The Stata Journal (2002)
2, Number 3, pp. 314–329

Speaking Stata: On numbers and strings

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

Abstract. The great divide among data types in Stata is between numeric and string variables. Most of the time, which kind you want to use for particular variables is clear and unproblematic, but surprisingly often, users face difficulties in making the right decision or need to convert variables from one kind to another. The main problems that may arise and their possible solutions are surveyed with reference both to official Stata and to user-written programs.

Keywords: pr0006, binary variables, categorical variables, Data Editor, dates, decode, destring, encode, identifiers, missing values, numeric variables, spreadsheets, string functions, string variables, tostring, value labels

1 The great divide

Stata datasets are composed of variables, and those variables may take one or more different types. What is more fundamental, however, is a division between two different kinds. Numeric variables may be `byte`, `int`, `long`, `float`, or `double`, depending on the number of bytes used for storage and whether values are held as integers or as real numbers with fractional parts, or more precisely, as the binary equivalents or approximations to these. String variables hold character strings, those characters being broadly alphabetic (such as `"a"` or `"A"`), numeric (such as `"3"` or `"7"`), or other (chiefly various punctuation characters such as commas `","` or (a very important example) spaces `" "`).

As a small matter of history, Stata's variable types were not all available from the beginning. Stata 1.0 was released in January 1985. Originally, variables could only be numeric, although value labels were possible. String variables were introduced in Stata 2.0 in June 1988, and they could hold any even number of characters from 2 (`str2` type) to 80 (`str80` type). Thus, a single character would need to be held in a `str2` variable. Odd numbers of characters in string variables were soon allowed in Stata 2.1, which was released in September 1990. At the same time, the `byte` type was introduced to join the existing numeric types of `int`, `long`, `float`, and `double`. The next major change was not until February 2002, when Stata/SE, a between-releases "Special Edition", raised the limit for string variables to 244 characters.

Distinctions between different string types and different numeric types can be crucial, depending on the trade-off between holding all detail accurately enough and not using so much filespace or memory that operations become inordinately slow (or in extreme circumstances, even impossible). It is wasteful of memory, and, even if available memory is not limiting, it is wasteful in processing time to hold variables in types larger than is necessary. Conversely, the opposite pitfall should be clear: you can truncate or otherwise lose information with a variable type that is too small, or more generally, inappropriate.

You cannot fit a value such as "New York" in a str7 variable, and you cannot fit 3.14159 in an int, as that dish is not right for holding that pie.

Three commands help in managing allocation of data types. For more information, see their manual entries in the *Stata Reference Manuals*; noting that replace is documented in [R] **generate**. compress is an easy and safe way of reducing memory use. It often produces much smaller datasets, yet never loses information. replace is generally smart in promoting variables whenever that is needed. A variable that starts out in life as an int containing smallish integers will be changed to a float if calculations in a replace introduce fractional parts. A string variable will be promoted whenever it gets too long as the result of a replace, subject to the upper limit associated with your executable (in Stata/SE 7.0, 244 characters and in Intercooled Stata 7.0 and Small Stata 7.0, 80 characters). recast is occasionally useful for upward promotions (and dangerously, when desired, demotions using its force option).

These commands provide ways of achieving changes of type, from one numeric type to another or from one string type to another, but none allows change across the great divide, from numeric to string or vice versa. Crossing this divide is our main concern in this article. Not surprisingly, the main solutions occur in pairs, depending on the direction of crossing, which helps to make sense of the variety of methods that exist.

Useful background for this article is given in various chapters of the *Stata User's Guide*, especially [U] **15 Data**, [U] **16 Functions and expressions**, [U] **26 Commands for dealing with strings**, and [U] **27 Commands for dealing with dates**.

2 Numbers or strings?

2.1 Introduction

If you want to do numeric calculations in Stata with a variable, it should be numeric. Normally that would be your natural choice, and normally only by mistake will variables that should be numeric start out as string. (Such "mistakes" are more common than you might think, as you may already know; more will be said later, especially in Section 5.) Despite its many uses for managing many kinds of datasets, even databases, Stata is primarily statistical software, and so almost all users deal mostly with numeric variables.

However, the opposite rule—if you have information in the form of nonnumeric characters, it should be held as string variables—does not always apply. Another way to hold the nonnumeric information is as value labels linked to integer values of a numeric variable.

2.2 Choosing the right kind

What influences or determines the choice?

