The Stata Journal (2018)
18, Number 4, pp. 981–994

Speaking Stata: Seven steps for vexatious string variables

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

Clyde B. Schechter
Albert Einstein College of Medicine
Bronx, NY
clyde.schechter@einstein.yu.edu

Abstract. String variables that seemingly should be numeric require some care. The column provides a step-by-step guide explaining how to convert them or—as the case may merit—to leave them as they are. Dates in string form, identifiers and categorical variables, and pure numeric content trapped in string form need different actions. Practical advice on good and not so good technique is sprinkled throughout.

Keywords: dm0098, string variables, numeric variables, dates, encode, egen, destring

1 Introduction

Here is a common Stata user experience. After opening a data file, you realize—or at least suspect—that you have string variables that should be converted or otherwise mapped into numeric variables. Usually, you are right. Often, however, people get confused about precisely how they should achieve that in Stata. The possibilities and the pitfalls are both numerous. The aim of this column is to provide a brief seven-step survival guide that—even if it does not guarantee to solve all of your problems—will give you a sketch map of the territory. We flag commands and functions that can help and signal where to learn more.

If this is your first introduction to the subject, you may be starting in the wrong place. You could go first to [U] **12 Data** to get an overview of data in Stata and the different kinds of variables possible.

If you think you are starting in a good place, the following summary should make sense.

The difference between numeric and string variables in Stata is a big deal. A numeric value like 42 within a numeric variable and a string value like `"42"` within a string variable are quite different. The double quotation marks (`" "`) are to Stata and to readers alike a signal that what they contain is a character string and will be treated as such. A numeric value like 42 and a string value like `"Stata user"` are even more different, as should be clear. The more subtle territory is for cases like the first example whenever values like `"42"` should be treated as numeric because the numeric interpretation is essential. Perhaps, Stata misread your data so far as you are concerned. Or, perhaps, you knew that a variable would be read in as string and are not surprised that you need to convert it.

2 Seven steps

2.1 Keep copies

Always keep copies of original data.

Keep copies of the dataset you start with as (say) a `.dta` file using `save` (see [D] **save**). Further, what may be as or more pertinent, do not discard any original data file, say, a text or spreadsheet file, that you used to `import` (see [D] **import**) the data. Sometimes, you may need to go back and start again.

Also, within a Stata session, keep the original string variables. Even if a command offers a `replace` option to overwrite the original variables, use its `generate()` option instead—unless and until you know exactly what you are doing, or you are not averse to the risk of making a mess and having to try again. This is always good advice, but because string conversions can easily go wrong, you should have a line of retreat so that you can pull back and try again.

2.2 Dates within strings usually require date-time functions

Does any string variable contain a date or date-time string? If so, a date-time function is usually needed to convert it to numeric.

This is usually easy to answer. You look inside the string variable using, say, `list` or `edit` (see [D] **list** or [D] **edit**), and see values like `"July 4, 1776"` or `"28 March 1952"` or anything that smacks of a half-yearly, quarterly, monthly, weekly, or daily date. Or you look inside and see a date-time, such as `"28 March 1952 02:00:00"`, which was 2 a.m. or 02:00 on 28 March 1952.

However, if the variable is just calendar years held as strings, with values like `"1984"` or `"2018"`, the solution will usually be to use `destring` (see [D] **destring**).

Otherwise, if the string variable holds a date or time, you almost always need to use date-time functions to convert it to a numeric date or time variable. In this circumstance, do not ever attempt to use `encode` or `destring`. (Later, we will say more on why not.)

The conversion is usually a two-step process.

First, find a conversion function. Consult `help datetime` to find the function you need. So daily date strings can be converted to Stata numeric dates with `date()` or `daily()`. We prefer to use `daily()` to set a good example to ourselves and others. The `date()` function was introduced in Stata when daily dates were the only kind of date receiving special support. (Years like 1984 or 2018 can usually be treated as they come in statistics, graphics, and data management alike.) However, it is too easy to guess that `date()` is a generic date conversion function. On the contrary, it does one thing only, yield numeric daily dates. The two functions `date()` and `daily()` share the same code, but `daily()` is the precise name, which we therefore recommend.

So given some string variable, `strdate`, say, your first step will be something like

```
. generate numdate = daily(strdate, "MDY")
```

if you have dates like `"July 4, 1776"`. It will be something like

```
. generate numdate = daily(strdate, "DMY")
```

if you have dates like `"28 March 1952"`.

