The Stata Journal (2002)
2, Number 4, pp. 411–427

Speaking Stata: On getting functions to do the work

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

Abstract. Functions in Stata take two main forms, built-in functions that are part of the executable and `egen` functions written in Stata's own language. These are surveyed, giving a variety of tips and tricks, and noting the large number of user-written `egen` functions available for download from the Internet. Two substantial examples, the calculation of percentile ranks and plotting positions, and the calculation of measures summarizing properties of the other members of a group, provide detailed illustrations of `egen` in action.

Keywords: pr0007, functions, egen, strings, percentile ranks, plotting positions, family data

1 Functions in Stata

The rationale for writing this column is not that the documentation for Stata is poor. In fact, the documentation for Stata is excellent, and I say that out of much more than politeness to Stata Corporation, our publisher. Enormous thought and effort has gone into producing and maintaining very full and detailed manuals. But in one way that is part of the problem faced by users, especially—but not only—new users. I have a couple of shelves of successive Stata manuals, starting with the manual for version 2, a single volume published in 1988. A decade or so ago, you could put the manual in your bag, read it at your leisure, and get a very good feeling for the whole program and its possibilities. Once that was acquired, you could build on it by learning about new features release by release. But now, when we have several volumes running to thousands of pages, it would require much more determination (and above all time) to do the same.

Naturally, almost no one need read everything, and you may know well that you can survive without survival, or that your survey can happily omit survey. Stata Corporation is committed to the idea that the official software is a unified whole, rather than a base together with detachable extra modules, but users are entitled to pick and choose. And equally naturally, much thought has been given by Stata Corporation and others to providing good solutions to this need to get to grips with Stata easily and effectively. They include the *User's Guide* as one synoptic overview, various texts and multimedia resources, courses and other materials on the Internet, and indeed this column.

One aim of these columns is to bring together material whose documentation is scattered over several sections of the manual, as with `by` *varlist*`:` (Cox 2002b) or the divide between numeric and string variables (Cox 2002c). Clearly, every organizing

© 2002 Stata Corporation

scheme splits some material that on another scheme belongs together. Another aim is to play up material that to my taste is understated in the manuals; thus, while `foreach` and `forvalues` are written up rather tersely in the *Programming Manual*, they deserve to be in the repertoire of all serious Stata users, whether they do or do not also write Stata programs (Cox 2002a). That is also the case with the subject of this column, the functions available for use with Stata.

The functions in Stata fall into two main groups. Here I set on one side those functions implemented as Stata commands, as many commands could fairly be described as implementing a function mapping arguments to results, and concentrate on what Stata's own documentation labels as functions.

First, Stata functions in the strict sense are those that are part of the executable, and they resemble in form functions as implemented in most comparable software, whether specialist statistical, mathematical, or scientific languages or general programming languages. Most frequently, calls to functions appear as part of expressions defining new variables or as part of conditions that evaluate to true or false. Thus, we might define a new variable `logx` as the natural logarithm of another variable `log(x)` or `ln(x)`, or we might test whether or not `mod(age,5) == 0`.

Second, and more idiosyncratic to Stata, are functions written in Stata's own language and driven by the `egen` command.

Here we look at these two groups in turn. Built-in functions are prominent in Stata, and my task is mostly to emphasize some general points and to give a few detailed tips. In contrast, `egen` is less prominent, and it seems common that users are relatively unaware of its existence and its possibilities, so my discussion is designed to counteract that.

2 Built-in functions

2.1 The general picture

Underlying all of Stata's built-in functions is the mathematical idea of a function, implemented as a mapping from arguments[1], which are enclosed in parentheses and separated by commas, to results.

Occasionally, a function needs no arguments, as with `uniform()`. The parentheses flag that it is indeed a Stata function, even if of an unusual kind, and they help reduce the number of reserved names. If just the name `uniform` were allowed to indicate the function, then that could no longer be a legal variable name.

Functions are documented fairly tersely at [U] **16.3 Functions** and even more tersely online under the help for functions. New functions are occasionally added between

[1] The quaint term *argument* makes more sense when understood as a metaphorical reference to the 'topic' that the function will 'deal with'. That is, the meaning of the word has shifted from a *dispute* to *the proof or evidence used in the dispute* and then to *the topic or theme being discussed*. Schwartzman (1994) gives interesting accounts of this and many other mathematical words.

releases; the online help in an updated Stata will show whether the function you need is in the software you have.