1. If the nonnumeric information you have defines categorical variables, especially if the number of categories is typically much fewer than the number of observations, it is best to use value labels. This is more efficient for storage and improves use of memory and processing time. In addition, some commands may not be applied to string variables, even when that might appear sensible; for example, you cannot use `graph` on string variables to get something like a histogram. In cases like this, the error message is typically `no observations`, meaning precisely that no numeric values are specified. To get such a graph, you need to produce the equivalent numeric variable with the nonnumeric values as value labels attached to integers.

2. If anything, the case for doing this is even stronger with binary variables, which you may also know as Boolean, dichotomous, dummy, indicator, logical, or quantal, depending on your upbringing. Whatever the name, they take on just two values, other than missing. As is well-known, a binary variable, especially one assigned values 0 and 1, may be used in many analyses: its mean has a direct and simple interpretation as a proportion, it may be used as a response variable in logit and probit models, and so on. All this, however, depends on a binary variable being held as numeric. (In passing, I commend the practice of naming a 0-1 binary variable for the category coded 1. If `sex` is 1 for female and 0 for male, then `female` is a better name. Which way round the coding is becomes more transparent.)

3. However, there are limits (in Stata/SE and Intercooled Stata 7.0, 65,536; in Small Stata 7.0, 1,000) on the number of value labels you may define that may be associated with a variable. If you have more than 65,536 categories, you are more likely to be dealing with individual identifiers than with classes of a categorical variable. An identifier is a name or a code that identifies people (places, businesses, whatever) uniquely, even if it is repeated in different observations, as in a panel dataset. But, whatever your situation, the limit in the version of Stata you are using is an unbreakable law, and so beyond that limit, nonnumeric identifiers must be held as string variables.

4. Apart from the question of limits, unique identifiers will often conveniently be held in string variables. There is little point in defining a value label if that value label occurs only once. It is also less likely that you would want to use such a variable as defining one axis of a graph.

5. Less obviously, identifiers that consist entirely of numeric codes are often better held as string variables. U.S. Social Security Numbers (SSNs) are one of the most frequently discussed examples on Statalist. They consist of nine digits, commonly written as three fields separated by hyphens with the form *aaa-gg-ssss*; these fields are in turn the area number, the group number, and the serial number. When stored without hyphens, these SSNs can be read into Stata as numeric variables, but small problems often arise later. More generally, to hold multi-digit identifiers without numeric precision problems (that is, holding every digit exactly) may require the use of a `long` variable. To display such a variable (as with `list`) may require changing format to avoid most digits being lost whenever identifiers

are presented in scientific notation. (See [R] **format**.) For example, a `float` numeric variable set equal to 123456789 will by default be `list`ed as `1.23e+08`, shorthand for 1.23×10^8. These are small and soluble problems, but they often cause puzzlement to Stata users. Holding such identifiers as strings, even though every character is numeric, solves those problems, with no apparent downside.

6. Similarly, categorical codes that are multi-digit numbers are often constructed hierarchically. That is, successive digits take you to finer detail within some classification system with numerical codes (of diseases or occupations or products, say). When such codes are held as numbers, working from fine to coarse categories, or vice versa, can be done by tricks with `int()` or `mod()`. These tricks strike many users as neat when they are familiar, but indirect or obscure when they are not. The corresponding operations on such codes held as strings can be done with `substr()` or `index()`, and these operations are often more transparent: the first three digits of a string variable `code` are specified by `substr(code,1,3)`, for example.

   Stata 7.0 includes a special command, `icd9`, for dealing with ICD-9 or ICD-9-CM codes from the International Classification of Diseases, Ninth Revision or the International Classification of Diseases, Ninth Revision, Clinical Modification. See [R] **icd9**. In Stata, such codes must be held as string variables, because some codes contain nonnumeric characters; even if that were not true, it would be better to hold them as strings for ease of data processing.

7. Another important class of data is dates, especially daily dates, which come in a variety of formats. A date recorded as (say) `"21 January 1952"` or `"1/21/1952"` can be interpreted as numeric once there is some agreement on what is day 0: Stata's convention is 1 January 1960, so 21 January 1952 is −2902. From that representation, we can easily find out, say, that 28 March 1952 was 67 days later, and so answer that and many more interesting or useful questions. However, variables with values like this can be read into Stata only as string variables: they can be converted later to numeric dates, usually with the `date()` function. (For more details, see [U] **27 Commands for dealing with dates**.)

