Top 10 Stata 'gotchas'

Jonathan Shaw*

January 2014

Abstract

Stata is a powerful and easy-to-use package for setting up data and performing statistical analysis. Nevertheless, some features often catch users out, resulting in unnoticed errors or hours working out what has gone wrong. I list my top 10 Stata 'gotchas' and suggest ways to combat them, with the aim of helping new users avoid the most common pitfalls.

1 Introduction

Stata is an excellent package for setting up and manipulating data and performing statistical analysis. Once you've got over needing to type things at the prompt, it is easy to use and is well documented. Nevertheless there are some features that often catch users by surprise—even users with considerable Stata experience. In this article, I run down my list of the top 10 'gotchas' likely to spring a surprise and suggest ways to combat them. My aim is to help new users steer clear of some of the most common errors and to provide those teaching Stata with a list of pitfalls to point out.

2 The gotchas

Here is a countdown of my top 10 gotchas, listed in order of increasing likelihood of causing problems.

2.1 Gotcha 10: Invalid initial values become missing values

Stata has five different types of numeric variable: `byte`, `int` and `long` for integer values and `float` and `double` for values that may not be whole numbers. There are maximum and minimum values that each variable type can store. For example, `byte` variables can store numbers between -127 and 100 (see [D] **data types** for the others). If you specify an initial value outside these maximum and

*Thanks to Joy Wang at StataCorp for her suggestions of gotchas to include.

minimum values, the result is missing values, as the following output demonstrates:

```
. clear all
. set obs 10
obs was 0, now 10
. generate byte var1 = 101
. summarize var1
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
        var1 |          0
```

Notice that you get no warning that missing values are being created. The only case when you do get a warning is if values are outside the acceptable range for double variables, which is enormous. This differs from the behaviour for variables that already exist. If the value is outside the permitted range, then the variable is automatically promoted to a larger storage type if possible (`byte` to `int`, `int` to `long`, and `long` and `float` to `double`) and a message displayed. If the variable is already a `double`, missing values are created and a warning printed.

```
. clear all
. set obs 10
obs was 0, now 10
. generate byte var1 = 1
. replace var1 = 101
var1 was byte now int
(10 real changes made)
```

You're most likely to get caught out by this gotcha if you're in the (good) habit of specifying the variable type when creating variables (e.g. `generate byte ...`). It's easy to specify a `byte` variable without thinking, and end up with a bunch of missing values. But the problem also occurs if you don't specify the type and the initial value is too big to fit in the default `float` type.

2.2 Gotcha 9: Backslash prevents macro expansion

The Stata manual describes macros as 'the variables of Stata programs'. They come in two flavors — local and global — and are an indispensable part of automating repetitive tasks. In my own work, I use macros for a wide range of tasks including storing file paths, constructing variable names in loops, accumulating graph commands and formatting output. Sometimes it's useful to be able to prevent a macro from expanding. For example, you might want a string to contain a reference to a local whose contents will only be determined at a later stage. The way to achieve this is to preface the macro reference with a backslash: `\`local1´`. The resulting string will contain the unexpanded local reference less the backslash, meaning that the local will get expanded each time the string is used. This works fine in most places, but it can cause trouble with Windows file paths, which are separated using backslashes. For example,

you might want to open a dataset whose name is stored in the local `datafile`. Here's what you might try typing, and the outcome:

```
. use "C:\data\`datafile´", clear
file C:\data`datafile´.dta not found
```

Notice that the local `datafile` has not been expanded because the backslash has been interpreted as saying: 'don't expand this local'. To avoid this, change how you indicate file separators immediately before macros: either use a double backslash, `\\`, or use forward slashes, `/`, which are understood as path separators even on Windows machines. See [U] **18.3.11 Constructing Windows filenames by using macros** for more on this.

2.3 Gotcha 8: Each execution of a do file has its own local space

In order for a do file—or a selection—to run correctly, it is often necessary for a number of locals to be set. For example, to execute code found in a loop, the loop index (e.g. `i`) must be set. New Stata users can often be confused by the fact that locals they defined interactively seem to have vanished when when a do file is executed: 'I just set `i`—what happened to it?!!'

