Stata Coding Practices: Programming (Ado-files)

Jump to: navigation, search

Programs and ado-files are the main methods by which Stata code is condensed and generalized. By writing versions of code that apply to arbitrary inputs and saving that code in a separate file, the application of the code is cleaner in the main do-file and it becomes easier to re-use the same analytical process on other datasets in the future. Stata has special commands that enable this functionality. All commands on SSC are written as ado-files by other programmers; it is also possible to embed programs in ordinary do-files to save space and improve organization of code.

Read First

This article will refer somewhat interchangeably to the concepts of "programming", "ado-files", and "user-written commands". This is in contrast to ordinary programming of do-files. The article does not assume that you are actually writing an ado-file (as opposed to a `program` definition in an ordinary dofile); and it does not assume you are writing a command for distribution. That said, Stata programming functionality is achieved using several core features:

- The `program` command sets up the code environment for writing a program into memory.
- The `syntax` command parses inputs into a program as macros that can be used within the scope of that program execution.
- The `tempvar`, `tempfile`, and `tempname` commands all create objects that can be used within the scope of program execution to avoid any conflict with arbitrary data structures.

The `program` command

The `program` command defines the scope of a Stata program inside a do-file or ado-file. When a `program` command block is executed, Stata stores (until the end of the session) the sequence of commands written inside the block and assigns them to the command name used in the `program` command. Using `program drop` before the block will ensure that the command space is available. For example, we might write the following program in an ordinary do-file:

```
cap prog drop autoreg
prog def autoreg

  reg price mpg i.foreign

end
```

After executing this command block (note that `end` tells Stata where to stop reading), we could run:

```
sysuse auto.dta , clear
autoreg
```

If we did this, Stata would output:

```
. autoreg

      Source |       SS           df       MS      Number of obs   =        74
-------------+----------------------------------   F(2, 71)        =     14.07
       Model |   180261702         2  90130850.8   Prob > F        =    0.0000
    Residual |   454803695        71  6405685.84   R-squared       =    0.2838
-------------+----------------------------------   Adj R-squared   =    0.2637
       Total |   635065396        73  8699525.97   Root MSE        =    2530.9

------------------------------------------------------------------------------
       price |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         mpg |  -294.1955   55.69172    -5.28   0.000    -405.2417   -183.1494
             |
     foreign |
    Foreign  |   1767.292    700.158     2.52   0.014     371.2169    3163.368
       _cons |   11905.42   1158.634    10.28   0.000     9595.164    14215.67
------------------------------------------------------------------------------
```

All this is to say is that Stata has taken the command **reg** `price mpg i.foreign` and will execute it whenever `autoreg` is run as if it were an ordinary command.

As a first extension, we might try writing a command that is not dependent on the data, such as one that would list all the values of each variable for us. Such a program might look like the following:

```
cap prog drop levelslist
prog def levelslist

  foreach var of varlist * {
    qui levelsof `var' , local(levels)
    di "Levels of `var': `: var label `var''"
    foreach word in `levels' {
      di "  `word'"
    }
  }

end
```

We could then run:

```
sysuse auto.dta , clear
levelslist
```

Similarly, we could use any other dataset in place of `auto.dta` . This means we would now have a useful piece of code that we could execute with any dataset open, without re-writing what is a mildly complex loop each time. When we want to save such a snippet, we usually write an ado-file: we name the file `levelslist.ado` and we add a starbang line and some comments with some metadata about the code. The full file would look something like this:

```
*! Version 0.1 published 24 November 2020
*! by Benjamin Daniels bbdaniels@gmail.com

// A program to print all levels of variables
cap prog drop levelslist
prog def levelslist

  // Loop over variables
  foreach var of varlist * {

    // Get levels and display name and label of variable
    qui levelsof `var' , local(levels)
    di "Levels of `var': `: var label `var''"

    // Print the value of each level for the current variable
    foreach word in `levels' {
      di "  `word'"
    }

  }

end
```

The file would then just need to be run using `run levelslist.ado` in the runfile for the reproducibility package to ensure that the command `levelslist` would be available to all do-files in that package (since programs have a global scope in Stata). However, this command is not very useful at this stage: it outputs far too much useless information, particularly when variables take integer or continuous values with many levels. The next section will introduce code that allows such commands to be customizable within each context you want to use them.

The `syntax` command

The `syntax` command takes a program block and allows its inputs to be customized based on the context it is being executed in. The `syntax` command enables all the main features of Stata that appear in ordinary commands, including input lists (such as variable lists or file names), `if` and `in` restrictions, `using` targets, `=` applications, weights, and options (after the option comma in the command).

The help file for the `syntax` command is extensive and allows lots of automated checks and advanced features, particularly for modern features like factor variables and time series (`fv` and `ts`). For advanced applications, always consult the `syntax` help file to see how to accomplish your objective. For now, we will take a simple tour of how `syntax` creates an adaptive command.

