
*! 1.0.0 Tom Swift 21jan2006
program myprog
```

   (As said, this line is indeed just a comment line; it bears no relation to the Stata `version` command. However, `which myprog` will echo this comment line back

to you whenever this `myprog` is visible to Stata along your ado-path. Both this comment and a `version` command should be used.)

2. Use sensible, intelligible names where possible for programs, variables, and macros.

3. Choose a name for your program that does not conflict with anything already existing. Suppose that you are contemplating *newname*. If typing either `which` *newname* or `which` *newname*`.class` gives you a result, StataCorp is already using the name. Similarly, if `ssc type` *newname*`.ado` gives you a result, a program with your name is already on SSC. No result from either does not guarantee that the program is not in use elsewhere: `findit` *newname* may find such a program, although often it will also find much that is irrelevant to this point.

4. Brevity of names is also a virtue. However, no platform on which Stata is currently supported requires an 8-character limit. Tastes are in consequence slowly shifting: an intelligible long name for something used only occasionally would usually be considered preferable to something more cryptic.

5. Note that actual English words for program names are supposedly reserved for StataCorp.

6. Use the same names and abbreviations for command options that are in common use in official Stata's commands. Try to adopt the same conventions for options' syntax; for example, allow a *numlist* where similar commands use a *numlist*. Implement sensible defaults wherever possible.

7. Use natural names for logical constants or variables. Thus `local OK` should be 1 if true and 0 if false, permitting idioms such as `if 'OK'`. (But beware such logicals' taking on missing values.)

8. Type expressions so they are readable. Some possible rules are as follows:

   a. Put spaces around each binary operator except `^` (`gen z = x + y` is clear, but `x ^ 2` looks odder than `x^2`).

   b. `*` and `/` allow different choices. `num / den` is arguably clearer than `num/den`, but readers might well prefer `2/3` to `2 / 3`. Overall readability is paramount; compare, for example,

      ```
      hours + minutes / 60 + seconds / 3600
      ```

      with

      ```
      hours + minutes/60 + seconds/3600
      ```

   c. Put a space after each comma in a function, etc.

   d. Use parentheses for readability.

   Note, however, that such a spaced-out style may make it difficult to fit expressions on one line, another desideratum.

9. Adopt a consistent style for flow control. Stata has `if`, `while`, `foreach`, and `forvalues` structures that resemble those in many mainstream programming languages. Programmers in those languages often argue passionately about the best layout. Choose one such layout for yourself. Here is one set of rules:

   a. Tab lines consistently after `if` or `else` or `while` or `foreach` or `forvalues` (the StataCorp convention is that a tab is 8 spaces and is greatly preferable if Stata is to show your programs properly).

   b. Do not put anything on a line after a brace—an opening `{` or a closing `}` (comments are a possible exception).

   c. Put a space before braces.

   d. Align the `i` of `if` and the `e` of `else`, and align closing braces `}` with the `i`, or the `e`, or the `w` of `while`, or the `f` of `foreach` or `forvalues`:

```
if ... {
        ...
        ...
}
else {
        ...
        ...
}
while ... {
        ...
        ...
}
foreach ... {
        ...
        ...
}
```

   In Stata 8 and later, putting the opening and closing braces on lines above and below the body of each construct is compulsory (with the exceptions that the whole of an `if` construct or the whole of an `else` construct may legally be placed on one line). For earlier releases, it is strongly advised.

10. Write within 80 columns (72 are even better). The awkwardness of viewing (and understanding) long lines outweighs the awkwardness of splitting commands into two or more physical lines.

11. Use `#delimit ;` sparingly (Stata is not C): commenting out ends of lines is tidier where possible (although admittedly still ugly). The `///` comment introduced in Stata 8 is most helpful here and is arguably more pleasing visually than `/* */`.

12. Use blank lines to separate distinct blocks of code.

13. Consider putting `quietly` on a block of statements rather than on each or many of them. An alternative in some cases is to use `capture`, which eats what output there might have been and any errors that might occur, which is sometimes the ideal combination.

14. You can express logical negation by either `!` or `~`. Choose one and stick with it. StataCorp has changed recently from preferring `~` to preferring `!`.

15. Group `tempname`, `tempvar`, and `tempfile` declarations.

16. Well-written programs do not need many comments. (Comment: I could certainly argue about that!)

17. Use appropriate `display` styles for messages and other output. All error messages (and no others) should be displayed as `err`; that is, type `di as err`. In addition, attach a return code to each error message; `198` (syntax error) will often be fine.

3 Helpful Stata features

Stata has a number of features that makes programming easier. Examples of ways a programmer can use these features are as follows:

1. Stata is very tolerant through version control of out-of-date features, but that does not mean that you should be. To maximize effectiveness and impact and to minimize problems, write programs using the latest version of Stata and exploit its features.

2. Make yourself familiar with all the details of `syntax`. It can stop you from reinventing little wheels. Use wildcards for options to pass to other commands when appropriate.

3. Support `if` *exp* and `in` *range* where applicable. This is best done using `marksample` `touse` (or occasionally `mark` and `markout`). Have `touse` as a temporary variable if and only if `marksample` or a related command is used. See `help marksample`.

4. `_result()` still works, but it is unnecessarily obscure compared with `r()`, `e()`, or `s()` class results.