The reason for this behaviour is that locals defined interactively are out of scope when a do file is run. Each do file has its own local space, and the only locals it can see are those defined in that do file. Moreover, once a do file has finished executing, all of its locals are deleted, meaning that they need to be recreated if the do file is run again. Note that if the do file contains programs, each program (defined by the `program define` command) has its own local space. See [U] **18.3.3 The difference between local and global macros** for details.

It is possible to make locals created in a do file available interactively once the do file has finished executing. This is achieved using the `include` command. As far as I know, this only works for whole do files (rather than selections in the Do-file Editor window). There doesn't seem to be any way of making locals defined interactively available from a do file.

2.4 Gotcha 7: The capture command hides all errors

Prefixing a command with `capture` stops that command from crashing when there's a problem and suppresses all error messages. This can be useful e.g. when you want to drop a program if it's in memory but not to worry if it's not. The downside is that it hides *everything* that goes wrong with the command. For example:

```
. generate byte var1 = 5
. generate byte var2 = 10
. capture drop var1 var2 var3
. describe var1 var2
```

```
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
var1            byte    %8.0g
var2            byte    %8.0g
```

This code creates two variables, `var1` and `var2` and then tries to drop them together with a third variable, `var3` that doesn't exist. You might expect this to result in `var1` and `var2` being dropped but that's not what happens. The `capture` command means you might not find this until much later, if at all.

This is the sort of thing it's easy to think won't catch you out because you can't imagine how it could end up being catastrophic. I thought that, and lived to regret it! The best way to avoid problems is to use `capture` sparingly and always check for a nonzero return code afterwards for signs of problems. This is done using the system variable `_rc` (see [U] **13.4 System variables (_variables)** for details).

2.5 Gotcha 6: Syntax errors in the syntax command

Stata's `syntax` command greatly simplifies writing your own commands that adhere to the standard Stata syntax. With it, you can specify the number of variables the command should accept; whether it permits an `if` condition, a number range and weights; a file path if relevant; and a list of possible options. All of this is acheived in a single line of Stata code (see [P] **syntax** for more details). Perhaps not surprisingly, given how powerful the command is, it is easy to make errors in the syntax command itself. One particularly troublesome case involves forgetting to specify a default value for an optional numeric option, as in:

```
. syntax varname, [bwidth(real)]
invalid syntax
r(197);
```

The error in this is not easy to spot, even for seasoned Stata programmers, and the error message doesn't do much to help. Because `bwidth` is optional, it should specify a default value:

```
syntax varname, [bwidth(real) 0.1]
```

Just to complicate matters, it's not an error to omit a default value for an optional string (rather than an optional integer or floating point value).

If you're using `syntax` commands and you get an invalid syntax error message, it is often worth checking whether any of the syntax commands are responsible (e.g. by displaying some text immediately before and after them). Being alert to this kind of issue will save you a good deal of time debugging your do files.

2.6 Gotcha 5: Missing values aren't always treated as infinity

In most places in Stata, missing values (denoted by a period, '.') are treated as being infinity. Thus, if you want to count how many positive, non-missing observations a variable has, you need to exclude the missings explicitly:

```
count if ((var1 > 0) & (var1 < .))
```

Missing values usually propagate across commands, as in the following (the period on the second line indicates missing):

```
. display 5 + .
.
```

But in some commands, such as the `sum` command, missing values are treated as zeros:

```
. clear all
. set obs 4
obs was 0, now 4
. generate byte var1 = cond(mod(_n,2)==1, 1, .)
(2 missing values generated)
. generate byte var1sum = sum(var1)
. list, noobs
```

var1	var1sum
1	1
.	1
1	2
.	2

The `sum()` function calculates the cumulative sum, counting from the top to the bottom of the dataset. You can see fro the values of var1sum that missing is treated as zero.

In others cases, missing values are treated as not being there:

```
. display max(-5,.)
-5
```

These difference in behaviour across commands are confusing and can lead to mistakes, particularly when doing complicated operations to set up data. In my experience, it's best to step through the creation of variables line by line as you write the code, checking that each step does what you expect it to do, particularly for pathological cases.

