View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Research Papers in Economics

The Stata Journal (2003)
3, Number 1, pp. 81–99

Speaking Stata: On structure and shape: the case of multiple responses

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

Ulrich Kohler
Social Science Research Center
Berlin, Germany
kohler@wz-berlin.de

Abstract. A frequent problem in data management is that datasets may not arrive in the best structure for many analyses, so that it may be necessary to restructure the data in some way. The particular case of multiple response data is discussed at length, with special attention to different possible structures; the generation of new variables holding the data in different form; valuable inbuilt string and `egen` functions; using `foreach` and `forvalues` to loop over lists; and the use of the `reshape` command. Tabulations and graphics for such data are also reviewed briefly.

Keywords: pr0008, composite variables, concatenation, egen, foreach, forvalues, graphics, indicator variables, multiple responses, reshape, split, string functions, tabulations

1 Introduction

A common source of problems in working with Stata, and indeed any statistical software, is that the data may not come in the most convenient structure for doing what you want to do. On occasion, restructuring the data may be as challenging as doing the more interesting statistical analysis. At worst, what could be more frustrating than having a clear research question, but not being able to see how to get an answer with your present data structure? In total, this is an enormous and multifaceted field that may be tackled with a variety of Stata commands. In this column, we will look at a particular problem, that of multiple response variables.

Multiple responses, in the sense used here, are defined by a degree of open-endedness. In particular, a question in a survey may receive zero or more positive answers depending on the characteristics or behavior of the respondent. For example, respondents might be asked: Have you experienced any of the following symptoms or received information on a subject from any of the following media? Do you ever drink tea, coffee, wine, beer, or water? Do you travel to work by foot, bicycle, motorcycle, car, bus, tram, train, boat, ski, skates, sledge, horse, camel, yak, . . . ? (That may seem a simple question to you, but consider commuters who cycle or drive to catch a train and then end their journey to work with a walk.) Note that we are not here discussing multivariate responses in general, nor repeated measures, nor panel data or longitudinal data, etc.

In statistical computing terms, such multiple responses may pose difficulties both for data structure and for data analysis. Most commonly, they are held as a set of variables, but sometimes it can be useful to hold them as a single variable. No structure is ideal for all purposes, and often you may want to convert from one structure to another. Similarly, you may want to look at results for individual variables, or at results calculated from one or more of these variables. Again, even a whole column cannot cover all possibilities.

Reference is made below to various user-written programs on SSC. Users of Stata 8 will find `ssc` incorporated as an official command and documented in [R] **ssc**. Users of Stata 7 will also have access to `ssc` as an official command if they have `update`d their Stata, but their documentation consists of the online help.

2 How data may be held

2.1 Indicator variables

Let us look first at a relatively simple example. This crucially important question might appear in a questionnaire:

Which of the following software packages do you use for data analysis?

1 R
2 S-Plus
3 SAS
4 SPSS
5 Stata
6 others

In this question, respondents are asked to mark the name of each package they use. Respondents may mark any number of packages. Note that the number before each package name is used as a code in some coding schemes discussed below.

For many statistical analyses, the answers of the respondents are best coded as a set of indicator or dummy variables:

```
          q1_R   q1_SPlus     q1_SAS    q1_SPSS   q1_Stata  q1_others
 1.          1          0          0          0          0          1
 2.          1          1          0          0          1          0
 3.          0          0          0          0          1          0
 4.          0          0          1          0          0          0
 5.          0          0          1          0          0          1
```

That is, there should be a variable for each possible answer, with value 1 if a respondent uses a specific package and 0 otherwise. The first respondent in this example uses R and some other package; the second respondent uses R, S-Plus, and Stata; and so on. We use names for the variables that have a common prefix. This is a small detail, but it makes it easier to refer to the variables collectively using a wildcard, such as `q1_*`.

Data on multiple responses with this coding scheme can be used immediately for many analyses. For example, you might want to know how many respondents use Stata. Type

```
. count if q1_Stata == 1
```

or type

```
. tabulate q1_Stata
```

You might want to see the distribution of the number of packages used by the respondents. This is just the row sum of the variables, most easily calculated by `egen`. See [R] **egen** for details on a variety of functions and a previous column (Cox 2002c) for a general discussion.

```
. egen npkg = rsum(q1_*)
. tabulate npkg
```

You might want to know the distribution of users of software packages. One method is to summarize the variables and compare their means, but a better method is through `tabstat`, say,

```
. tabstat q1_*, s(sum) c(s)
```

The use of row sums and of variable sums across 1s and 0s underlines the value of holding data in indicator variable form.

