Amelia Blume
SOC 561
Community Resource

Stata loops: Preventing Errors and Debugging

Website Overview

Welcome! This website is here to help guide you through the most common ways of preventing errors and debugging loops and nested loops in STATA. You'll notice several tabs at the top of this page. The first tab, preventing errors, includes information on the stata commands capture and nostop. The second tab, debugging, includes information on the stata commands trace, pause, and assert. Both of these tabs include useful pictures and instructions to help you navigate the use of these commands. The final tab, additional resources, provides links to other useful pages on preventing errors and debugging.

This website exits as a part of a larger collective of resources on data management in stata overseen by Jennifer Earl, PhD. More information about these resources, as well as links to the main webpage can be found under additional resources.

I hope you find the information located here useful as you construct your own loops!

Preventing Errors: capture and nostop

CAPTURE:

Capture allows stata to continue running by suppressing possible errors. Capture is

most commonly seen in programming when it's use will not negatively impact the rest of a program. For example, running the command capture log close at the beginning of a new dofile will close a log if you happen to have left one open the last time you were using stata, but will not create an error if there was no open log.

When using capture when creating loops, you should be sure to NOT overuse this command, as you will not be able to trace where your errors occured. Rather capture should only be used when you anticipate an error will occur and you are aware of how the program is breaking and why.

The following provides further overview of the capture command:

Title

[P] capture — Capture return code

Syntax

capture [:] *command*

capture {
stata_commands
}

Description

capture executes *command*, suppressing all its output (including error messages, if any) and
generated by *command* is stored in the built-in scalar _rc.

capture can be combined with {} to produce capture blocks, which suppress output for the bl
more information.

Remarks

capture is useful in do-files and programs because their execution terminates when a commar
commands with the word capture allows the do-file or program to continue despite errors. A
appropriately to any situation by conditioning their remaining actions on the content of th

capture can be combined with noisily to display the output and any error messages regardles

. capture noisily regress y x

will either display an error message and store the return code in _rc or display the output

NOSTOP:

Like capture, nostop allows stata to suppress errors and continue to run. The nostop command tells stata to keep running an entire dofile, rather than just a single line of code, like capture. nostop should only be used in very rare circumstances.

However, nostop, does have some utility. According to Jenkins (2006) nostop can be useful when you have written a command and want to produce a script that documents how incorrect syntax leads to a specific error.

Again, nostop should only be used in very rare circumstances with a specific intent. Over use of this command can result in flawed programming and you may not be able to identify where your errors occured or how to fix them-- thus creating more work for yourself.

Debugging: trace, pause, and assert

TRACE:

The trace command allows you to observe what each line of loop is evaluating as. This should be done for all nested loops, in order to check that each is iterating. Trace can be especially useful when you are first learning how to create loops, as it allows you to see information that is usually hidden that may be useful in troubleshooting. Below you will find the basic syntax for trace as well as an example of the utility for using trace to debug loops.

```
Title

    [P] trace -- Trace program execution


Syntax

    Whether to trace execution of programs

        set trace { on | off }
```

USING TRACE:

Let's say I want to create a dummy variable for whether or not there was a blockade during any of the protest activities reported by respondents. First I run my syntax for this loop with out trace turned on and get the following result:

```
. set trace on


.
. gen blockade=0


.
. forval i = 1(1)4 {
  2.          replace blockade=1 if act`i'==57
  3. }
- forval i = 1(1)4 {
- replace blockade=1 if act`i'==57
= replace blockade=1 if act1==57
(508 real changes made)
- }
- replace blockade=1 if act`i'==57
= replace blockade=1 if act2==57
(299 real changes made)
- }
- replace blockade=1 if act`i'==57
= replace blockade=1 if act3==57
(132 real changes made)
- }
- replace blockade=1 if act`i'==57
= replace blockade=1 if act4==57
(46 real changes made)
```

```
. set trace on

.
. gen blockade=0

.
. forval i = 1(1)4 {
  2.          replace blockade=1 if | act`i'==57
  3. }
- forval i = 1(1)4 {
- replace blockade=1 if | act`i'==57
= replace blockade=1 if | act1==57
|act1 invalid name
  }
r(198);
```

```
. gen blockade=0

.
. forval i = 1(1)4 {
  2.          replace blockade=1 if | act`i'==57
  3. }
|act1 invalid name
r(198);
```

While the above notifies me of what the error in my syntax was, it does not show me exactly where it occured or why. By turning trace on, I get the following result:
Even when an error does not occur, having trace turned on allows you to be certain that your loops are running correctly. The following shows the creation of the same dummy variable, but this time without errors. When trace is not turned on it reports that changes were made, but not what they are.

```
. gen blockade=0


.
. forval i = 1(1)4 {
  2.          replace blockade=1 if act`i'==57
  3. }
(508 real changes made)
(299 real changes made)
(132 real changes made)
(46 real changes made)
```

Turning trace on allows you to see exactly what your loop has done, step by step. This way if there are any errors they are easier to find and assess.

PAUSE:

Pause allows control to return to the keyboard during the execution of a dofile. This allows for data to be checked and allows you to stop a dofile and fix errors if need be.

Below you will find the basic syntax for pause as well as an example of the utility for using pause to debug loops.

Title

[P] pause — Program debugging command

Syntax

pause { on | off | [*message*] }

Description

If pause is on, pause [*message*] command displays *message* and temporarily suspends execution of t
to the keyboard. Execution of keyboard commands continues until you type end or q, at which tim
resumes. Typing BREAK in pause mode (as opposed to pressing the Break key) also resumes program
signal is sent to the calling program.