   What might be called run-together dates with all digits numeric, say, `19520121` or `1211952`, can be read in either as numeric or as string. As already mentioned, a string representation has the advantage that precision and format problems are less likely, so long as the string type is large enough. For all that, you are most likely to want to convert dates in one of these representations to Stata dates. Such conversion is at root a matter of separating the components of the run-together dates and then combining them once more using Stata functions for both. At present, official Stata lacks facilities for handling run-together dates as such, but a user-written program may be accessed in an up-to-date Stata[1] by typing `ssc describe todate`.

[1] `ssc` was added to Stata 7.0 on 14 November 2001. If necessary, `update` your Stata (see McDowell 2001). If you do not have Stata 7.0, but you do have Internet access, *http://www.stata.com/support/faqs/res/findit.html* provides some explanation.

A run-together date such as `19520121` needs to be split into year, month, and day variables and then put together using the function `mdy()`. The splitting is at most three commands, which, if the date is held as string, all use the function `substr()`. What `todate` does is wrap this all into one, using a `pattern()` option to specify what the digits mean, so that

```
. todate date, gen(Date) pattern(yyyymmdd) format(%d)
```

performs the whole conversion to a Stata daily date variable. Half-years, quarters, months, and weeks are also supported, as are incompletely specified centuries and multiple variables.

8. Evidently, any nonnumeric characters in dates, such as slashes or month names, are sufficient to imply that in the first instance the corresponding variables should be held as string. More generally, any nonnumeric characters in otherwise numeric fields are sufficient to make direct numeric representation impossible. So, for example, a variable containing age intervals such as `0-9` or `10-19` must be held as string or as numeric with value labels because of the included hyphens. That need not stop approximate numeric equivalents from being computed, say, containing the midpoint of each interval, or the endpoints being pried apart and put into separate and equivalent numeric variables.

9. If you are typing in the data, either by yourself or with assistance, it may be much less labor to enter data as numeric. Value labels may be specified either before or later, and need only be typed once. (Sometimes, the associated value labels may already be defined, or the same value labels may be associated with several variables, as when a group of variables all record yes or no answers.) Saving of effort is greatest to the extent that small integers can be entered rather than longer strings.

10. A quite different consideration arises sometimes, especially when teaching. One elementary error is to forget, particularly for statistical rather than data management commands, that a variable may be numeric to Stata without being a variable which may fairly be included in a statistical model as is. It is arguable that a habit of holding arbitrary numeric codes as strings provides some protection against this blunder, helping you to avoid foolish statistics. In practice, however, this will rarely be a deciding issue.

2.3 Missing values

A reminder of what counts as missing with numeric and string variables may be helpful here.

Numeric missing in Stata is represented by a period `.`, which is always treated as larger than any other numeric value. Thus, a `sort` of a numeric variable sorts missing values to the end of a dataset.

String missing in Stata almost always means an empty or null string, `""`. An empty string contains nothing (or does not contain anything, depending on your metaphysical predilections), whatever the type of string variable concerned. A `sort` of a string variable sorts empty strings to the beginning of a dataset.

No special meaning is given by Stata to strings consisting of one or more spaces. If you want such strings to be treated as missing, consider using `replace` with the `trim()` function to reduce them to empty strings.

Occasionally, some commands treat `"."`, or even that together with any leading or trailing spaces, as indicating missing. This is anomalous and deserves brief comment.

`destring` is a case in point, and in this instance, the anomaly can be justified. As will be discussed in detail in Section 5, `destring` is for situations in which a variable should be numeric, but is by mistake string. For example, suppose you typed a column of data into Stata's Data Editor, but by mistake typed a nonnumeric character in the value for the first observation. (You may have been thinking of a header line, spreadsheet style.) Thereafter, you typed numeric characters, including `.` for missing values. The result of all this is that Stata interprets the column as a string variable, but that is almost certainly wrong. `destring` feels free to interpret the string `"."`, or any string that `trim()` reduces to `"."`, as really numeric missing. It is, not surprisingly, much more circumspect about other nonnumeric characters.

Another exception is `compare`, introduced into official Stata 3.0 in March 1992. As explained in [R] **compare**, this command, in deference to some users' habits, understands both `"."` as well as empty strings as indicating string missing. With perfect hindsight, this broad-mindedness was perhaps a mistake, but it does very little harm and is better left unchanged, just in case a change breaks someone's long-standing do files or programs.

Finally, note that the `string()` function, discussed in more detail in Section 4, yields `"."`, not `""`, as the string equivalent of numeric missing.