One characteristic of many built-in functions is that for speed and for accuracy they must be part of the code for the Stata executable (which is written in C). It would not be sensible for code for (say) tail probabilities of distributions or the gamma, digamma, and trigamma functions to be written in Stata's own programming language.

Two general but very practical points that are sometimes overlooked deserve emphasis.

Stata can be used as a calculator by feeding an expression to `display`; see [R] **display**. This can be very useful not only for getting one-off results but especially for a series of calculations that you want to document within a log. You do not need to read a dataset into memory to be able to do this.

Another noteworthy detail is that functions can feed directly on the results of other functions. This is often the way to calculate functions not provided as primitives in a single step without needing to calculate intermediate values or `generate` intermediate variables yourself. Let us look at some brief examples.

Lognormal random numbers can be obtained from the expression

`exp(`*mean* `+` *sd* `* invnorm(uniform()))`

Here *mean* and *sd* refer to the mean and standard deviation of the normal or Gaussian distribution yielded (approximately) by taking the natural logarithm of this expression. You would in practice replace *mean* and *sd* by numeric values, or the names of numeric variables, or even expressions evaluating to the numbers required.

The beta function $\Gamma(a)\Gamma(b)/\Gamma(a+b)$ can be obtained from the expression

`exp(lngamma(`a`) + lngamma(`b`) - lngamma(`a `+` b`))`

Finally, the hypergeometric probabilities $\binom{r}{k}\binom{n-r}{m-k}/\binom{n}{m}$ are obtained from

`comb(`r`,`k`) * comb(`n`-`r`,`m`-`k`) / comb(`n`,`m`)`

where `comb(`r`,`k`)` is used to calculate the individual binomial coefficients[2], namely $\binom{r}{k} = {}^rC_k$. The syntax of `comb(,)` is evidently constrained to fit on a single line, but echoes the use of $Z(r,k)$ by Argand around 1815 (Cajori 1929, p. 68) and the more recent use of $C(r,k)$ by Hamming (1973).

You might prefer, as a matter of taste, to split an operation involving several steps into a series of simple changes that are easier to think through, to read and to debug. The usually small price to pay for this may be the load of storing some extra values or variables, at least temporarily.

[2]Why does something so fundamental have such an ugly and obscure name? Conway and Guy (1996) use the excellent term *choice number*, which deserves wider currency.

2.2 Some especially useful functions

In many problems, you consciously want a function and merely have to look up the syntax to be used. In some problems, however, you may not be aware that functions exist to do what you want to do, so it is worth skimming the list and making a mental note of anything close to your likely needs. In addition to various mathematical, statistical, and string functions, be aware of specialized date functions, time-series functions, and matrix functions, which I shall not survey here. What follows are a few highlights, with associated tips and tricks. The selection is necessarily arbitrary.

Key string functions

A large fraction of basic problems with strings can be solved with one or more of `index(,)`, to find substrings, `substr(,,)` to return substrings, `length()`, returning the length of strings, and `trim()`, to remove any leading or trailing spaces. The first two are the most useful.

`index(`s_1`,`s_2`)` returns the position in s_1 in which s_2 is first found or zero if s_1 does not contain s_2. Here both s_1 and s_2 can be literal strings enclosed in `" "` or names of string variables: if the latter, Stata looks at the values in each observation. The principle is made clear by simple examples:

`index("Stata","a")` is 3, as the first `"a"` occurs as the third character after `"St"`.

`index("Stata","at")` is 3, as the position returned is that of the first character of s_2.

`index("Stata","q")` is 0, as `"q"` occurs nowhere within `"Stata"`.

In practice, the most common application is looking for occurrences of a literal string within a string variable. Thus `index(location,"NY")` is positive whenever the string variable `location` contains the literal string `"NY"`, so we can imagine selections such as

```
if index(location, "NY") > 0
```

or the complementary subset

```
if index(location, "NY") == 0
```

Whenever the fact of inclusion or exclusion is vital, and the precise location is immaterial, there are some handy shortcuts, which over a long period of Stata use can save you many keystrokes. The first selection can be abbreviated to

```
if index(location, "NY")
```

and the second to

```
if ~index(location, "NY")
```

We are invoking the rule that nonzero and zero arguments are regarded as indicating logically true and false respectively, as was explained in detail in an earlier column (Cox 2002b). Note also the use of the logical not operator, here denoted by `~`, although `!` may also be used.

This idea can be extended. If we wanted to include also instances of `"NJ"`, then we select

```
if index(location, "NY") | index(location, "NJ")
```

as the logical operator or, denoted by `|`, returns true if any of its arguments are true. Examples like this make the short-cut even more worthwhile.