If pause is off, pause does nothing.

Pause is off by default. Type pause on to turn pause on. Type pause off to turn it back off.

```
Title

    [D] assert -- Verify truth of claim


Syntax

        assert exp [if] [in] [, rc0 null fast]

    by is allowed; see [D] by.


Description

    assert verifies that exp is true.  If it is true, the command produces no output.  If it is not
    "assertion is false" and issues a return code of 9; see [U] 8 Error messages and return codes.


Options

    rc0 forces a return code of 0, even if the assertion is false.

    null forces a return code of 0 on null assertions.

    fast forces the command to exit at the first occurrence that exp evaluates to false.
```

USING PAUSE:

Let's return to our previous example, of creating a dummy variable for protest activities. This time the error will be less obvious and we will need to pause and troubleshoot.

Below we can see that no real changes have occured, and all of the data appears as if none of the respondents reported blockades. The issue is the code for act`i' where a typo caused stata to replace bloackade=1 for
act`i' ==577 instead of 57

```
. gen blockade=0

.
. forval i = 1(1)4 {
  2.          replace blockade=1 if act`i'==577
  3. }
(0 real changes made)
(0 real changes made)
(0 real changes made)
(0 real changes made)
```

```
.
. pause

.
.
. tab blockade, missing
```

blockade	Freq.	Percent	Cum.
0	23,616	100.00	100.00
Total	23,616	100.00	

```
. gen blockade=0

.
. forval i = 1(1)4 {
  2.          replace blockade=1 if act`i'==57
  3. }
(508 real changes made)
(299 real changes made)
(132 real changes made)
(46 real changes made)

.
.
. tab blockade, missing
```

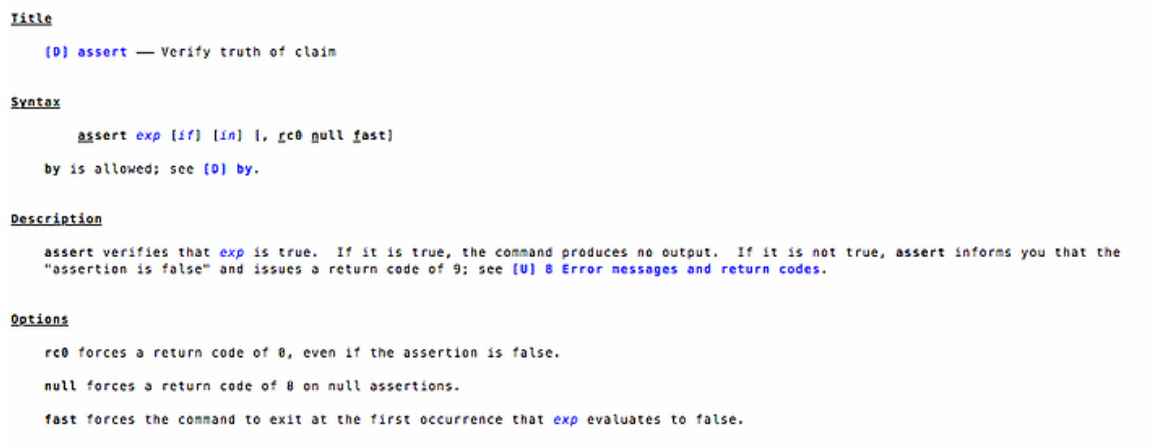
blockade	Freq.	Percent	Cum.
0	22,631	95.83	95.83
1	985	4.17	100.00
Total	23,616	100.00	

Once the problem is assessed, I can fix my code and continue running the rest of my dofile.

ASSERT:

Assert is a guard statement that protects against unexpected errors. Using a guard statement when creating loops is a best practice technique and helps to assure that loops are generating properly. Specifically, assert tells stata that a specific statement must be true, and if it is not, the dofile should break and stop running.

Below you will find basic syntax for assert, as well as its options.



```
Title

    [D] assert — Verify truth of claim

Syntax

        assert exp [if] [in] [, rc0 null fast]

    by is allowed; see [D] by.

Description

    assert verifies that exp is true.  If it is true, the command produces no output.  If it is not true, assert informs you that the
    "assertion is false" and issues a return code of 9; see [U] 8 Error messages and return codes.

Options

    rc0 forces a return code of 0, even if the assertion is false.

    null forces a return code of 0 on null assertions.

    fast forces the command to exit at the first occurrence that exp evaluates to false.
```

Additional Resources

For further information and resources on preventing errors and debugging loops in STATA please see:

From the Stata Journal:

NOSTOP:
http://www.stata-journal.com/sjpdf.html?articlenum=pr0023

LISTS:
http://www.stata-journal.com/sjpdf.html?articlenum=pr0005

From Stata (Official):

TRACE:
http://www.stata.com/support/faqs/programming/debugging-program/

Other Resources:

Top 10 STATA 'gothcas' (Including capture and other macro errors):
http://www.ifs.org.uk/docs/stata_gotchasJan2014.pdf

Other Community Resources on Loops:

https://www.youtube.com/watch?v=WsqvGdqzknY

http://hrstenson.tumblr.com/

http://loopscontinued.tumblr.com/

All STATA Programming Community Resources:

http://jearl.faculty.arizona.edu/node/19