3 encode, decode, and their limitations

3.1 encode and decode: having it both ways

The first kind of conversion operations that we will look at are mappings back and forth between string variables and numeric variables with value labels. `encode` produces a new variable that is numeric with value labels from a string variable. `decode` produces a new variable that is string from a numeric variable with value labels. See [R] **encode**. Both commands generate new variables; the variable you start with remains in memory unless you later `drop` it yourself. `encode` and `decode`, therefore, leave you with the same information in a different form, for maximum flexibility. `encode` and `decode` were introduced with string variables in Stata 2.0, and they remain basic Stata commands. `decode` is perhaps less frequently used, but no matter: whenever an operation and its inverse both make sense, there is an excellent case for including both in Stata. As

explained in [R] **encode**, `decode` can be very useful for match merging two datasets on a variable that has been `encode`d inconsistently.

Both commands work only on one variable at a time. Use of either on several variables might be accomplished best with the help of `foreach`, as explained in my previous column (Cox 2002a). Suppose we want to `encode` variables `a`, `b`, `c`, `d`, and `e`. Here is a simple way of doing that, yielding `Na`, `Nb`, `Nc`, `Nd`, and `Ne`:

```
. foreach v in a b c d e {
.         encode `v', gen(N`v')
. }
```

3.2 Alphanumeric ordering and how to avoid it

One important detail is that by default, `encode` uses alphanumeric (i.e., ASCII) order of string values to determine the order of integer codes. In the absence of any other information, this is surely the best systematic way of carrying out the mapping. There are downstream consequences, however: the alphanumerically ordered value labels are usually uppermost in tables, on graphs, and in other output, and this order remains wired into the association between integers and value labels. It is often, indeed, the order that you want. Grades `"A"` `"B"` `"C"` `"D"` `"E"` are mapped to 1 2 3 4 5 and tabulated in that order, which will often be fine (although also computing (6 − grade) might be useful for calculations). But, perhaps equally often, an alphanumeric order is a nuisance, because it fails to respect some inherent meaning. String values `"excellent"`, `"good"`, `"fair"`, `"poor"`, and `"bad"` belong in that sequence, and not sorted into `"bad"`, `"excellent"`, `"fair"`, and so forth.

The most general solution to the problem of arbitrary alphanumeric order is to write the order you want into a set of value labels defined before the `encode`, and then to specify that `encode` uses those mappings. Thus,

```
. label def opinion 1 "excellent" 2 "good" 3 "fair" 4 "poor" 5 "bad"
. encode answer, gen(opinion) label(opinion)
```

obliges Stata to ignore its default ordering and to follow the one you prefer.

Alphanumeric ordering may bite in a manner that is puzzling until you recall precisely what such ordering means. Thus, age categories in ten-year intervals `"0-9"` `"10-19"` ... `"90-99"` will remain in this evidently natural order on alphanumeric ordering. But split `"0-9"` into `"0-4"` and `"5-9"`, and add `"100-109"`, and you will find that `"5-9"` will be sorted after `"40-49"` and that `"100-109"` will be sorted before `"20-29"`. This is easy to explain. Although the ten numeric digits—in numeric order 0 through 9—have the same alphanumeric order when represented as numeric characters `"0"` through `"9"`, all strings when sorted are ordered as in a dictionary, so that ties are broken on successive characters without any reference to their meaning. Once more, the best solution to avoid an unsatisfactory result from `encode` is to use a predefined set of value labels codifying your desired order.

3.3 Start 1 encoding and how to avoid it

`encode` by default starts integer coding at 1. Sometimes you would prefer 0 as a start, especially for binary variables. One way to achieve this is by specifying the desired value labels ahead of the `encode`, as in the previous examples.

Retrospectively, fixing the values of a start 1 variable to start at 0 by subtracting 1 is easy enough. The fiddly part is fixing the value labels to match. If you need to do this a lot, two possible user-written tools are `labedit` (Gleason 1998, 1999) and `labvalch` (type `ssc desc labutil`).