2.2 Ranked multiple responses

A common variant is that the question asks you to rank choices, say, from most common to least common use, or in some other way.

```
          q1_1      q1_2      q1_3      q1_4      q1_5      q1_6
  1.         1         6         0         0         0         0
  2.         5         2         1         0         0         0
  3.         5         0         0         0         0         0
  4.         3         0         0         0         0         0
  5.         6         3         0         0         0         0
```

Thus, using the coding scheme indicated previously, person 1 uses R most and some other package next. There is more information recorded in this variant form, as the first data structure can be obtained from this one, but not conversely. Two common variations on this scheme are to use numeric missing rather than 0 and to use string variables including names rather than numeric codes.

This structure evidently makes it easy to focus on which package is most commonly used, and so forth. It makes it difficult to focus on which packages are used at all, and so forth.

We mention here a possibility that researchers may encounter or produce tied ranks in some projects. How best to handle tied ranks is not considered here.

2.3 "Order of mention" multiple responses

Yet another situation is that answers have been coded in the order in which the respondent mentioned them. Such data may look like ranked multiple responses, but their interpretation may or may not be similar. In some fields, it appears common to take order in which responses were mentioned as tacit indication of an underlying order. For example, suppose that you were asked to state brands of some item that you purchase or that you know about. Marketing people could be very interested in what springs most readily to your mind. Whether "order of mention" is tantamount to ranking is a substantive matter for you to consider.

2.4 Composite string variables

Sometimes the answers to multiple responses are put into one string variable. Commonly, this is the concatenation of the codes of possible (positive) answers. For our example data, such a variable could look like

```
       spkg1
  1.      16
  2.     521
  3.       5
  4.       3
  5.      63
```

where we use numeric codes as above, or it could look like

```
           spkg2
  1.    R others
  2. Stata S-Plus R
  3.       Stata
  4.         SAS
  5.  SAS others
```

The variable `spkg1` states for the first observation that this respondent uses the software packages 1 and 6, which means R (1) and some other package (6). This may look like a numeric variable, but it should be a string variable. In our experience, both producing such a variable from other variables and working with such a variable are much easier when it is a string variable than when it is an integer-valued numeric variable. In any case, as soon as the number of possibilities exceeds 10, you will need to punctuate to avoid ambiguity. Otherwise, someone mentioning symptoms 1 and 3 from a list would be treated the same as someone mentioning symptom 13; both would be represented by `"13"`. Similarly, as in the example of `spkg2` just given, once nonnumeric characters are used, there is no reason not to include punctuation to make elements clearer, unless you are near the limiting size of string variables.

Note various issues that can arise in practice. If packages are being ranked, then `"Stata others"` has a different meaning from `"others Stata"`, but not otherwise. In particular, with unranked data, be warned: values that to you are identical but nevertheless differ literally will be tabulated or counted separately. Similar comments apply to leading and trailing spaces, accidental misspellings, or inconsistencies in uppercase and lowercase. In the latter situation, problems may be solved by working consistently, say, in lowercase with the aid of the `lower()` function; see [R] **functions** (or see [U] **16.3.5 String functions** for all previous releases).

This structure is useful particularly for showing combinations of choices, say, in tables of the composite variable. As the number of possible answers grows, the number of possible combinations also grows rapidly. Even setting aside the possibility of ranking, k choices mean 2^k possible combinations. However, this is a fact whatever the data structure.

A detail important here is whether the variable really is a string variable or (despite our general advice) a numeric variable. When tabulating a string variable, Stata will sort `"12"` before `"2"`; when tabulating a numeric variable, Stata will sort 2 before 12. Which convention is better for you will depend on your purpose. Thus, with a string representation all choices with `1` as first character will be tabulated adjacently, while with a numeric representation, all choices coded by a single digit will be tabulated adjacently. Either could be useful.

2.5 Single variable in a long data structure

Another data structure holds all information in a single variable with repeated observations for each individual in the dataset. An example might be something like

```
              id        q1
  1.           1         R
  2.           1    others
  3.           2         R
  4.           2    S-Plus
  5.           2     Stata
  6.           3     Stata
  7.           4       SAS
  8.           5       SAS
  9.           5     Stata
```

In the jargon associated especially with the `reshape` command, this is an example of a long data structure.

The answers, here in `q1`, could be held in a string variable or in a numeric variable with value labels attached. To make full use of the information in such data, an identifier variable, here `id`, is essential. (An identifier variable was not needed for any of our earlier examples, although we guess that in practice most researchers will have one in any case.) Note that there is no requirement to show zero or missing responses; that is, to make explicit the fact that the person with `id` 1 does not use programs other than those mentioned. Thus, this data structure is economical as a way of holding multiple response data, but it is correspondingly awkward as a way of holding other data on the same individuals. Suppose, for example, that we were also holding data on individuals' age, sex, and field of study. This information would be best held repeated for each observation, which is inefficient (but otherwise not especially problematic).