We are not going to go into much more detail on the date functions you might need. The main point for now is that a date conversion function will map string dates such as `"July 4, 1776"` and `"28 March 1952"` to numeric values, here −67019 and −2835. These can be explained as daily dates on a scale on which 1 January 1960 is 0 or the origin 0. More generally, Stata's origin for any kind of date or date-time (other than yearly dates) is 0 for the first possible date in 1960, and so for the first half year, quarter, month, week, day, or millisecond, as the case may be.

We said conversion function, but sometimes you need something more complicated than a single function. Here is one twist we have often seen: data are quarterly but arrive with string dates indexed by the last day in each quarter: 31 March, 30 June, and so forth. Here a daily date is not what you really need, because to Stata that would imply that you have 4 observations with nonmissing values out of 365 or 366 possible within each year, so your dataset is implied to be mostly gaps. Rather, what you need is to map these dates into quarterly dates using `qofd(daily())`. As in elementary algebra, the function on the inside, `daily()`, is implemented first, followed by `qofd()`, which, as you may well have guessed, maps daily dates to quarterly dates. (Think: `q`uarter `of` `d`aily date.)

We now should have a numeric variable. If the result is something unintelligible, what have we really gained? The answer lies in the second step of the process.

Second, assign the display format you want. If we now apply a date display format, then Stata knows to use that where relevant, especially in listings or on graphs. The command

```
. format numdate %td
```

assigns the default daily date format to a numeric daily date variable. The default is naturally just the default. For other possibilities, see

```
. help datetime display formats
```

There is one awkward exception to the rule of using date-time functions to convert date strings. Sometimes, your strings such as `"12:34:56.78"` or `"25:14:36"` indicate durations or elapsed times, so times from some start or stage or state to another. Fans of various sports in which people, vehicles, or animals race each other should immediately be at home here.

Sometimes, such durations can be held as Stata date-times, in which the only allowed unit is milliseconds. If so, then the implication that the clock started at 00:00:00.000 on 1 January 1960 must usually be ignored as incorrect and irrelevant. More commonly, it may be a better strategy to decide which numeric units you want to work in—commonly, minutes or seconds—and then use `split` (see [D] **split**) first and set up your own conversion to a numeric variable with those convenient units. Using Stata's `msofseconds()`, `msofminutes()`, and `msofhours()` functions to convert units would relieve you of the burden of a few multiplications and make your code more transparent and readable.

2.3 Identifiers and categorical variables may be mapped to numeric

If the string variable holds identifiers or categorical values, and you need a numeric variable to use in modeling or other statistical analysis, then you should map it to a numeric variable using `encode` (see [D] **encode**). `egen`'s (see [D] **egen**) `group()` function is another possibility.

Identifiers can be informative. People working in economics, business, finance, geography, or various other social sciences will be familiar with the use of company or country or regional names as identifiers. Here, typically, such string identifiers are informative. You (should) care about which observation is which. You should know something about Alabama or Albania or Aberdeen or Azerbaijan or Apple that informs your interpretation of your statistical work.

Identifiers might be uninformative. Conversely, people working in medicine or epidemiology or other health sciences will be familiar with patient, practitioner, or other identifiers that are deliberately anonymous. They may still be informative, in this sense. If the results for patient `"XYZ123"` look really odd, there may be scope to check back on, if not the original records, then some data source upstream of your Stata dataset.

Either way, in Stata numeric identifiers indicating panels are required before you can identify your dataset to Stata as such using `tsset` or `xtset` (see [TS] **tsset** or [XT] **xtset**).

Categorical variables may arrive as strings. If different strings are also different levels of a categorical variable, say, `"male"` or `"female"`, or `"Democrat"` or `"Republican"`, you will also need a numeric representation for most statistical purposes. Numeric variables are also often needed, or at least more widely usable, for graphical analysis.

The purpose of `encode` is to map identifiers or categories arriving as strings to numeric variables. That is also what `egen, group()` does, although the latter can be used in other ways too. We will look at these commands in turn after considering what kind of numeric representation is most helpful.

A common and excellent convention is to use successive integers from 1 up as numeric identifiers. There is no compelling logic behind this convention except simplicity and convenience. For example, you just need to know the highest numerical code to be able to loop easily over such categories. Typically, that means use of `forvalues` (see [P] **forvalues**).