Concretely, suppose you had a variable `female` coded as 1 and 2, and you decide that this would be better coded as 0 and 1. You could just

```
. replace female = female - 1
```

or, if you are familiar with `recode` (see [R] **recode**), you might prefer

```
. recode female 1=0 2=1
```

but neither does anything to any value labels attached to `female`. If they also need fixing, the slow but sure way is just (supposing they have the same name as the variable)

```
. label define female 0 "male" 1 "female" 2 "" , modify
```

noting the detail of deleting the label for 2 by setting it to an empty string. The way to do this with `labvalch` is

```
. labvalch female, from(1 2) to(0 1) delete(2)
```

which for just one variable is no gain. But for several sets of value labels, all to be shifted from start 1 to start 0, there could be a payoff:

```
. foreach lbl in female married employed retired yesno {
.         labvalch `lbl', from(1 2) to(0 1) delete(2)
. }
```

3.4 Ordering classes according to another variable

Defining a choice of labels by explicitly typing a `label define` before an `encode` is not ideal for all situations. One common problem is that we want an ordering of classes according to their values on some other variable, so that a table or a graph will then show a more intelligible pattern, cutting free of the arbitrariness of the alphabet. Some Stata programs do this for you on the fly, but we still need to be able to do it for ourselves whenever no such program exists for a particular task. This problem can arise for any categorical variable, whether represented by a string variable or by a numeric variable with value labels.

Concretely, we might want categories ordered according to their associated frequency, or on the mean or maximum of some other variable. Again, we could compute the ordering criterion directly and use the results to identify an appropriate set of value labels, which we then type out in a `label define`, but let us see if we can automate most of that. In this kind of problem, the devices used most often vary from solutions from first principles using `by:` (Cox 2002b) to canned `egen` functions (see [R] **egen**). (The fact that `egen` includes several functions applicable to string variables is often overlooked.)

With the auto data, let us define the manufacturer name as the first word of `make`, and then count values by manufacturer:

```
. egen manuf = ends(make), head
. bysort manuf : gen freq = -_N
```

Note the minus sign. We are looking ahead, and foreseeing a `sort`, and, in particular, foreseeing that we will want highest values first, so that they will appear (say) as the first rows of a table or the left-hand bars of a bar chart. That order is thus the opposite of Stata's default sort order in which lowest values come first; negating values will reverse the order for us. (Naturally, if the default sort order is what you want, you should omit the minus sign.)

Now we want to put the categories of `manuf` in the order defined by `freq`. Another `egen` function is very useful here.

```
. egen group = egroup(freq manuf), label(manuf)
```

`egroup()` produces the distinct categories defined by its arguments, in their (joint) `sort` order. The results are successive integers from 1 up, guaranteed (for example) to yield tidy graphs. `egroup()` here has two arguments. `freq` is mentioned first, because our main ordering is to be by frequency. `manuf` is mentioned next to ensure that any ties on `freq` are broken properly (for example, there are 7 Buicks and 7 Olds: 14 values thus have `freq` equal to 7, which must be split into those two groups).

`egroup()` is a user-written extension of official Stata's `egen, group()`. (To access, type `ssc desc egenmore`.) The extension itself is just one extra wrinkle: the option `label` may take an argument, *lblvarlist*, which specifies that we want the new variable to have value labels that are the values (or the value labels if they exist) of *lblvarlist*. By contrast, the `label` option of `group()`, which takes no argument, uses all the variables in the *varlist* supplied to `group()` to construct value labels. With these data, we would have labels like `"-7 Buick"`, the `-7` coming from the frequency, negated. Clearly, we want only the manufacturer names in the labels.

(*Continued on next page*)

```
. tab group
```

group(manuf)	Freq.	Percent	Cum.
Buick	7	9.46	9.46
Olds	7	9.46	18.92
Chev.	6	8.11	27.03
Merc.	6	8.11	35.14
Pont.	6	8.11	43.24
Plym.	5	6.76	50.00
Datsun	4	5.41	55.41
Dodge	4	5.41	60.81
VW	4	5.41	66.22
AMC	3	4.05	70.27
Cad.	3	4.05	74.32
Linc.	3	4.05	78.38
Toyota	3	4.05	82.43
Audi	2	2.70	85.14
Ford	2	2.70	87.84
Honda	2	2.70	90.54
BMW	1	1.35	91.89
Fiat	1	1.35	93.24
Mazda	1	1.35	94.59
Peugeot	1	1.35	95.95
Renault	1	1.35	97.30
Subaru	1	1.35	98.65
Volvo	1	1.35	100.00
Total	74	100.00	

There are two (and perhaps a half) basic steps here, but the method is quite general. First, generate the variable on which you want to order classes as a group statistic. Remember negation if you want to reverse the order. Second, use `egen, egroup() label()` to produce the groups in the right order and with the right labels. These steps produce a variable that may then be used for tables, graphs, etc.