Data on multiple responses in this structure can be used immediately for many analyses. For example, you might want to know how many respondents use Stata. If `q1` is a string variable, type

```
. count if q1 == "Stata"
```

or if `q1` is a numeric variable in which Stata is represented by 5, type

```
. count if q1 == 5
```

Data in this structure may be used easily for analyses of subsets defined by separate answers, either a particular subset or several subsets. The information yielded by `count`, and more, is available from

```
. tabulate q1
```

which shows the distribution of users of software packages.

You might want to see the distribution of the number of packages used by the respondents. This is just the number of observations for each individual (distinct `id`) for which `q1` is not missing. If `q1` is never missing, this is yielded by

```
. bysort id : generate npkg = _N
```

Irrespective of whether `q1` is ever missing, this is yielded by

```
. bysort id : egen npkg = count(q1)
```

as `count()` counts how often its argument is not missing: see [R] **egen**.

However, if this were followed by

```
. tabulate npkg
```

the individual with `id` 1 would be shown twice, that with `id` 2 three times, and so on. We need a way of selecting each `id` just once. An `egen` function is dedicated to this task: `tag()`. This tags just one observation in each group of identical values with value 1 and any other observations in the same group with value 0.

```
. egen tag = tag(id)
. tabulate npkg if tag
```

Note that the idiom `if tag` as a contraction of `if tag == 1` is always safe, as `tag()` never produces missing values. This tagging method has many other uses whenever we wish to relate multiple response data to other data for each individual.

A final advantage of this structure is that it is also applicable to ranked multiple variables, given an extra variable holding ranks. It is then easy using (e.g.) `generate`, `egen`, `tabulate`, `by:`, `if`, etc. to produce many basic analyses.

Despite some major advantages, this data structure is awkward for working with conditions specifying more than one answer. There are some ways to approach this, but they are not very attractive. We could tag those who use both R and Stata in this way, illustrated by the case of string variables:

```
. bysort id : egen R_and_Stata = sum(q1 == "R" | q1 == "Stata")
. replace R_and_Stata = R_and_Stata == 2
```

One part of the argument to `sum()`, `q1 == "R"`, will pick up any observation for which that is true. The other part of the argument, `q1 == "Stata"`, will pick up any observation for that is true. The sum of a result of 1 if each condition is satisfied just once for an individual should be 2. Naturally, that sum is not affected by any number of results of 0 arising whenever any condition is false. However, although we could make some progress with such questions and this data structure, it turns out that other data structures are far superior whenever examining two or more answers simultaneously.

2.6 Missing values and the appropriate denominator

Missing values are likely to be common with multiple response data. Even if everybody answered the question—which is unusual in many surveys—it may not be the case that everybody gives the same number of responses. Even when asked to rank a fixed number of specified items, respondents often stop ranking when they are indifferent to items, perhaps through lack of experience or knowledge.

A related issue is the appropriate denominator in calculating proportions or percents. Again, there will almost always be a difference between "number of respondents" and "number of responses". Either or both may be of substantive interest.

We flag here two pertinent details specific to Stata.

First, remember when working with integer variables that numeric missing counts as nonzero and therefore true. A detailed discussion of this point was given in an earlier column (Cox 2002a). This can be especially important when trying to produce, or when working with, indicator variables for which the possible nonmissing values are just 1 and 0.

Second, note that `egen, eqany()` and `egen, neqany()`, like `egen, tag()`, do not ever return missing results. We say more on this later.

3 How to change data structure

As data are commonly represented in different data structures, and no one structure is ideal, we need to consider how to change data structures. We will look in turn at various frequently needed restructurings.

3.1 Many-to-one mappings: concatenating variables

You can concatenate variables by adding them as string variables or as the string equivalent of numeric variables. A tool specifically for this purpose is `egen, concat()`. See [R] **egen** for more detailed discussion and examples. For example, given

```
            q1_1      q1_2      q1_3      q1_4      q1_5      q1_6
  1.           1         6         0         0         0         0
  2.           5         2         1         0         0         0
  3.           5         0         0         0         0         0
  4.           3         0         0         0         0         0
  5.           6         3         0         0         0         0
```

you can type

```
. egen response = concat(q1_*)
```

You need not worry about whether the variables are numeric or string, as `egen, concat()` automatically converts to string equivalent. You might want to remove the zeros padding out the result

```
         response
  1.       160000
  2.       521000
  3.       500000
  4.       300000
  5.       630000
```

which is easy with one of Stata's inbuilt string functions ([R] **functions** again):

```
. replace response = subinstr(response,"0","",.)
```