`substr(`s`,`n_1`,`n_2`)` returns the substring of s starting at the n_1th column for a length of n_2. If $n_1 < 0$, the starting position is interpreted as distance from the end of the string. If n_2 is numeric missing (`.`), the remaining portion of the string is returned. Here, and indeed elsewhere, it helps to think of numeric missing as arbitrarily large. As before, s may be a literal string or the name of a string variable, but is most commonly the latter. If you knew that the first four characters of a variable should be cut, then you want `substr(`*strvar*`,5,.)`. If you knew that the last four characters of a variable should be cut, then you want `substr(`*strvar*`,1,length(`*strvar*`)-4)`.

Using functions in combination like this is very common, the main little problem being to get parentheses in the right places and balanced properly. It helps to remember that, as in mathematics, Stata works from the inside outwards and thus evaluates the innermost parenthesized expression first. An even more common example of combination is to find something with `index(,)` and then extract what you want with `substr(,,)`. How would you tell Stata to extract the first word of a string? One definition is that it ends just before the first space. Hence it is given by

`substr(`*strvar*`,1,index(`*strvar*`," ")-1)`

The first time you see expressions like this they can seem daunting, so let us spell this out step by step. We look for the first space within the values of *strvar*, which is at `index(`*strvar*`," ")`, and the position before that is 1 less. As the first word starts at position 1, the position of the last character is the same as the length of the first word.

This will work fine if and only if the first word (in the usual sense) does start at the first character and it is followed by at least one space. If there are leading spaces, we must instruct Stata to ignore them by specifying `trim()`:

`substr(trim(`*strvar*`),1,index(trim(`*strvar*`)," ")-1)`

In addition, if there is no space after the first word, then `index(`*strvar*`," ")` will evaluate to 0 and the length of the substring to be extracted will evaluate to -1. That is not illegal, but Stata returns an empty string when asked for a zero or negative length of string. There are various ways to tackle this, including a three-step calculation:

```
. generate str1 newvar = ""
. replace newvar = substr(trim(strvar),1,index(trim(strvar)," ")-1)
. replace newvar = strvar if missing(newvar)
```

In this example, lie various lessons. Complications in problems can usually be tackled by using various other functions from the toolkit provided, although this may entail breaking down a problem into component problems. The alternative of a function to match every distinct problem is not even an ideal, as in practice it would mean a

bloated Stata with far too many functions to be easy to use. As it happens, specific `egen` functions have been written for many of the key problems to do with extracting words.

cond(): two definitions in one

`cond(`x`,`a`,`b`)` gives one of two possible results depending on x: a if x evaluates to true (not 0) and b if x evaluates to false (0). The results can be either both numeric or both string. The gain here is entirely in concision, replacing two statements by one. Thus, let us `generate` a string variable to be used as a graph symbol:

```
. generate str1 symbol = cond(residual >= 0, "+", "-")
```

This could replace two statements, say

```
. generate str1 symbol = "+" if residual >= 0
. replace symbol = "-" if residual < 0
```

(You may have thought of another way to do it, as `substr("-+", 1 + (residual >= 0), 1)`.)

`cond(,,)` can be nested, thus yielding a multiple definition. My prejudice is that the resulting statements can be very difficult to read, to understand and above all to debug. I recommend, however, recasting most two-way branches in this manner.

max() and min()

`max(`x_1`,`x_2`,...,`x_n`)` returns the maximum of x_1, x_2, ..., x_n. Missing values are ignored. If all arguments are missing, missing is returned. `min()` is similar. If typing lots of commas to separate variables is irksome, it is likely that you should check out `egen`'s `rmin()` and `rmax()` functions.

missing()

`missing(`x`)` returns 0 if x is not numeric or string missing; otherwise, it returns 1. An advantage of `missing()` is that it works over all kinds of variables, so that you will not be tripped up by a type mismatch error. Be sure to exploit the shortcuts `if missing(`x`)` and `if ~missing(`x`)`. It is never necessary to say `if missing(`x`) == 1` or `if missing(`x`) == 0`. An example appeared earlier without comment, to make it seem intuitive.

mod(): modulus or remainder

`mod(`x`,`y`)` computes the modulus of x with respect to y, that is, the remainder on dividing x by y. Your watch or clock shows time modulo 12 hours or modulo 24 hours (modulo conventions about representing times in the hours before 1 a.m. or 1 p.m.).