4 real() and string(): the brute force solutions

There are brute force methods of crossing the divide. The function `real()` extracts the numeric content of a string expression, such as a string variable. The function `string()` converts a numeric expression, such as a numeric variable, to a string. The documentation for some software talks of coercion from one type to another. Although that is not a common term in discussing Stata functions, it captures well the flavor of what is discussed in this section.

Given any doubt, `real()` yields numeric missing. Therefore, watching carefully for messages about missing values is always worthwhile. Having a look at the problem observations may reveal some problems with easy fixes, or at least a variable better tackled with `destring` (see Section 5). For example, `real("1,234")` yields numeric missing. The intelligence that strips the comma is not built in to the function. More generally, given string variable *strvar*, which contains mostly numeric information, these inspections should pinpoint what cannot be interpreted as numeric:

```
. tabulate  strvar if real( strvar) == .
. list  strvar if real( strvar) == .
```

The brute force may seem to lie on one side. Although not all characters are numeric, surely all numeric characters are characters. Nevertheless, the companion function `string()` is also fairly described as a brute force solution, for the same fundamental reason: without some forethought, it is possible to miss information in your data. Nine-digit identifiers such as U.S. Social Security Numbers provide a simple but forceful example. `string(123456789)` yields `"1.23e+08"`, meaning 1.23×10^8. The key detail needed is that `string()` is really a function taking two arguments, the second of which is a numeric or date or time format. `string(123456789, "%12.0g")` yields `"123456789"` in the way that you would expect. The point is simply that the default format is not, and cannot be, ideal for all circumstances.

For simple cases, `real()` and `string()` may perform perfectly well, but beware their brutality when data are more complicated than you realized.

5 destring and tostring: correcting mistaken choices

5.1 Learning from history

As we have seen, the main solutions to problems with the great divide occur in pairs, a pair for each operation and its inverse. The last pair we will look at is the official Stata command `destring` and its sibling, the user-written command `tostring`. `destring` has the longer history. After publication and revision in the *Stata Technical Bulletin* (STB) (Cox and Gould 1997; Cox 1999a, 1999b), it was adopted in official Stata in version 7. `tostring` came later (Cox and Wernow 2000a, 2000b).

The history of `destring` has no practical implications, except for any users who started with the STB version and became accustomed to the ways in which it worked, but it raises major issues to do with command design. When `destring` was incorporated in official Stata, it was largely rewritten to produce a more focused command. Why was that?

First, the original `destring` command violated Stata's philosophy in that it was too easy to change much of your dataset without the safeguard of having to spell out some injunction such as "`, replace`". `destring` now not only insists on that, it also obliges you to specify `force` whenever the creation of a numeric variable would lose information contained in the original string variable.

More specifically, `destring` allowed a variable to be replaced by an `encode`d equivalent, unless users specifically opted out of that. Positively, it offered the power to solve several dataset problems with no more than a single `destring` (no variable list, no options). Existing numeric variables would be unchanged, string variables with purely numeric content would be replaced by their numeric equivalents, and other string variables would be `encode`d if possible. But, by the same token, this made it too easy to misunderstand what had been done, with all sorts of further consequences. If `x` were

string with numeric content, then `summarize x` is a perfectly reasonable thing to type after `destring`. If `x` were string with no numeric content, and as such was `encode`d, then `summarize x` is all too likely to be wrong or meaningless. If the price of liberty is eternal vigilance, then in the case of the original `destring`, it was bought too dearly.

What underlies all this is a general principle: it is best when Stata commands do one thing well. In retrospect, it was a bad idea to let `destring` overlap in functionality with `encode`. On adoption in Stata 7.0, therefore, it was decided to sharpen the distinction between them. `encode` is designed for situations in which you have a string variable containing nonnumeric text (e.g., `"male"`, `"female"`), and wish to have the equivalent information as a numeric variable with labels. `destring` is designed for situations in which you have a string variable containing numeric text (e.g., `"1"`, `"2"`), which you wish to convert to the numeric variable that it should properly be. Usually, that variable is now string because of some mistake. How could such a mistake be made?

5.2 Mistaken string variables

Stata's Data Editor

One way in which a mistake can arise is when entering data into Stata's Data Editor (see [R] **edit**). Some users, especially if new to Stata and accustomed to the license provided by spreadsheets, may type header information, say, a column title or explanatory text, into the top rows of the editor. To Stata, however, these rows define the first few values of its variables; its rule is that all cells in any given column must be of the same variable type.