Given a structure of indicator variables

```
            q1_R   q1_SPlus     q1_SAS    q1_SPSS   q1_Stata  q1_others
  1.           1          0          0          0          0          1
  2.           1          1          0          0          1          0
  3.           0          0          0          0          1          0
  4.           0          0          1          0          0          0
  5.           0          0          1          0          0          1
```

you might prefer a concatenation more interpretable than `"100001"`, `"110010"`, etc. This code yields values like `"R others"`:

```
. gen str1 q1 = ""
. qui foreach p in R SPlus SAS SPSS Stata others {
.       replace q1 = q1 + "`p' " if q1_`p' == 1
. }
. replace q1 = trim(q1)
```

(In Stata 8, specifying `str1` has become optional.) For more detail on `foreach`, see [P] **foreach** or a previous column dedicated to it and its siblings (Cox 2002b).

3.2 Many-to-one mappings: reshaping to long

First, let us suppose that our data are

```
              id      q1_R     q1_SAS   q1_SPlus   q1_Stata  q1_others        sex
  1.           1         1          0          0          0          1       male
  2.           2         1          0          1          1          0     female
  3.           3         0          0          0          1          0       male
  4.           4         0          1          0          0          0     female
  5.           5         0          1          0          1          0     female
```

which is an example of what in `reshape` jargon is described as a wide data structure. The variables `q1_*` are numeric indicator variables. Later, we will comment on data in which ranks are given.

Conversion of this structure to a long data structure, in which program choice is represented by a single variable, is a problem best tackled by the `reshape` command, documented at [R] **reshape**. The key to `reshape` problems is to think in terms of a data matrix in which data are ordered by rows and columns, indexed as conventionally in matrix algebra by i and j, respectively. The rows we have are defined by the distinct values of `id`, and the columns we have are the variables `q1_*`. The variable names have in common a stub `q1_`, and they differ in the suffixes following the stub, `R`, `SAS`, etc. If the variable names do not have this stub plus suffix form, you will need to apply `rename` (see [R] **rename**) or possibly some other renaming command such as `renvars` (Cox and Weesie 2001) before you can apply `reshape`.

Our reshaping will be mapping the columns of the data matrix (variables `q1_*`) into one column, with other variables being rearranged to match. We specify the stub, and we also need to spell out that the data variable to be created will be string.

```
. reshape long q1_ , i(id) string
```

The result is

```
             id          _j        q1_         sex
  1.          1           R          1        male
  2.          1         SAS          0        male
  3.          1       SPlus          0        male
  4.          1       Stata          0        male
  5.          1      others          1        male
  6.          2           R          1      female
  7.          2         SAS          0      female
  8.          2       SPlus          1      female
  9.          2       Stata          1      female
 10.          2      others          0      female
 11.          3           R          0        male
 12.          3         SAS          0        male
 13.          3       SPlus          0        male
 14.          3       Stata          1        male
 15.          3      others          0        male
 16.          4           R          0      female
 17.          4         SAS          1      female
 18.          4       SPlus          0      female
 19.          4       Stata          0      female
 20.          4      others          0      female
 21.          5           R          0      female
 22.          5         SAS          1      female
 23.          5       SPlus          0      female
 24.          5       Stata          1      female
 25.          5      others          0      female
```

which is almost where we want to be. There is possibly no point in being explicit about programs not used, so we could

```
. drop if q1_ == 0
```

and follow by `drop`ping that variable altogether and using a more intuitive name:

```
. drop q1_
. rename _j q1
```

Here is the result:

```
            id        q1       sex
  1.         1         R      male
  2.         1    others      male
  3.         2         R    female
  4.         2     SPlus    female
  5.         2     Stata    female
  6.         3     Stata      male
  7.         4       SAS    female
  8.         5       SAS    female
  9.         5     Stata    female
```

As seen, we need not worry about variables such as `sex`, which are constant within `id`. They will get carried along automatically.

We promised to look at data in which ranks were given, for example,

```
            id      q1_1      q1_2      q1_3       sex
  1.         1         R    others                 male
  2.         2         R   S-Plus     Stata     female
  3.         3     Stata                          male
  4.         4       SAS                        female
  5.         5     Stata       SAS              female
```

The data matrix we have here has rows defined by the distinct values of `id` and columns that are the variables `q1_*`. The new data structure will have a single variable indicating software rank. This can be done directly:

```
. reshape long q1_ , i(id) j(rank)
```

The result is

```
            id       rank       q1_        sex
  1.         1          1         R       male
  2.         1          2    others       male
  3.         1          3                 male
  4.         2          1         R     female
  5.         2          2    S-Plus     female
  6.         2          3     Stata     female
  7.         3          1     Stata       male
  8.         3          2                 male
  9.         3          3                 male
 10.         4          1       SAS     female
 11.         4          2               female
 12.         4          3               female
 13.         5          1     Stata     female
 14.         5          2       SAS     female
 15.         5          3               female
```