Most commonly, x and y are positive integers or variables containing positive integers[3]. For example, the condition `if mod(age,5) == 0` selects ages divisible by 5, such as 5, 10, 15, 20 and so forth (Conroy 2002). `if mod(_n,2)` selects every observation with an odd observation number, and `if !mod(_n,2)` selects every observation with an even observation number. Why? There are only two possible remainders when dividing integers by 2, 1 if the number is odd and 0 if the number is even. Explicit comparison by using `==` or `~=` can often be avoided by relying, once more, on the principle that nonzero is true and zero is false.

3 egen: a library of functions in Stata's own language

3.1 The general picture

`egen` (see [R] **egen**) is a wrapper command for `egen` functions, which are written in Stata's own language following some fairly simple rules. They offer a way of adding user-written functions to the language. The main limits are: first, `egen` functions produce a single new variable; second, the manipulations must be suitable for programming as interpreted code; third, `egen` functions cannot be used anywhere in which you might use a built-in function. They can only appear within `egen` commands. In addition, `egen` does not support weights, but that can be fudged by specifying weights through an option.

Explaining how to write an `egen` program would take us outside the remit of this column, which is dedicated to showing how far you can get in Stata *without* programming, but suffice it to say that it is normally easy. Almost always, an existing program for an `egen` function will serve as a template, and you may need only to make changes amounting to a few lines.

It is helpful to know the following rule: `egen` function *function* is implemented as `_g`*function*`.ado`. Thus, if you were to write a new `egen` function `logit()`, it should be defined within `_glogit.ado`.

The manual entry for Stata 7 details 39 `egen` functions, but others exist outside official Stata. A search using `findit` on 14 October 2002 identified 70 more, published in the *Stata Technical Bulletin* or on the Internet, setting aside those now adopted directly or indirectly in Stata, and a few others now obsolete. Several more may be available by the time you read this. Some of these were stimulated by user questions, especially on Statalist. More information on any listed here is available through `findit` (McDowell 2001).

(*Continued on next page*)

[3]Some authors define the modulus for all real arguments. Knuth (1997, p. 39) defines x mod y as $x - y\lfloor x/y \rfloor$ if $y \neq 0$ and x if $y = 0$, where $\lfloor z \rfloor$ yields the greatest integer $\leq z$, the floor of z. Stata allows all positive y but returns missing for $y \leq 0$.

Name	Function	Author
`clsort`	sort a single variable	P. van Kerm
`cutjl`	modification of `cut()`	J. M. Lauritsen
`egenmore`	47 various	N. J. Cox, C. F. Baum, S. Stillman, N. Winter
`egenodd`	7 various	N. J. Cox (1999, 2000)
`egen_ics`	4 various	J. Weesie
`gtype`	genotype from alleles	D. Clayton
`htype`	haplotype from alleles	D. Clayton
`ms`	moving sum	S. Driver
`prod`	product	P. Ryan (2001)
`rmedf`	row median	S. Kolenikov
`rpos`	observations with $\geq k$ positive values	F. Wolfe
`rprod`	row product	P. Ryan (1999)
`soundex`	soundex	M. Blasnik
`std01`	standardize to range $[0,1]$	S. Kolenikov
`wtmean`	weighted mean	D. Kantor

Most `egen` functions are for data management or for storing summary statistics within variables, especially group summaries. Numeric and string results are both produced, and various date and other time series functions also feature fairly strongly. In a sense, the existence of `egen` is a concession to human frailty, as most functions could be emulated in a few lines by anyone highly fluent in Stata's constructs, especially those expert in `by` *varlist:*, `foreach` and `forvalues`. At the same time, it can be very useful to have encapsulations of commonly used code, especially when a problem seemingly simple can be complicated by the failure of simplistic assumptions. We have already seen an example, the first 'word' problem: `egen, ends() head` covers all the cases discussed here, allowing a command statement to produce the new variable.

To close, and give a better flavor of how `egen` can be used, we will give two more substantial examples.

3.2 How can I calculate percentile ranks or plotting positions?

The problem

Some kinds of data are often reported as percentile ranks. A test score may be reported as a percentile rank of 95% if 95% of scores are less than or equal to that score. A newborn baby's weight may be reported in the same way.

The idea of a plotting position is essentially similar, except that conventionally plotting positions are reported as proportions rather than percents. Their name stems from their use in probability plots. Given a set of ordered values, proportions are assigned recording the fraction of values at or below each value. These proportions may then be used directly or indirectly in plots. For example, a quantile-quantile plot for testing normality of distribution compares observed quantiles with quantiles for

a sample of the same size from a normal distribution, determined by evaluating the quantile (inverse distribution) function for the normal at the plotting positions.