2.7 Gotcha 4: No observations satisfy equality condition

An expression testing whether a variable is equal to a specific value can sometimes not be satisfied by any observations, even when you are sure that there should be some matching cases. For example, consider the following output:

```
. clear all
. set obs 10
obs was 0, now 10
. generate id = _n
. generate frac = id/10
. assert (10*frac == id)
8 contradictions in 10 observations
assertion is false
r(9);
```

This creates an `id` variable that counts from 1 to 10 and a `frac` variable that is `id` divided by 10. It then asserts that 10 times `frac` is equal to `id`—which you'd think should hold in every case—but we get 8 contradictions.

To understand why this happens, we need to know a bit about how variables are stored in Stata. All numeric variables are represented internally in binary (1s and 0s). As noted earlier, there are two broad classes of numeric variable: integer variables used to store whole numbers (`byte`, `int` and `long` types) and floating point variables used to store any numbers, particular those that are not whole numbers (`float` and `double` types). If you don't specify the type when generating a variable, Stata uses the default `float` floating point type. Floating point variables store values up to a finite degree of precision. Some numbers can be represented exactly in binary, such as whole numbers that are not too big. Many other numbers cannot be represented exactly, including 0.1 and many multiples of it. What gets stored is the best available approximation.

We're now in a position to understand what's going on in the code above. `frac` contains approximations of the numbers 0.1, 0.2,...,1.0. Of these, the only ones that can be stored exactly are 0.5 and 1. Multiplying the others by 10 won't result in the whole numbers we would expect. To avoid this error, you should never check whether a floating point variable is equal to some value; instead, you should check whether it is very close to that value. For example, the last line of the code above would have been better if it read:

```
assert abs(10*frac - id) < 0.00001
```

Check [U] **13.11 Precision and problems therein** some alternative solutions.

A related issue is when distinct numerical values end up being stored as the same value because they are sufficiently large that a given data type is unable to distinguish between them. One particularly problematic case is when long IDs are read into a float variable, something that can quite easily happen using the `infile` command. Floats can store every whole number up to 16,777,216 (2^{24}). Beyond that, there are some gaps. At first it's every odd number that is missing (16,777,217, 16,777,219, 16,777,221, etc); after 33,554,432 (2^{25}) every other even number is also missing. Gradually more and more whole numbers can't be represented. These numbers are stored as the closest number that can be represented. For example, 16,777,217 is stored as either 16,777,216 or 16,777,218. To avoid this, use the `import delimited` command in preference to `infile` if possible (`import delimited` better tailors the data type to the variable contents). If this is not possible, it is important to check that the range of your variables won't cause problems.

2.8 Gotcha 3: Variables take preference over scalars

Expressions in Stata can contain references to variables and to scalars (as well as other things like macros). Variables are allowed to have the same name as scalars, opening up the potential for confusion. Stata deals with this by assuming first you're referring to a variable, and only looking for a scalar if no suitable variable is found. What makes this really confusing is abbreviation of variable names. Under default settings, Stata allows you to abbreviate variable names so long as the abbreviation is unambiguous relative to other variable names. In expressions, variable abbreviations are taken in preference to scalars. This can result in some unexpected behaviour, as the following code snippet demonstrates:

```
. clear all
. set obs 10
obs was 0, now 10
. generate byte id = _n
. scalar i = -6
. display i
1
```

You might expect the `display` command at the end to display the contents of the scalar `i`, which is -6. In fact, `i` is taken as a valid abbreviation of `id`, so it displays 1, which is the value of `id` for the first observation. It's very easy to get caught out by this if you don't realise a scalar name you've used is an unambiguous abbreviation of a scalar name.

One simple way to stop this happening is to turn off variable abbreviation (`set varabbrev off`). It is also a good idea to get in the habit of always referring to scalars in expressions using the `scalar()` function. This avoids any confusion.

2.9 Gotcha 2: Preserved data is automatically restored

The `preserve` command saves a temporary copy of your data that you can later revert to using the `restore` command. I use it to provide a recovery point in case of an error, in merging datasets and to perform repeated analysis that requires the same starting dataset.

One feature of `preserve` that catches many users by surprise is that, if you preserve data in a do file, the data is automatically and silently restored when the do file finishes even if no restore command has been reached (including when the do file crashes!) This is illustrated by running following code containing an error from a do file:

```
sysuse nlsw88.dta
preserve
collapse (mean) wage, by(age)
lin wage age
restore
```

The error is that `lin` should have read `line`, as the error message makes clear.