We do not need observations with missing `q1`, and we can clean up the variable name:

```
. drop if missing(q1_)
. rename q1_ q1
```

(In Stata 8, you can say `mi()` as an abbreviation for `missing()`. Over time, knowing that could save many keystrokes.) The result is

```
            id       rank         q1        sex
  1.         1          1          R       male
  2.         1          2     others       male
  3.         2          1          R     female
  4.         2          2     S-Plus     female
  5.         2          3      Stata     female
  6.         3          1      Stata       male
  7.         4          1        SAS     female
  8.         5          1      Stata     female
  9.         5          2        SAS     female
```

This example was of a string variable. Any value labels attached to a numeric variable survive the `reshape`, so it is immaterial whether `q1` is string or numeric with labels. However, in practice it is a good idea to ensure that the numeric variables in the data matrix all share the same value labels.

3.3 One-to-many mappings: indicator variables

Given a composite variable, with values such as `"125"` or `"Stata R"`, how can it be converted to a set of indicator variables? One answer lies in the `index()` function, one of Stata's string functions. We assume here that you are following our advice and holding the codes as a composite string variable. If not, then in the examples below use (e.g.) `index(string(`*varname*`))` rather than `index(`*varname*`)`.

`index()` finds the position of one string within another. It is yet another useful string function ([R] **functions**). The string `"I"` has position 10 in the string `"Where am I?"`, as the first starts at the 10th position within the second. `index()` returns 0 if the first string is nowhere included in the second (e.g., `"you"` is not within `"Where am I?"`). Thus, in general, a positive result from `index()` means that one string is included within another (often, precisely where is of no consequence), and a zero result means that it is not.

We can also feed to `index()` any expression which evaluates to a string, such as the name of a string variable, so that a new variable can be generated as follows:

```
. generate byte q1_1 = index(spkg1, "1") > 0
```

`index(spkg1, "1")` will return a positive number if `"1"` is included in a value of `spkg1` and 0 otherwise. `index(spkg1, "1") > 0` will in turn evaluate to 1 if true and to 0 if false, thus yielding an indicator variable. For background, see the previously mentioned column discussing true and false in Stata (Cox 2002a).

Note in passing the specification of a byte variable type, possible in this case because we know that the possible values are well within the limits for that data type, as discussed at [U] **15.2.2 Numeric storage types**. Using an economical data type for an indicator variable can be helpful whenever space is short.

We will want to generate similar variables for other answers. Doing this variable by variable can be avoided, for example, by using `forvalues`:

```
. forvalues i = 1/6 {
.       generate byte q1_`i' = index(spkg1, "`i'") > 0
. }
```

For more detail on `forvalues`, see [P] **forvalues** or Cox (2002b). A further extension would be something like

```
. forvalues i = 1/6 {
.       capture assert index(spkg1, "`i'") == 0
.       if _rc {
.               generate q1_`i' = index(spkg1, "`i'") > 0
.       }
. }
```

What is going on here? Any statement tested by `assert` will yield a so-called return code that is zero if the statement is true for all observations examined and a return code that is nonzero (in fact, 9) if it is false. We test to see if any observations contain values other than zero before we generate a new variable. The `capture` ensures that everything continues smoothly, whatever the outcome. See [R] **assert** and [P] **capture** for further examples.

In particular, in our dataset nobody uses SPSS, and so arguably we could dispense with an indicator variable for that choice. When we get to

```
assert index(spkg1, "4") == 0
```

this assertion will be true of all the data and the return code from `assert` will be 0. So, the return code—which is accessible in `_rc`—will be nonzero and thus true. More generally, this approach will avoid creation of variables for any choices that were possible, but which happen to have been chosen by none of the sample.

This approach will work well with choices coded by one-digit characters, numeric or otherwise. You need to be more careful, however, when the choices include, say, `"1"`, `"10"`, or `"11"`, as a search for the character `"1"` will then find it whenever it occurs as part of `"10"` or `"11"`, say. Given space separation, as in `"1 10 11"`, one possibility is to search for `" 1 "` within the string expression `" " +` *string_variable* `+ " "`. Another possibility is to split the variable into "words" and then work from the resulting variables. This is explained in more detail in the next subsection. Typically easier, however, are unambiguous strings, as exemplified by

```
. foreach p in R S-Plus SAS SPSS Stata others {
.       local P : subinstr local p "-" ""
.       gen byte q1_`P' = index(spkg2, "`p'") > 0
. }
```

This code generates the variables `q1_R`, `q1_SPlus` and so forth, with values 1 and 0 just like in the example before. Incidentally, in the case of S-Plus, we need to catch the hyphen, which may not appear as a character in a variable name. Rather than massaging our example to remove all the wrinkles, we left this quirk as an illustration of the little problems that arise. The syntax for substituting within strings is documented at [U] **21.3.6 Extended macro functions**. Note again that this is all totally literal and thus dependent on consistent spelling, use of spaces, and use of lowercase and uppercase. On that last point alone, we can be more broad-minded in this way,

```
. foreach p in S-Plus SAS SPSS Stata others {
.       local P : subinstr local p "-" ""
.       gen byte q1_‘P’ = index(lower(spkg2), lower("‘p’")) > 0
. }
```

but we need a separate approach for R, given that the character "r" is evidently part of the string "others".