Official Stata has no functions for these calculations, although user-defined functions may be found. They may, however, be performed easily using `egen` functions. The lack of built-in functions has one advantage; it obliges users to select the precise procedure to be used, given that several slightly different rules exist.

Note that the calculation of the empirical cumulative distribution, at least as implemented in Stata, is a related but subtly different problem. In official Stata, this calculation may be performed using `cumul` (see [R] **cumul**). `cumul` is most closely geared to preparing a plot of cumulative probabilities (y-axis) versus observed values (x-axis). When tied values occur, `cumul` will not assign the same cumulative probability to each tied value. This is immaterial and indeed even desirable for graphical purposes: vertical line segments will be produced whatever the convention about ties, and when visible plot symbols are used, it is often helpful to get an impression of the frequency of ties. In contrast, the focus here is mainly on problems in which identical values should all be assigned the same proportion or percent.

Ranks

To assign ranks, we might `sort` on a variable (say `mpg` from the auto dataset) and then use the observation number:

```
. sort mpg
. generate rank = _n
```

As sorting in Stata puts lowest values first, the lowest value, or more precisely, the value sorted to the top of the dataset, would be assigned rank 1. This is also the most common statistical convention. However, this code is too simple to work perfectly except in the very simplest situation with no tied values and no missing values. For example, looking at data for `mpg`, we see that the first two values after sorting are tied at 12, yet these are assigned ranks 1 and 2 by the `generate` command above. In statistics it is common to assign the same rank to each of several tied values such that the sum of the ranks is preserved. Ranks 1 and 2 give a sum of 3 to be preserved, so each would become rank 1.5 instead. We could carry out this reassignment for all tied values ourselves, but it is convenient that it is encapsulated in `egen, rank()`:

```
. egen rank = rank(mpg)
```

This function also takes care of any missing values. The `sort` command would sort any (numeric) missing values to the end of the dataset, so that `generate rank = _n` would then assign them the highest ranks. In most circumstances, however, a missing value should be matched by a missing rank, and this is what is done by `rank()`. If you are curious to see how the reassignment for ties is done, have a look inside the code with your favorite text editor (Stata's own `doedit` would be fine). Typing

```
. which _grank
```

will tell you where the code is on your system. (Recall that the prefix `_g` is generic to all `egen` functions.)

`egen, rank()` allows separate calculations of ranks for each of several groups defined by a classifying variable:

```
. bysort foreign: egen rank = rank(mpg)
```

(For more on `bysort` or, more generally, on `by`, see the online help or my earlier column (Cox 2002b).)

Yet another nuance is being able to rank in reverse. Suppose that we preferred to assign the highest (and best) value of `mpg` rank 1. To do this, we exploit the fact that `egen, rank()` feeds on an expression (denoted *exp* in the `egen` syntax diagram), which can be more complicated than a single variable name, so we can work with (in particular) negated values. Thus

```
. egen rank = rank(-mpg)
```

would reverse the ranks for us. Finally, before leaving ranks as such, note that `egen, rank()` also has `field`, `track`, and `unique` options. The name `track` was suggested by track events such as running in which not only does the lowest time win, but two values that tie for first would also both be ranked first equal. (In sports, no one talks about rank 1.5.) Similarly, the name `field` was suggested by field events such as jumping or throwing in which the greatest distance or height wins, and there is a corresponding rule for ties. `unique` arbitrarily assigns unique ranks to each of several tied values and is of less concern here.

Plotting positions

Coming now to the heart of the matter, consider a magnificent sample of seven values, here ordered for convenience:

0 0.57722 1 1.61803 2.71828 3.14159 10

Evidently, the median, the middle of the ordered values, is 1.61803 and should be assigned a percentile rank of 50%, or a plotting position of 0.5, as the value halfway through the distribution. The ranks corresponding to these values are clearly, with the default Stata (and statistical) convention of lowest first, 1 2 3 4 5 6 7. Using i and n to denote rank and number of values, a fraction of i/n would produce plotting positions for this sample of $1/7 \ldots 7/7$, but $4/7$ for the middle value. Nor does such a rule treat the tails symmetrically. Similarly, a rule of $(i-1)/n$ would produce plotting positions $0/7 \ldots 6/7$, but $3/7$ for the middle value, and does not treat the tails symmetrically either. From this, one obvious compromise is to split the difference at $(i-0.5)/n$. This rule goes back at least as far as 1914, when it was introduced into statistical hydrology by Allen Hazen (1869–1930), an American civil engineer in private practice, otherwise best known for his work on urban water supplies.