Another common and even more excellent convention is to code binary states as 1 and 0. That leads directly to useful statistics. Three examples now follow making that point, but skip them if you know this well already.

- The mean of a binary variable coded 0 and 1 is the fraction, proportion, or probability of whichever state is coded as 1. Think of 7 females coded 1, 1, 1, 1, 1, 1, 1 and 3 males coded 0, 0, 0. The total of those codes is 7 and the mean is $7/10 = 0.7$, which is just the proportion of females out of your group of people. Coded the other way, with male coded 1 and female coded 0, then 0.3 is the proportion of males and 0.7 is easily found by subtraction. Either way, coding with 0 and 1 is not completely arbitrary because each coding comes with an interpretation as an indicator variable.
- Binary response variables (dependent variables) in models such as logit and probit regression should be coded 0 and 1 for the same reason. The aim of such modeling is prediction of the probability of the state coded 1, survival or success or whatever it may be.
- Dummy or indicator variables for binary states are used commonly as predictors in regression-type models.

Let's create a silly sandbox, three distinct names held in a string variable:

```
. input str4 name

           name
  1. frog
  2. toad
  3. newt
  4. end
```

You can just ask for a numeric variable that is equivalent with `encode`:

```
. encode name, gen(num_name)

. list

     +-----------------+
     | name   num_name |
     |-----------------|
  1. | frog       frog |
  2. | toad       toad |
  3. | newt       newt |
     +-----------------+
```

With `list`, the result looks just the same, but that is illusory. You are seeing value labels created by `encode`. Let's drill down to see the correspondence.

```
. label list
num_name:
           1 frog
           2 newt
           3 toad
```

Note the very simple rule, which should not surprise you. Unless instructed otherwise, `encode` uses the distinct string values of the variable it is fed and maps them in alphabetical (more generally, alphanumeric) order to integers 1 up. Here the order in the data was first `"frog"`, then `"toad"`, and last `"newt"`, but Stata ignored that.

Here it is obvious from a quick inspection that sorting to alphabetical order yields `"frog"`, `"newt"`, and `"toad"`. But what order is produced if leading characters include uppercase characters, numeric characters, or punctuation characters (including spaces)? A not quite tautological answer is that the order is precisely the order that the `sort` command would yield. If in doubt, be aware that Mata will also give you an answer using the corresponding `sort()` function:

```
. mata: sort(tokens("frog toad newt")´, 1)
          1
     +--------+
  1  |  frog  |
  2  |  newt  |
  3  |  toad  |
     +--------+

. mata: sort(tokens(`" " frog" "frog" "frog " "´)´, 1)
           1
     +---------+
  1  |   frog  |
  2  |   frog  |
  3  |  frog   |
     +---------+

. mata: sort(tokens(`" "| frog|" "|frog|" "|frog |" "´)´, 1)
             1
     +-----------+
  1  |  | frog|  |
  2  |  |frog |  |
  3  |   |frog|  |
     +-----------+
```

A key detail there is that `sort()` sorts columns, so row input needs to be transposed to column input, which is what the prime (`'`) does. Note the small trick of using pipe or conditional symbols (`|`) to make clear how spaces are handled. Many other characters would work as well.

A small but often irritating problem touched on in that example is the occurrence of leading or trailing spaces, which in our experience are always accidental and never informative. Usually, when found, they should just be removed with `trim()`. Not illustrated here, but usually also worthwhile, is standardizing internal spaces with `itrim()`. Help on such functions can be found under `help string functions`.

A glance at the help for `encode` shows that this default of assigning value labels automatically is not the only game in town. You can have any other preferred mapping to integers with value labels you like (a few details aside), but you must first tell Stata what that preferred mapping is. That means defining a set of value labels before you `encode`, or at least having one already in the dataset that some previous user put there.

A simple example: If you have a bundle of string variables all with values `"Yes"`, `"No"`, `"Don't know"` (and other similar responses), it will certainly make sense to have the resulting set of numeric variables share the same set of value labels. If you use the alphabetical default, then the coding will be 1 for `"Don't know"`, 2 for `"No"`, and 3 for `"Yes"`, which is unlikely to be the order you want to see in tables or graphs. So that is a case for defining labels carefully in advance.