5. Make effective use of information available in `e()` and `r()`. If your program is to run in a context that implies that results or estimates are available (say, after `regress`), make use of the stored information from the prior command.

6. Where appropriate, ensure that your command returns the information that it computes and displays so that another user may employ it `quietly` and retrieve that information.

7. Ensure that programs that focus on time series or panel data work with time-series operators if at all possible. In short, exploit `tsset`.

8. Define constants to machine precision. Thus use `_pi` or `c(pi)` rather than some approximation, such as 3.14159, or use `-digamma(1)` for the Euler–Mascheroni constant γ rather than 0.57721. Cruder approximations may give results adequate for your purposes, but that does not mean that you should eschew wired-in features.

9. Familiarize yourself with the built-in material revealed by `creturn list`. Scrolling right to the end will show several features that may be useful to you.

10. SMCL is the standard way to format Stata output.

4 Respect for datasets

In general, make no change to the data unless that is the direct purpose of your program or that is explicitly requested by the user.

1. Your program should not destroy the data in memory unless that is essential for what it does.

2. You should not create new permanent variables on the side unless notified or requested.

3. Do not use variables, matrices, scalars, or global macros whose names might already be in use. There is absolutely no need to guess at names that are unlikely to occur, as temporary names can always be used (type `help macro` for details on `tempvar`, `tempname`, and `tempfile`).

4. Do not change the variable type unless requested.

5. Do not change the sort order of data; use `sortpreserve`.

5 Speed and efficiency

Here is a list of basic ways to increase speed and efficiency:

1. Test for fatal conditions as early as possible. Do no unnecessary work before checking that a vital condition has been satisfied.

2. Use `summarize, meanonly` for speed when its returned results are sufficient. Also consider whether a `quietly count` fits the purpose better.

3. `foreach` and `forvalues` are cleaner and faster than most `while` loops and much faster than the old `for` that still satisfies some devotees. Within programs, avoid `for` like the plague. (Note to new Mata users: this does not refer to Mata's `for`.)

4. `macro shift` can be very slow when many variables are present. With 10,000 variables, for example, working all the way through a variable list with `macro shift` would require around 50 million internal macro renames. Using `foreach` or `while` without a macro shift is faster.

5. Avoid `egen` within programs; it is usually slower than a direct attack.

6. Try to avoid looping over observations, which is very slow. Fortunately, it can usually be avoided.

7. Avoid `preserve` if possible. `preserve` is attractive to the programmer but can be expensive in time for the user with large data files. Programmers should learn to master `marksample`.

8. Specify the type of temporary variables to minimize memory overhead. If a `byte` variable can be used, specify `generate byte 'myvar'` rather than letting the default type be used, which would waste storage space.

9. Temporary variables will be automatically dropped at the end of a program, but also consider dropping them when they are no longer needed to minimize memory overhead and to reduce the chances of your program stopping because there is no room to add more variables.

10. Avoid using a variable to hold a constant; a macro or a scalar is usually all that is needed. One clear exception is that some graphical effects depend on a variable being used to store a constant.

6 Reminders

In this section, I describe a few general procedures that will improve one's code:

1. Remember to think about string variables as well as numeric variables. Does the task carried out by your program make sense for string variables? If so, will it work properly? If not, do you need to trap input of a string variable as an error, say, through `syntax`?

2. Remember to think about making your program support `by` *varlist*`:` when this is natural. See `help byable`.

3. Remember to think about weights and implement them when appropriate.

4. The job is not finished until the `.hlp` is done. Use SMCL to set up your help files. Old-style help files, while supported, are not documented, while help files not written in SMCL cannot take advantage of its paragraph mode, which allows lines to autowrap to fit the desired screen width. For an introduction to the SMCL required to write a basic help file, see [U] **18.11.6 Writing online help** or `help examplehelpfile`.

7 Style in the large

Style in the large is difficult to prescribe, but here are some vague generalities:

1. Before writing a program, check that it has not been written already! `findit` is the broadest search tool.

2. The best programs do just one thing well. There are exceptions, but what to a programmer is a Swiss army knife with a multitude of useful tools may look to many users like a confusingly complicated command.

3. Get a simple version working first before you start coding the all-singing, all-dancing version that you most desire.

4. Very large programs become increasingly difficult to understand, build, and maintain, roughly as some power of their length. Consider breaking such programs into subroutines or using a structure of command and subcommands.

5. The more general code is often both shorter and more robust. Sometimes programmers write to solve the most awkward case, say, to automate a series of commands that would be too tedious or error-prone to enter interactively. Stepping back from the most awkward case to the more general one is often then easier than might be thought.

6. Do not be afraid to realize that at some point you may be best advised to throw it all away and start again from scratch.

8 Note: Use the best tools

Find and use a text editor that you like and that supports programming directly. A good editor, for example, will be smart about indenting and will allow you to search for matching braces. Some editors even show syntax highlighting. For much more detailed comments on various text editors for Stata users, see
http://fmwww.bc.edu/repec/bocode/t/textEditors.html.

9 Acknowledgments

Many thanks to Kit Baum, Bill Gould, Alan Riley, and Vince Wiggins for general benedictions and numerous specific contributions.

10 References

Kernighan, B. W. and P. J. Plauger. 1978. *The Elements of Programming Style*. New York: McGraw–Hill.

About the Author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also co-authored fifteen commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is an Editor of the *Stata Journal*.