There is a small amount of literature on choice of plotting positions, focusing variously on their use for estimation of parameters or on testing of hypotheses given specific

distributions. Many of the positions commonly recommended are members of a family $(i-a)/(n-2a+1)$. Thus, $a = 0.5$ yields Hazen's rule; $a = 0.375$ is a rule suggested by Gunnar Blom (b. 1920), especially for normal probability plots; $a = 0$ is a rule advocated by Waloddi Weibull (1887–1979) and Emil J. Gumbel (1891–1966). All of these rules yield 0.5 for the single middle value when the sample size is odd. Note that Hazen's rule is wired into the official Stata command `quantile`, while the Weibull–Gumbel rule is wired into the official Stata commands `pnorm`, `qnorm`, `pchi`, and `qchi`. For further discussion, see Barnett (1975), Cunnane (1978), or Harter (1984).

Another rule, which, for example, is often used in spreadsheet software, is $a = 1$, that is, $(i-1)/(n-1)$. This also yields 0.5 for the single middle value when sample size is odd and treats the tails symmetrically. However, the results of 0 and 1 for sample extremes may not be universally suitable. In particular, for probability plots for the normal and other distributions on the whole of the real line quantile functions (inverse distribution functions) do not have finite values for arguments of 0 and 1; thus, sample extremes are not plottable with this rule. This rule in practice is often used with the `track` rule for ties.

Obtaining results in Stata

The choice of precise rule, from the possibilities mentioned or from any others that may appeal, is up to the user. All that remains is to calculate the sample size n. In the simplest circumstance, this is just the built-in `_N`. To take proper account of missing values, and to be able to work with multiple groups, it is better to use `egen, count()` to count the number(s) of nonmissing values.

Suppose that we want Hazen plotting positions for *varname*:

```
. egen n = count(varname)
. egen i = rank(varname)
. generate hazen = (i - 0.5) / n
```

Or, we want Weibull plotting positions, but separately by *byvar*:

```
. bysort byvar: egen n = count(varname)
. by byvar: egen i = rank(varname)
. generate weibull = i / (n + 1)
```

Or, we want spreadsheet-style percent rank:

```
. bysort byvar: egen n = count(varname)
. by byvar: egen i = rank(varname), track
. generate pcrank = (i - 1) / (n - 1)
```

All of these examples return results between 0 and 1; for percents, just multiply by 100.

In sum, despite a mass of small details to be thought about, these calculations boil down to the application of two `egen` functions.

3.3 How do I create variables summarizing for each individual properties of the other members of a group?

Examples: data on families

Suppose that you have data on families. For each person in each family, it may be useful to calculate variables that summarize properties of the other members of the same family. How many other children are there? What is their average, maximum, or minimum age? Is there an older child or a younger child? The more general problem can be described as summarizing properties, for each individual, of the other members of the same group.

Let's look at some invented data. For what follows, it is essential to have a group identifier, so in this example we have an identifier for each family. It is not always essential to have an individual identifier, but what follows does depend upon each person occurring just once in the dataset. In practice, however, such data do usually include individual identifiers.

	family	person	female	age
1.	1	1	1	36
2.	1	2	1	16
3.	1	3	1	14
4.	2	1	0	45
5.	2	2	1	42
6.	2	3	0	14
7.	2	4	1	12
8.	2	5	0	10
9.	3	1	0	39
10.	3	2	1	36
11.	3	3	0	11
12.	3	4	1	9
13.	3	5	1	7
14.	3	6	1	3

We will suppose that `female` is recorded as 1 for female and 0 for male. Such 0-1 coding is in a sense arbitrary, but makes life easier, especially for statistical modeling in which the response is a binary variable and (more directly important here) for counting values within each group.

Specific problem: for each child, how many other children are there?

Let's define children as those for whom age is 17 and under. For each child, how many other children are there? This is simply the number of children in the family, minus 1 if each person is a child. (In family 3, with 4 children, for each child there are 3 other children.)