Just the first observation can be crucial in interacting with the Data Editor, which attempts to be as smart as possible and to divine your intentions from what you type. Enter any nonnumeric text in the first cell of a column, and the editor instantly thinks, "Aha! This user wants this variable to be a string variable", and it promptly and silently creates that variable as string. Even if all cells below are entered with numeric text, this causes `edit` no puzzlement, as numeric characters are perfectly acceptable in a string variable. The editor will silently promote a string variable to a wider type if later values require that. However, the editor will never unilaterally change a variable of yours from string to numeric, or from numeric to string, and the same holds for all other Stata commands. To do that would entail guessing at what you really intend, with the possibility of making an incorrect guess and making a mess of your dataset. To do that could seriously compromise the integrity of your data.

This is a strong principle, which can admit no exceptions, even though the consequences may be irritating. Suppose, for example, that the mistake lay only in a first row of text values across a dataset, defining the first observation. Realizing your mistake, you simply delete the first row. Now, perhaps most—or even all—of the columns in the editor contain purely numeric text, for the simple reason that they were all intended to contain numeric variables. Should the editor change its mind, and change these variables unilaterally to numeric? No. For all the editor knows, you really wanted the

numeric text to be a string variable. After all, Section 2 gave several good reasons for storing numeric text in a string variable. More generally, Stata should not be in the business of making guesses about your data, or of assuming that it knows better than you do precisely what you intend your data to be. It follows that you must specify your intentions explicitly, using `destring`.

Naturally, `edit`'s principle that first impressions count is not the only basis on which it could have been designed. However, a data editor in which you were obliged to specify a variable type before a variable was entered would not have the friendliness and freedom of action that one might fairly expect.

Spreadsheets and other software

The same kind of issue can arise when importing data from spreadsheets, from other application programs (say, under Windows, by way of the clipboard), or from ASCII files created by these programs, by text editors, or by word processors. Spreadsheets provide good examples of some of the possible difficulties that lead to mistaken string variables. The same or similar problems can also occur with other programs or their creations, which might be read in with a command like `insheet`. (See [R] **insheet**.)

Some of these problems depend on which software you are using. A list of various pitfalls that have been reported thus far may, with good fortune, include some that will never open up before you.

1. A single cell in a column containing a nonnumeric character, such as a letter, is enough for Stata to treat that column as a string variable. As implemented in Stata 7.0, `destring` includes an option for stripping commas, dollar signs, percent signs, and other nonnumeric characters. It also allows automatic conversion of percent data.

2. What appear to be purely numeric data in a spreadsheet are often treated by Stata as string variables because they include spaces. You, or whoever created your data file, may inadvertently enter space characters in cells that are otherwise empty. Although a spreadsheet may strip leading and trailing spaces from numeric entries, it will not trim spaces from character entries. One or more space characters by themselves constitute a valid character entry and are stored as such. Stata may dutifully read the entire column as a string variable. If so, you could delete such stray spaces in the spreadsheet, or you can use a text editor or scripting language on an exported text file, or again you could use `destring`.

3. Much formatting within a spreadsheet interferes with Stata's ability to interpret the data reasonably. Just before saving the data as a text file, make sure all formatting is turned off, at least temporarily.

4. With integer-like codes such as ICD-9 codes or U.S. Social Security Numbers that do not contain a dash, leading zeros will get dropped when you paste into Stata. One solution is to flag within the first line that the variable is string: add a

nonnumeric character in your spreadsheet on that line, and then remove it in Stata. Alternatively, the missing leading zeros can be replaced in Stata in a conversion to string:

```
. gen str12  strvar = string( numvar,"%012.0f")
```

The second argument on the right-hand side of this command is a format specifying display of leading zeros in conversion of *numvar* to its string equivalent. See [R] **format**.

5.3 Mistaken numeric variables

`destring` has a sibling, `tostring`, for the less common situation in which you decide that the information held in numeric variables really should be held in string form. As discussed in Section 2, this need arises most commonly when the numeric variable is really an identifier of some kind, or a complicated categorical code, and you wish to carry out string manipulations using Stata's string functions.

`tostring` is essentially a convenience command, with various features, including automatic determination of which string type is needed; a default format (`%12.0g`) better suited to integers, whether held as `byte`, `int` or `long`; and the ability to work with several variables at once. Conversely, some features are, in retrospect, bad, or at least arguable, choices, and will be withdrawn in future versions: being able to `decode`; being able to overwrite existing variables without specifying "`, replace`"; and being able to ignore details (especially in fractional parts) without specifying a `force` option.