Finally, you may catch choices never made, just as before:

```
. foreach p in R S-Plus SAS SPSS Stata others {
.       local P : subinstr local p "-" "
.       capture assert index(lower(spkg2), lower("‘p’")) == 0
.       if _rc {
.               gen byte q1_‘P’ = index(lower(spkg2), lower("‘p’")) > 0
.       }
. }
```

3.4 One-to-many mappings: splitting variables

A composite string variable with values such as "125" or "43" can be split into individual str1 variables by a simple loop. You just need to find out the length of the composite, say from describe. Suppose that you want to split a str7 variable:

```
. forvalues i = 1/7 {
.       gen str1 r‘i’ = substr(response,‘i’,1)
. }
```

A composite string variable with values such as "Stata R" or "coffee,beer", in which words or phrases or other elements are separated by some punctuation, say, a space or a comma, is best handled by another approach. In Stata 8, this can be done with the split command (see [R] **split**). In Stata 7, you can use the predecessor of that command, also split, available from SSC. In Stata 6, you can use the predecessor of *that* command, strparse, also available from SSC.

3.5 One-to-many mappings: reshaping to wide

First, let us suppose our data are like

```
              id          q1       sex
   1.          1           R      male
   2.          1      others      male
   3.          2           R    female
   4.          2      S-Plus    female
   5.          2       Stata    female
   6.          3       Stata      male
   7.          4         SAS    female
   8.          5         SAS    female
   9.          5       Stata    female
```

which is an example of what was earlier described as a long data structure. To resolve an ambiguity, let us specify that `q1` is a string variable. Later, we will comment on the case of a numeric variable with value labels. Finally, we will comment on data in which ranks are given.

To convert this structure to a wide data structure in which each distinct answer in `q1` is represented by a single variable, we need to use `reshape`, as discussed earlier.

Once again, the key to such reshape questions is to think in terms of a data matrix in which data are ordered by rows and columns, indexed as conventionally in matrix algebra by i and j, respectively. The rows we desire are defined by the distinct values of `id`, and the columns we desire are defined by the distinct values of `q1`. Those values will be used as the suffixes of a set of variable names. If `q1` is a string variable, we immediately have a small problem, as indicated earlier: the hyphen within S-Plus is not acceptable within a variable name. We could fix this by another string function ([R] **functions**)

```
. replace q1 = subinstr(q1,"-","",.)
```

or in more difficult situations we could `encode` a string variable into a numeric variable. In the matrix itself, we want indicator variables in which 1 represents yes and 0 no. All our observations at present are in effect instances of 1, but we need to make that explicit:

```
. gen byte one = 1
```

That creates a variable which is 1 in every observation. In most circumstances, such a variable would be pointless, but here it is essential. Once again, the variable is created as a byte variable, to economize on storage. You can dispense with this detail if you have plenty of memory to spare.

Now, we can `reshape`:

```
. reshape wide one, i(id) j(q1) string
```

We need not worry about variables such as `sex`, which are constant within id. They will get carried along automatically. (If, contrary to assumption, they are not constant within `id`, then you will get an error message and no `reshape`, as something that should be true of your data is in fact false.) Here is the result of the `reshape`:

```
            id      oneR     oneSAS  oneS_Plus   oneStata  oneothers        sex
  1.         1         1          .          .          .          1       male
  2.         2         1          .          1          1          .     female
  3.         3         .          .          .          1          .       male
  4.         4         .          1          .          .          .     female
  5.         5         .          1          .          1          .     female
```

We are almost done, but, depending on taste, there may be some cleaning up to do. First, we have a stub for the new variables that may not be to our liking. One specific way to fix that is with `renpfix` (see [R] **rename**)

```
. renpfix one q1_
```

Second, we may wish to change all the missings in `q1_*` to 0. Once again, a specific command can do this, `mvencode` (see [R] **mvencode**):

```
. mvencode q1_*, mv(0)
```

We promised to comment on the case in which the argument of `j()`, here `q1`, is a numeric variable with value labels attached. The code is very similar:

```
. gen byte one = 1
. reshape wide one, i(id) j(q1)
. renpfix one q1_
. mvencode q1_*, mv(0)
```

However, note that a side-effect of `reshape` in this case is that the value labels associated with `q1` get dropped. For this reason, using a string variable is attractive here whenever practicable, bearing in mind that the values of the string variable are destined to be variable name suffixes: hence, only alphabetical, numeric, and underscore characters are allowed.