For any calculation like this, it is always worth looking to see whether `egen` provides an answer, at least to part of the problem. In particular, `egen, sum()` with `by` *varlist*`:` is natural for producing sums, including counts, separately for groups defined by one or more variables. `egen, count()` used in this way is also often useful, but a little less general in application, so we will concentrate for now on `sum()`.

```
. bysort family: egen nchild = sum(age <= 17)
. replace nchild = nchild - (age <= 17)
```

`age <= 17` will be true (evaluates to 1) whenever `age` is less than or equal to 17, and false (evaluates to 0) otherwise. Adding up the 1s and 0s within `egen, sum()` is the same as counting the observations for which `age <= 17`. We then subtract `age <= 17` from each observation. The effect of the `bysort family:` prefix is to count within families. The effect of the `replace` correction is confined to individual observations.

The syntax for `egen` indicates that `sum()` works on an expression *exp*. As just illustrated, the argument need not be a single variable, but, very usefully, can be something more complicated. Interested in not other children, but other female children? This is hardly more difficult:

```
. bysort family: egen nsisters = sum(age <= 17 & female)
. replace nsisters = nsisters - (age <= 17 & female)
```

This solution also assigns values to adults, those with `age` greater than or equal to 18. This could be useful, or not useful, depending on your substantive problem. If you wanted to exclude adults completely from the calculation, you could specify `if age <= 17` on the `egen` command, and values for adults would then be missing.

Note that if we wanted to count not "other children" but "other adults", we should be a little more careful. The expression `age >= 18` includes missing values for `age`, as in Stata missing counts higher than any other numeric value. Often we will want to exclude those with the condition `age >= 18 & age < .` unless we know that missing ages can be treated as implying an adult.

Generic problem: totals and means

Other totals, and by extension means, can be calculated using the same general approach. Put very simply,

1. Calculate the sum for each group.
2. Subtract each member's contribution from that sum (possibly, the contribution is 0).
3. If needed, calculate the mean as the sum divided by the number of values.

What is the average age of the other children in each family? Here is one solution:

```
. bysort family: egen sumage = sum(age) if age <= 17
. replace sumage = sumage - age
. generate meanage = sumage / nchild
```

This solution excludes the adults. Not only are they not included in the summation of `age`, but they also receive missing values for the result. In the `replace` command, we can be cavalier about excluding or not excluding the adults; either way, the missing values will not be changed by either subtraction or division.

What if we want to include the adults? That is, we want a record, for each adult, of the average age of the children. Here is a solution to that:

```
. bysort family: egen sumage = sum(age * (age <= 17))
. replace sumage = sumage - age * (age <= 17)
. generate meanage = sumage / nchild
```

In this, the multiplier `age <= 17` has the effect that the summand is 0 whenever `age` is 18 or more, so that the sum is the correct sum and is assigned to all observations in each family.

Generic problem: other statistics

What we have done so far hinges delicately on two properties of sums: first, the sum for "everybody else" is just the sum for "everybody" minus the sum (the value) for this observation; and second, that the value of a sum is not affected by adding or subtracting 0. When we turn to other summary statistics, we can no longer rely on these properties. We need a more general approach.

In broad terms, we need to do the work within a loop:

for each member in the family {
    calculate a statistic from data on the family
    assign the result to that member of that family
}

Specific problem: maximum age of the other children

Let's suppose that we want to know, for each child, the maximum age of the other children in the same family. Within the loop, we will find ourselves assigning chunks of values; for that task, we cannot use `generate` repeatedly. We can use `replace` repeatedly, so we need to `generate` a variable before we can do that:

```
. generate maxage = .
```

Next, we need an identifier running from 1 upwards to assign to each person in the family. In our little dataset, there was already such an identifier, but if there was not, one could easily be created using `bysort`:

```
. bysort family: generate pid = _n
. summarize pid
```

Note that under `by` *varlist*: `_n` is interpreted within each group of observations, not for the whole dataset. For this problem, it does not matter that `pid` is arbitrary;

we just need a systematic way of doing the calculations in turn for each member of the family. The `summarize` shows us the maximum value of `pid`, which we will need shortly. We could also pick up the value of the maximum as `r(max)`, which is important for any automation of the whole process.