If you do not think up all the possibilities in advance, `encode` will extend the set of value labels if it meets a string that you have not included in the definition of value labels. Sometimes, that is fine, and sometimes, it means some work cleaning up. So, if someone typed `"Dont know"`, omitting the apostrophe, Stata will not be smart on your behalf and realize that this really means the same as `"Don't know"`. But let's not look further down that side street. We cannot cover all the difficulties of data management in one column. We are not gonna (and dont wanna) get into all the strange spellings people might use.

For yet further tips on `encode`, see Schechter (2011).

In some circumstances, the `group()` function of `egen` is a more flexible alternative to `encode`. By default, it does not create value labels. That can be really useful. Note that at the time of writing (Stata 15.1), `encode` has a limit of 65,536 distinct values of the variable, which may make it unsuitable for use with identifiers in large datasets.

In a program, you may just want to cycle over the distinct groups, and you do not immediately care which is which. Or you do not care to keep hold of the names or other identifiers underlying the many panels or groups you might have. Other way round, this function does allow you to ask for value labels to be attached, but it does not offer any alternative to the default of using sort order to define those value labels. So, unlike with `encode`, you cannot insist on `egen` using a prespecified set of value labels.

For more in this territory of identifiers and categorical variables, see Cox (2007).

2.4 Some string identifiers can be left as they come

Sometimes, however, there is little or no gain in converting a string variable to numeric.

Sometimes, you have identifiers that identify individual observations but that you are not going to use in any graphical or statistical analysis. That is especially likely if you have many such identifiers and perhaps only a few observations for each (quite possibly, just one). The identifiers might be personal identifiers, even entirely numeric, such as nine-digit (contractions of) Social Security Numbers in the United States.

For whatever reason, suppose that you are not going to be using such variables in calculations. Sometimes, you will save some storage by making them numeric with value labels, but unless memory is short, they can often be left as they came.

It is tempting to say that such identifiers have no use, yet it is always prudent to leave identifiers in a dataset. As already mentioned, sometimes you need to check individual cases against other records. Another example is that you need identifiers to merge (see [D] **merge**) with other datasets. Yet another example is that unique identifiers provide a standard, repeatable way to get a dataset in a particular order. (When that matters, you really want it.)

2.5 Do not try to import metadata

Sometimes, your data have been read in as string because metadata have been included by mistake. That needs to be fixed before you can do anything very productive with the data.

Spreadsheet data files—and many other data files—often include metadata, commonly but not only as header lines. Even a minimal, Spartan dataset in a worksheet may include one line of column headers explaining what is in each column.

Here "metadata" refers to information about the data that may include definitions, explanations, or comments. In Stata, such information may belong in the dataset as variable names, variable labels, value labels, or notes or other characteristics. Or metadata indicate how missing or other exceptional values arose or are coded. Sometimes, the information is interesting or useful and contributes to interpretation, but you do not need to include it all in a dataset.

The dilemma is simple. Because Stata permits string variables, it will not exclude columns or fields in a dataset that appear to be string or that include some values that can be treated only as string. But it cannot decide on your behalf what should and should not be part of the data. It can even make sense to have legitimate variable names, variable labels, and value labels as values of string variables.

If you read in metadata by mistake, you can try dropping the problematic observations and then seeing what else is needed, typically an application of destring (see section 2.6). However, often it is best to go back and import the data again. Commands such as import excel and import delimited (see [D] **import excel** and [D] **import delimited**) have key options to allow skipping of rows and columns in the data file. Diehard devotees of the command language who prefer to avoid menus or user interfaces may still want to experiment with these commands called up with dialog boxes. Typing (say)

```
. db import excel
```

has the very helpful side effect that the dialog shows you how Stata "sees" the file. Often, it then becomes obvious that there are lines of stuff that you do not want as data.

Commands such as `import excel` called up through dialog boxes echo to the Command window (and thus any open log files) the command syntax that you could have typed. That can be especially helpful when you have several files with the same format or layout and wish to loop over those.

The problem can be worse than so far explained. Text that prevents Stata from reading in columns as numeric variables could occur anywhere in a data file. In a large file, that could mean problematic material in unpredictable places that thus can be hard to find, except that section 2.6 offers advice that can help.

We have collaborated with keen spreadsheet users who sometimes remember, but often forget, that they sprinkled comments capriciously but with good intentions in odd corners of their worksheets. However, no homunculus inside Stata can be smart on your behalf to work out what should be ignored and what should be included in the dataset.