We also promised to look at data in which ranks were given. This is even easier.

```
          id        q1       sex      rank
  1.       1         R      male         1
  2.       1    others      male         2
  3.       2         R    female         1
  4.       2    S-Plus    female         2
  5.       2     Stata    female         3
  6.       3     Stata      male         1
  7.       4       SAS    female         1
  8.       5       SAS    female         2
  9.       5     Stata    female         1
```

The data matrix we seek has rows defined by the distinct values of `id`, and columns defined by the distinct values of `rank`. In the matrix itself, we want variables indicating software. This can be done directly:

```
. reshape wide q1, i(id) j(rank)
. renpfix q1 q1_
```

In this problem, any value labels attached to a numeric variable `q1` do survive the reshape, so it is immaterial whether `q1` is string or numeric with labels.

3.6 Many-to-many mappings

Many-to-many mappings: using egen

The most common problem here seems to be the creation of indicator variables from variables indicating successive choices. One pertinent tool in official Stata for the case of integer codes held in numeric variables is `egen, neqany()`. The result can be thought of as number of variables equal to any of the values specified. A sibling is `egen, eqany()`. The result can be thought of indicating whether values of variables are equal to any of the values specified.

For example, given the ranked responses

```
         q1_1     q1_2     q1_3     q1_4     q1_5     q1_6
  1.        1        6        0        0        0        0
  2.        5        2        1        0        0        0
  3.        5        0        0        0        0        0
  4.        3        0        0        0        0        0
  5.        6        3        0        0        0        0
```

we can `generate` the corresponding variables:

```
. forvalues i = 1/6 {
.       egen Q1_`i' = neqany(q1_*), val(`i')
. }
```

Note first that we loop over the possible answers (the values of the data), here the integers 1/6. More complicated sets of answers might be better handled using `foreach`. For each possible answer, in turn 1 2 3 4 5 6, we count how many of the variables, here the `q1_*`, are equal to any of the values specified, here just a single value in each case. Note also that we use uppercase `Q1` as a prefix for the new variables. Above all, appreciate that the new variables do not retain all the information in the originals, as we are ignoring the information on rank order.

Using `neqany()` rather than `eqany()` is a small wrinkle: with this example, we expect that each package will be mentioned at most once, but counting with `neqany()` allows a data check: any multiple count will show up as a value of 2 or more, and we can identify any respondent trying to subvert the questionnaire by repeatedly mentioning their favorite software. Alternatively, if it seems appropriate, we treat that as a measure of strength of interest.

Naturally, if you prefer, you can use `eqany()`. This is guaranteed to produce an indicator variable with values 1 or 0.

If you choose either of these functions, note that, as mentioned earlier, neither function ever produces missing values as a result. This may be surprising, but it was intended as a feature, given what were seen as the most likely uses of the generated new variables and how they might appear within Stata commands. However, if all the variables supplied as arguments are missing in an observation, then the result of `eqany()` or `neqany()` will be 0 for that observation. If you want to recode such 0s to numeric missing, here is one way to do it. We exploit the fact that the observation-wise (row-wise) maximum will be returned as missing by `egen, rmax()` if and only if all values examined in an observation are missing.

```
. egen rmax = rmax(q1_*)
. forvalues i = 1/6 {
.       replace Q1_`i' = . if rmax == .
. }
```

A crucial limitation is that both functions `eqany()` and `neqany()` apply only to integer codes. With arbitrary string codes, say,

```
          q1_1      q1_2      q1_3      q1_4      q1_5      q1_6
  1.         R    others
  2.     Stata    S-Plus
  3.     Stata
  4.       SAS
  5.    others       SAS
```

we need to create our own numeric measures from first principles, say,

```
. foreach p in R S-Plus SAS SPSS Stata others {
.       local P : subinstr local p "-" ""
.       gen byte q1_`P' = 0
.       forval i = 1/6 {
.             qui replace q1_`P' = q1_`P' + (index(q1_`i',"`p'") > 0)
.       }
. }
```

Here, we need a double loop, one over possible responses, initializing a variable to 0, and one over existing variables, adding 1 each time we find the package name inside. Counting whether `index()` returns a positive count is here a little more general than testing for equality, as it guards against the possibility that leading and/or trailing spaces have somehow been added to the variable. Nothing is done here directly about consistency of case—we have already seen how to tackle that—or about catching misspellings.