First, let's add simple syntax allowing the user to select the variables and observations they want to include. We might write:

```
cap prog drop levelslist
prog def levelslist

syntax anything [if]
preserve

  // Implement [if]
  marksample touse
  qui keep if `touse' == 1

  // Main program loops
  foreach var of varlist `anything' {
    qui levelsof `var' , local(levels)
    di " "
    di "Levels of `var': `: var label `var''"
    foreach word in `levels' {
      di "  `word'"
    }
  }

end
```

There are several key features to note here. First, we write `anything` in the **`syntax`** command to allow the user to write absolutely anything they like as the arguments to be passed into the program. By default, this is assigned to the string local `` `anything' `` and can be recovered throughout the program. Recall that local macros in Stata have strictly local scope; in this case, that means locals from the calling do-file will not be passed into the program, and locals from the program will not be passed back into the calling do-file.

Second, we write `[if]` in brackets to declare that the user can optionally declare an if-restriction to the command. This does nothing on its own: it simply creates another local string macro called `` `if' `` containing the restriction. However, Stata provides the implementation shortcut **`marksample`** to implement this restriction. By calling **`marksample`** `touse` , Stata creates a temporary variable `` `touse' `` for every observation indicating whether it satisfies the if-restriction or not.

Then, the if-restriction must be applied: we can **`preserve`** the data and then **`drop`** the ineligible observations before running more code. This is an appropriate choice here for several reasons: **`preserve`** will always restore the data to the original state at the end of program execution, no matter what happens later in the program, due to its scope; **`restore`** is not even needed here. For this reason, we will often only use **`preserve`** in this context in programming, and prefer other methods for loading and re-loading data inside the program block.

Now, we can run commands like:

```
sysuse auto.dta , clear
levelslist foreign
levelslist foreign make if foreign == 1

sysuse census.dta
levelslist region
levelslist state if region == 1
```

Other `syntax` elements work similarly, although they are not parsed through `marksample` (except `in`). The `using` syntax is typically used to target a file on the operating system; when you want to import or export data this is the feature of choice, and you should always test and implement it with compound double quotes (for example, `` `" `using' "' ``) and determine whether or not you want to pass `using` itself into the `` `using' `` macro by writing `[using/]` instead. See the helpfile for details.

Finally, the options syntax allows optional triggers to be implemented. Let's allow the user to request value labels, by writing:

```
cap prog drop levelslist
prog def levelslist

syntax anything [if] , [VALuelabels]
preserve

  // Implement [if]
  marksample touse
  qui keep if `touse' == 1

  // Main program loops
  foreach var of varlist `anything' {
    qui levelsof `var' , local(levels)
    di " "
    di "Levels of `var': `: var label `var''"
    foreach word in `levels' {
      // Implement value label option if specified
      if "`valuelabels'" != "" {
        local thisLabel : label (`var') `word'
        local thisLabel = ": `thisLabel'"
      }
      di "  `word'`thisLabel'"
    }
  }

end
```

When the `valuelabels` option is specified (using either `, val` as an allowed abbreviation by the capitalization or writing out its full name), the `` `valuelabels' `` macro will contain `"valuelabels"`. Otherwise it will be empty. Therefore simple conditionals allow options to be checked and executed. Now we could run:

```
sysuse census.dta
levelslist region , val
```

and we would get:

```
Levels of region: Census region
  1: NE
  2: N Cntrl
  3: South
  4: West
```

However, we can see that the command would then fail if we ran `levelslist region state , val`, because `state` is a string variable and cannot have labels. So we might want to allow the user to specify a list of variables to show labels for, as the following:

```
cap prog drop levelslist
prog def levelslist

syntax anything [if] , [VALuelabels(string asis)]
preserve

  // Implement [if]
  marksample touse
  qui keep if `touse' == 1

  // Main program loops
  foreach var of varlist `anything' {
    qui levelsof `var' , local(levels)
    di " "
    di "Levels of `var': `: var label `var''"
    foreach word in `levels' {
      // Implement valuelabels option
      local thisLabel ""
      if strpos(" `valuelabels' "," `var' ") >= 1 {
        local thisLabel : label (`var') `word'
        local thisLabel = ": `thisLabel'"
      }

      // Display value (and label if requested)
      di "  `word'`thisLabel'"
    }
  }

end
```

Because we now allow the option as `[VALuelabels(string asis)]`, it will either contain the string written into the option or it will contain nothing. We need to rewrite the implementation slightly. First, we need to reset `` `thisLabel' `` so it is emptied whenever it does not apply. Second, we need to use a tool like `strpos()` to check if a variable occurs in the list - when we write the helpfile, we will make clear that this option needs to take a list of variables. Is is possible to require this through the options syntax itself but it can introduce issues (if, for example, the command first loads data, a **`varlist`** check might fail on the data currently in memory). In this kind of operation, it is doubly clear that the full names of variables need to be used (to avoid needing to pull in commands like **`unab`**). Also, note the use of extra spacing around both arguments of `strpos()`; these ensures that variables whose name are a substring of another do not trigger the option. Now, we can run `levelslist region state , val(region)` and get the results we wanted.

The `temp` commands

Stata has a set of `temp` commands that can be used to store information temporarily. This functionality

Categories (/Special:Categories):

- Coding Practices (/Category:Coding_Practices)
- Stata Coding Practices (/Category:Stata_Coding_Practices)