2.6 destring is your friend

If you have got to here, you have ruled out your string variable as being a date; or as being an identifier or categorical variable to be `encode`d; or as being contaminated by metadata. Then your best guess is that `destring` is what you need, but there is still a question of why Stata read in your data as string, when on your story they should be numeric.

The key lies in exploiting the options of `destring` to the full. It can happen that once you have (say) removed metadata, then all of what remains in a variable may allow interpretation as numeric, but you will not always be so fortunate.

Do familiarize yourself with the options `ignore()`, `percent`, and `dpcomma`.

Researchers can be tempted to try `destring, force` to fast forward the otherwise tedious task of data management. But, as in the rest of life, `force` is the choice of despair when all else has failed and you know that a small degree of force will get the result you need.

The most common easy problems for `destring` to solve are

- non-Stata codes (which may make perfect sense otherwise) for missing or otherwise intractable values. For example, the data source may have used `"NA"` to code missings. In that latter case,

  ```
  . destring whatever, generate(wanted) ignore("NA")
  ```

  will remove such codes; that will typically leave empty strings, which `destring` will regard as implying numeric missing values.

- punctuation in the widest sense, say, commas used as decimal points (say, 12,3 for 12.3) or % signs to indicate percents.

2.7 Find out what cannot be converted

A short glance at your dataset or long experience with Stata or particular kinds of dataset may allow a quick guess at what is wrong, as just explained. Often, you need something more systematic to look at all the values that cannot automatically be handled by `destring`.

Consider the following code:

```
. tabulate whatever if missing(real(whatever))
```

Let's spell that out. You are asking to `tabulate` a problematic string variable whenever (i) you try to convert it to numeric using the function `real()` and (ii) it is true that the result is missing.

Let's take that even more slowly. `real()` is the brute-force function underlying (and predating) `destring`. If in doubt, it returns missing. So consider these examples using `display`:

```
. display real("42")
42
. display real("42%")
.
. display real("12O")
.
```

The function `real()` is not especially smart. It gives up at the slightest kind of difficulty, shrugging its shoulders and emitting numeric missing. So the percent sign `%` is beyond its understanding. The letter `"O"` in the last example is a problem too far (did you spot it?). Sometimes, however, a researcher knows enough to be confident to guess that characters have been mistyped; perhaps, the letter "O" should be the number 0, and the letter "I" should be the number 1.

It is because `real()` is not smart enough to convert numeric content trapped inside string containers that we need `destring`, which has extra bells and whistles attached to cope with most difficulties.

Now that we have explained what `real()` is doing, we can focus on the `missing()` function. It has just one job and one aim in life, to return 1 if you feed it something that is missing and 0 otherwise. Thus, how does that work with our little examples?

```
. display missing(real("42"))
0
. display missing(real("42%"))
1
. display missing(real("12O"))
1
```

So our handle for looking at what `real()` cannot convert to numeric is the qualifier

```
... if missing(real(whatever))
```

Notice that we do not need to type

```
... if missing(real(whatever)) == 1
```

because there is no need to test whether `missing()` returns 1 or 0: the function is doing that any way.

We have emphasized the scope for

```
. tabulate whatever if missing(real(whatever))
```

which ideally will show the kinds of strings that `destring` will struggle with—unless we fix the problem first or identify an appropriate option of `destring` to fix the problem directly. Other commands can be useful with the same qualifier, such as

```
. list whatever if missing(real(whatever))
. browse whatever if missing(real(whatever))
. edit whatever if missing(real(whatever))
```

Although this is a simple device, it will sometimes produce puzzling output whenever the problem is control characters that cannot be displayed. Your strings will then look fine but will not be. Nor can they be removed through, say, `trim()`. We will just flag this as an extra problem that deserves discussion at length, while hinting that the `char()` and `uchar()` functions provide ways of identifying awkward characters. Again, `help string functions` is the place to start.

On the whole, we do not especially recommend using `edit` here to change your data even if you are keeping a log of all that you are doing. It is best to identify from close scrutiny of problematic strings what the generic problems are and then fix them through `destring` options. One benefit of that is whenever data files from the same source need to be treated in the same way. There is a clear gain to having a single command rather than being obliged to fire up `edit` to make changes one by one.

3 Recap and review

We can now recap the seven steps recommended:

1. Always keep copies of original data.
2. Does any string variable contain a date or date-time string? If so, a date-time function is usually needed to convert it to numeric.
3. If the string variable holds identifiers or categorical values, and you need a numeric variable to use in modeling or other statistical analysis, then you should map it to a numeric variable using `encode` (see [D] **encode**). `egen`'s (see [D] **egen**) `group()` function is another possibility.
4. Sometimes, however, there is little or no gain in converting a string variable to numeric.