Within the loop, we need a way of excluding each value of `pid` from the calculation. Here is one way to do it using `forvalues`:

```
. quietly forvalues i = 1 / `r(max)' {
.         generate include = 1 if pid != `i' & age <= 17
.         bysort family: egen work = max(age * include)
.         replace maxage = work if pid == `i'
.         drop include work
. }
```

The `forvalues` construct loops over values of the local macro `i`, which is set in turn to 1, then to 2, and so on, up to the maximum of `pid` as returned by `summarize`. The macro is automatically incremented each time through the loop. In practice, most Stata programmers use the abbreviation `forval`. Within the loop, the value of `i` is referred to as `` `i' ``. For more details, see [P] **forvalues** or Cox (2002a). The `generate` statement produces a variable that is 1 if the observation is to be included in the calculation, and missing otherwise. The expression `age * include`, which is then fed to `egen, max()`, is `age * 1` or `age` when `include` is 1, and `age * .` or missing `.` when `include` is missing. What `egen, max()` does is exclude missings from the calculation, and only if all the values in each group are missing will the maximum be returned as missing. Stata has a general rule that numeric missing is larger than any other numeric value, but as mentioned in section 2.2.3, it assumes when calculating maxima that you really want the largest nonmissing value. We then use the result of that calculation to `replace` the `maxage` value for the current member of the family. Finally, it is easiest to `drop` the variables `include` and `work` so that Stata can start fresh next time around the loop.

Why is this loop not the following?

```
. quietly forvalues i = 1 / `r(max)' {
.         bysort family: egen work = max(age) if age <= 17 & pid != `i'
.         replace maxage = work if pid == `i'
.         drop work
. }
```

The reason is that this will not work as desired. The result of the `egen` calculation will be missing for observations excluded by the `if` condition. In fact, the result of the loop is that all values of `maxage` will be missing.

For each child, there is an older one (strictly, one or more) if `maxage` is greater than `age`,

```
. generate olderch = maxage > age if age <= 17
```

We could use a similar approach to get the minimum age of the other children, and thus to determine whether there are younger children.

The same general scheme can be used for other `egen` functions that take an expression *exp* as an argument. See the help for `egen`.

4 Summary

Functions are provided to do work for you, to encapsulate in a single call the calculation of a new value or a new variable from appropriate arguments. Stata's approach to supplying functions mixes standard and original styles, built-in functions and those that can be run using `egen`. This combination affords users with a large and growing toolkit. Knowing what is in that toolkit and how to make best use of it is a key part of getting to use Stata more easily and more effectively.

5 What's next?

Thus far in this series our focus has been resolutely microscopic, on the level of commands or blocks of commands needed to achieve specific aims. In the next column we will look at a macroscopic issue, the structure of your dataset and how to get it into a shape you need.

6 Acknowledgments

Many thanks to those Stata users who by questions or programs have contributed to the stock of `egen` functions, and especially to Kit Baum for his maintenance of the Statistical Software Components archive.

7 References

Barnett, V. 1975. Probability plotting methods and order statistics. *Applied Statistics* 24: 95–108.

Cajori, F. 1929. *A history of mathematical notations. Volume II: Notation mainly in higher mathematics*. La Salle, IL: Open Court.

Conroy, R. M. 2002. Choosing an appropriate real-life measure of effect size: the case of a continuous predictor and a binary outcome. *Stata Journal* 2(3): 290–295.

Conway, J. H. and R. K. Guy. 1996. *The book of numbers*. New York: Copernicus.

Cox, N. J. 1999. dm70: Extensions to generate, extended. *Stata Technical Bulletin* 50: 9–17. In *Stata Technical Bulletin Reprints*, vol. 9, 34–45. College Station, TX: Stata Press.

—. 2000. dm70.1: Extensions to generate, extended: corrections. *Stata Technical Bulletin* 57: 2. In *Stata Technical Bulletin Reprints*, vol. 10, 9. 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.

—. 2002c. Speaking Stata: On numbers and strings. *Stata Journal* 2(3): 314–329.

Cunnane, C. 1978. Unbiased plotting positions–a review. *Journal of Hydrology* 37: 205–222.

Hamming, R. W. 1973. *Numerical methods for scientists and engineers*. New York: McGraw-Hill.

Harter, H. L. 1984. Another look at plotting positions. *Communications in Statistics, Theory and Methods* 13: 1613–1633.

Knuth, D. E. 1997. *The art of computer programming. Volume I: Fundamental algorithms*. Reading, MA: Addison-Wesley.

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

Ryan, P. 1999. dm71: Calculating the product of observations. *Stata Technical Bulletin* 51: 3–4. In *Stata Technical Bulletin Reprints*, vol. 9, 45–48. College Station, TX: Stata Press.

—. 2001. dm87: Calculating the row product of observations. *Stata Technical Bulletin* 60: 3–4. In *Stata Technical Bulletin Reprints*, vol. 10, 39–41. College Station, TX: Stata Press.

Schwartzman, S. 1994. *The words of mathematics: an etymological dictionary of mathematical terms used in English*. Washington, DC: Mathematical Association of America.

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*.