The code example here uses addition to produce an analog of `egen, neqany()`. One way of producing an analog of `egen, eqany()` is to use the or operator `|`, as `0 | 1` and `1 | 1` both yield 1. See help on logical operators at [U] **16.2.4 Logical operators**.

Many-to-many mappings: user-written programs

A program `zb_qrm` by Eric Zbinden (SSC; Stata 5) maps from a set of numeric variables with codes 1 upwards to a set of indicator variables for those codes. It also displays information on the occurrence pattern of indicators.

A program `mrdum` by Lee Sieswerda (SSC; Stata 7) is similar, but on the whole more general.

These programs differ over what is an appropriate denominator, all observations or all observations containing at least one response. As flagged previously, various choices may be sensible depending on the problem being tackled.

4 Tabulation and graphics

Tabulation and graphics are evidently both large and complex subjects. Our aim in this section is just to give some pointers to commands that may be of use.

Stata's official tabulation commands do not give much support to multiple response variables, although we gave an example earlier of the application of `tabstat`. A device that sometimes works very well for multiple responses held as long structures is to pretend to Stata that the data are panel data, and then to use `xttab` for tabulation (see [XT] **`xttab`**). For example, individuals may define panels, as expected, and rank or order of mentions of items may define a time surrogate. Another general strategy is to use an `egen` function to calculate something; (possibly) `egen, tag()` to tag just one observation in each of several groups; and then `list` to show the results. `list` is greatly enhanced in version 8 and may produce displays which are both informative and well presented. Using `collapse` or `contract` followed by `list` is more drastic, but may be just what you need, always noting that both commands are destructive.

Alternatively, user-written commands in this territory include

1. `tabcond` (Nicholas J. Cox, SSC; Stata 7). Tabulates frequencies satisfying up to 5 specified conditions. Zero frequencies are shown explicitly.

2. `tabm` (Nicholas J. Cox, SSC as part of `tab_chi`; Stata 7) Tabulates two or more comparable variables, in a combined two-way table of variables by values. Either all variables should be numeric, or all variables should be string.

3. `tabsplit` (Nicholas J. Cox, SSC as part of `tab_chi`; Stata 6) Tabulates frequencies of occurrence of the parts of a string variable. By default, the parts of a string are separated by spaces. Optionally, alternative punctuation characters may be specified.

4. `tabw` (Sasieni 1995; Stata 3.1). For each variable in a list, tabulates the number of times it takes on the values 0, 1, . . . , 9; the number of times it is missing; and the number of times it is equal to some other value. String variables are not tabulated but are identified at the end of the displayed table.

Graphics is, as said, a large area. We restrict ourselves to pointing to some new commands in Stata 8, which are helpful in showing frequencies of categorical data, especially `graph bar` and `graph hbar`. The options `ascategory` and `asyvars` offer considerable flexibility for multiple-response data, sometimes best thought of as different categories of one variable, and sometimes best thought of as several variables yoked together.

5 Summary

The problem of handling multiple responses is common in various fields, including social statistics and medical statistics. It appears to be one without a magic key, and there is no data structure optimal for all purposes. In practice, therefore, the watchword is flexibility, and for that, Stata users need to know their tools. Among those mentioned several times in this column—and indeed in some previous columns—are various inbuilt string and `egen` functions; using `foreach` and `forvalues` to loop over lists; and the use of the `reshape` command.

6 What's next?

Passing mention has been made in this column of some changes introduced in Stata 8, distributed from January 2003. In the next column, we will look more directly at one of many innovations in the new version, a suite of functions that facilitate the handling of lists.

7 Acknowledgment

Lee Sieswerda made several very helpful comments on a draft.

8 References

Cox, N. J. 2002a. Speaking Stata: How to move step by: step. *Stata Journal* 2(1): 86–102.

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

—. 2002c. Speaking Stata: On getting functions to do the work. *Stata Journal* 2(4): 411–427.

Cox, N. J. and J. Weesie. 2001. dm88: Renaming variables, multiply and systematically. *Stata Technical Bulletin* 60: 4–6. In *Stata Technical Bulletin Reprints*, vol. 10, 41–44. College Station, TX: Stata Press.

Sasieni, P. 1995. sg36: Tabulating the counts of multiple categorical variables. *Stata Technical Bulletin* 25: 15–17. In *Stata Technical Bulletin Reprints*, vol. 5, 93–96. College Station, TX: Stata Press.

About the Authors

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 ten commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is Executive Editor of *The Stata Journal*.

Ulrich Kohler is a sociologist at the Social Science Research Center in Berlin who has used Stata for several years. His interests include social inequality and political sociology. With Frauke Kreuter, he is author of the German textbook *Datenanalyse mit Stata*.