6 Statistically defined variable types?

It will be clear by now that the distinctions between Stata's variable types are based mainly on computing distinctions, rather than any statistical or other distinctions based on the scales of measurement used, the roles played by variables in statistical models, or similar considerations. In particular, categorical variables may be held in quite different ways within Stata. Stata does show a marked preference that they be held in numeric variables with value labels attached, insofar as that allows the widest possible range of commands to be used, but for some purposes, it is also a good idea to hold the same information in string form.

Concretely, Stata has (for example) no exact equivalent to the ideas of factors or to ordered factors available in S, S-Plus, or R (see, for example, Chambers and Hastie 1992). What may seem at first an exception to this assertion, the `category()` and `continuous()` options of `anova` (explained in [R] **anova**), is in fact a demonstration of it; that is, `anova` requires a specification of how it is to treat variables. It has no way of knowing from the information stored by Stata how variables are to be interpreted. Also, this specification is purely for the moment, and has no consequences for any future commands, except those that process output from the model just fitted. On occasion, when the best way to treat some variable (say age) is in doubt, `anova` commands could

even treat a variable as in turn categorical and continuous, in a bid to identify the better representation. What Stata gains and what it loses from this way of treating variables makes a topic for interesting future discussions.

7 Summary

The variety of routes that cross the great divide between numeric types and string types in Stata may be reduced to intelligible order, despite some mistaken ventures by pioneers. Use `decode` or `encode` to generate new variables of different kinds holding the same information. Use `real()` and `string()` for conversions or coercions, checking each time on whatever cannot be coerced to different kind, and is thus assigned missing. Use `destring` or `tostring` to correct mistakes whenever you have variables of one kind that really should be of the other kind.

8 What's next?

In the next column, we will look at the confusing territory stretching from built-in functions to `egen` functions. In engineering terms, the range is from 'black boxes' (you cannot look inside) to 'white boxes' (you can look inside and borrow ideas). Even with the large and growing number of functions that they provide, there will always be gaps, so the key question of how best to fill those gaps for yourself will also be discussed.

9 Acknowledgments

Some of the work reported here benefited greatly from the collaboration and comments of William Gould, Jeremy Wernow, and Vince Wiggins. Material on spreadsheets is based partly on an FAQ to which contributions were made by Ted Anagnoson, Dan Chandler, Ronan Conroy, James Hardin, David Moore, Paul Wicks, and Eric Wruck.

10 References

Chambers, J. M. and T. J. Hastie. 1992. *Statistical Models in S*. Pacific Grove, CA: Wadsworth and Brooks/Cole.

Cox, N. J. 1999a. dm45.1: Changing string variables to numeric: update. *Stata Technical Bulletin* 49: 2. In *Stata Technical Bulletin Reprints*, vol. 9, 14. College Station, TX: Stata Press.

—. 1999b. dm45.2: Changing string variables to numeric: update. *Stata Technical Bulletin* 52: 2. In *Stata Technical Bulletin Reprints*, vol. 9, 14. College Station, TX: Stata Press.

—. 2002a. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2(2): 202–222.

—. 2002b. Speaking Stata: How to move step by: step. *Stata Journal* 2(1): 86–102.

Cox, N. J. and W. W. Gould. 1997. dm45: Changing string variables to numeric. *Stata Technical Bulletin* 37: 4–6. In *Stata Technical Bulletin Reprints*, vol. 7, 34–37. College Station, TX: Stata Press.

Cox, N. J. and J. B. Wernow. 2000a. dm80: Changing numeric variables to string. *Stata Technical Bulletin* 56: 8–12. In *Stata Technical Bulletin Reprints*, vol. 10, 24–28. College Station, TX: Stata Press.

—. 2000b. dm80.1: Update to changing numeric variables to string. *Stata Technical Bulletin* 57: 2. In *Stata Technical Bulletin Reprints*, vol. 10, 28–29. College Station, TX: Stata Press.

Gleason, J. R. 1998. dm56: A labels editor for Windows and Macintosh. *Stata Technical Bulletin* 43: 3–6. In *Stata Technical Bulletin Reprints*, vol. 8, 5–10. College Station, TX: Stata Press.

—. 1999. dm56.1: Update to labedit. *Stata Technical Bulletin* 51: 2. In *Stata Technical Bulletin Reprints*, vol. 9, 15. College Station, TX: Stata Press.

McDowell, A. 2001. From the help desk. *Stata Journal* 1(1): 76–85.

About the Author

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