Often, however, it is less obvious what has gone wrong and some detective work is required. One good way of doing this can be to look at the current state of the data (e.g. the last variable that has been created). But, in this case, looking at the data after the crash might lead you to believe that the `collapse` command hadn't been reached. In fact it had, but Stata restored the preserved data when it encountered the error.

Automatically restoring data after a do file terminates is desirable behaviour to minimise the damage caused by errors, but it can cause hours of head-scratching if you don't realise what is happening. To avoid the automatic restore, you can preserve the data outside the do file, or use the `pause` command to pause execution of the do file at a specified point. Alternatively, you can use the `snapshot` command instead of `preserve`.

2.10 Gotcha 1: Misspelt macro names expand to nothing

And so to the number 1 gotcha: misspelt macro names. One useful feature of macros is that expanding a non-existent macro doesn't result in an error but instead expands to nothing. To see how this might be useful, consider the following code that accumulates a list of variables:

```
forvalues i = 1/5 {
  local vars "`vars´ var`i´"
}
```

Assuming the local `vars` didn't exist before the `forvalues` loop, afterwards it will contain '`var1 var2 var3 var4 var5`'. But notice that this only works because expanding a non-existent macro in the first iteration of the loop didn't cause it to crash.

The downside of this behaviour is that mistakes typing macro names can easily go unnoticed. For example, instead of referring to the local `vars` you might accidentally type `vras`. Stata would only complain if what `vras` contains (probably nothing) isn't what it was expecting. This can be dangerous and is why programming languages often don't allow you to refer to a variable before it has been declared. To see the sort of damage that can be done, here's a mistake I made just a few months into my first job:

```
tempfile temp
save `tepm´, replace
```

The local tepm didn't exist so expanded to nothing. As a result, the command reduced to:

```
save, replace
```

overwriting the whole of a dataset I really didn't want to lose!

The best way to avoid this happening depends on the context. For the case that caught me out, omitting the `replace` option on the `save` command or making the dataset a read-only file would have caught the error. In other situations, different solutions are likely to be appropriate. In general, however,

it is always important to take particular care with code that changes variables or datasets in ways that can't easily be reversed.

2.11 Honorable mentions

A number of gotchas didn't make it onto the top 10, but are nevertheless worth a brief mention.

The first two relate to the `merge` command, which allows you to join observations from two datasets based on identifier variables. An important shortcoming with the command is that, if there are any identically-named variables in the two datasets in addition to the identifier variables, then the ones in the dataset in memory (the 'master dataset') will overwrite those in the dataset being loaded from disk (the 'using dataset'). No warning message is given. This might be what you wanted, but it can cause havoc if you didn't realise there was a clash.

The second problem with `merge` involves forgetting to specify what to merge on. Prior to Stata 11, if you omitted the list of variables to merge on, Stata thought you wanted to merge by observation (i.e. match the first observation in the master dataset with the first observation from the using dataset, and so on for the remaining observations). This could result in hours of head-scratching if it wasn't what you intended. This syntax continues to work, but a warning is displayed when it is used. Obviously it is preferable to use the new merge syntax, which prevents this problem from happening.

We've already had one gotcha about the `syntax` command. Another issue relates to the treatment of options without arguments whose names start with the letters '`no`' such as `notes`. Normally, the syntax command will copy options specified by users into locals with the same names as the options. In this case, however, the local to look in will be called `tes`. This is because Stata interprets this option as wanting to turn off `tes` (i.e. 'no tes please'). It's easy to waste quite a while trying to work out why your notes aren't being displayed when the reason is that you're looking in the wrong local!

The final honorable mention also relates to a previous version of Stata. Prior to Stata 13, string expressions could have a maximum of 244 characters, and string functions could return a maximum of 244 characters. This was strange because strings themselves were allowed to be extremely long. It was exceedingly easy to truncate long variable lists or commands stored in a local by manipulating them using a string function. The best ways to stop this happening were to avoid string expressions unless necessary and to use extended macro functions rather than string functions.

3 Conclusion

I have listed my top 10 'gotchas' likely to catch users out suggested ways to combat them. No doubt you will have your own suggestions to add to the list. My hope is that those set out here will help make new users and those teaching Stata aware of the most common pitfalls.