5. Sometimes, your data have been read in as string because metadata have been included by mistake. That needs to be fixed before you can do anything very productive with the data.

6. If you have got to here, you have ruled out your string variable as being a date; or as being an identifier or categorical variable to be `encode`d; or as being contaminated by metadata. Then your best guess is that `destring` is what you need, but there is still a question of why Stata read in your data as string, when on your story they should be numeric.

7. A short glance at your dataset or long experience with Stata or particular kinds of dataset may allow a quick guess at what is wrong, as just explained. Often you need something more systematic to look at all the values that cannot automatically be handled by `destring`.

The highlights therefore—whenever string variables really need to be mapped to numeric—are

- date-time functions such as `daily()` to produce numeric dates;
- `encode` or sometimes `egen`'s `group()` function for identifiers or categorical variables; and
- `destring` for numeric content trapped in a string container.

What is key here is that each solution attacks a different and specific problem. We should underline that no one command or function is an all-purpose solution for values wrapped up in inappropriate variables.

Specifically, neither `encode` nor `destring` knows anything about dates. The default mapping used by `encode` is quite wrong for dates and will typically not even leave them in the right order. Stripping dates of slashes or text elements with `destring` will make them harder to use, not easier. Stata can struggle with run-together dates such as 201810 for October 2018, although code can be suggested (Cox 2018).

To spell this out, and as a check of your understanding, let's consider a toy example where neither `destring` nor `encode` is the right answer.

```
. clear

. input str10 mydate

          mydate
  1. "23/12/1960"
  2. "12/09/2012"
  3. end
```

An intelligent child could tell you what these data mean. Alas, Stata in many ways lacks the insight of an intelligent child. If you `destring` this variable by stripping out the slash characters, you have not solved the problem. You have made it worse, as even converting it back to string and pushing the result through `daily()` shows:

```
. display daily(string(23121960), "DMY")
.
```

You may be able to think of ways of splitting numbers like 23121960 into their component parts, but none beats the solution, clear from 2.2 above, of feeding the original string to `daily()`.

Similarly, pushing these data through `encode` will map `"23/12/1960"` to 2 and `"12/09/2012"` to 1—because that is how the strings sort robotically: the first characters, `"1"` and `"2"`, resolve the sort order. The original strings will be copied to value labels. This false solution is insidious: you know you have produced a numeric variable, and in the Data Editor and in listings, it will look fine because you will see the original dates as value labels. However, almost always, the results will be pure garbage.

You might entertain yourself briefly thinking up exceptions: a short run of dates from `"1/10/2018"` to `"9/10/2018"` would be mapped to 1 to 9: hence, so far, so good. Yet adding just `"30/9/2018"` or `"10/10/2018"` is quite sufficient to mess up the correspondence.

The problem extends beyond dates. It should now be clear that pure numbers should not be `encode`d but pushed through `destring`. Thus, again, strings `"100"`, `"30"`, and `"2"` will not even be `encode`d in the right order. Try that if you do not see why.

The art of mapping from string to numeric is thus just one of keeping straight the right solution for each quite different problem.

4 Acknowledgments

Problems raised by many members of the community, particularly on Statalist, have helped extend our awareness of the many forms in which data may arrive and may require surgery or engineering before they are suitable for analysis.

5 References

Cox, N. J. 2007. Stata tip 52: Generating composite categorical variables. *Stata Journal* 7: 582–583.

———. 2018. Stata tip 130: 106610 and all that: Date variables that need to be fixed. *Stata Journal* 18: 755–757.

Schechter, C. 2011. Stata tip 99: Taking extra care with encode. *Stata Journal* 11: 321–322.

About the authors

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 coauthored 16 commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*. His "Speaking Stata" articles on graphics from 2004 to 2013 have been collected as *Speaking Stata Graphics* (2014, College Station, TX: Stata Press).

Clyde Schechter is a professor of family and social medicine at the Albert Einstein College of Medicine, where he works as an epidemiologist and focuses on clinical and health services research projects involving many specialties and disciplines. He has used Stata since version 4, is an active participant in the Statalist forum, and is an occasional contributor to the *Stata Journal*.