The Stata Journal

The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

For more information on the *Stata Journal*, including information for authors, see the webpage

http://www.stata-journal.com

The *Stata Journal* is indexed and abstracted in the following:

- CompuMath Citation Index®
- Current Contents/Social and Behavioral Sciences®
- RePEc: Research Papers in Economics
- Science Citation Index Expanded (also known as SciSearch®)
- Scopus™
- Social Sciences Citation Index®

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata, Mata, NetCourse, and Stata Press are registered trademarks of StataCorp LP.

The Stata Journal (2011)
11, Number 3, pp. 460–471

Speaking Stata: Fun and fluency with functions

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

Abstract. Functions are the unsung heroes of Stata. This column is a tour of functions that might easily be missed or underestimated, with a potpourri of tips, tricks, and examples for a wide range of basic problems.

Keywords: dm0058, functions, numeric, string

1 A column on functions

In Stata, functions in the strict sense take zero or more arguments and return single results. They are documented in [D] **functions** and the corresponding help. Examples are `runiform()`, `ln(42)`, and `strpos("Stata", "S")`. The `()` syntax is generic, even for functions that take no arguments. Think if you like of an open mouth expecting to be fed.

Stata is a command-driven language, which makes it easy to underestimate the value of knowing functions thoroughly. There are functions you know you need and functions you do not know you want. The aim of this column is to tell you more about the latter. The column includes only some highlights, not the complete tour. So if you know you need date, trigonometric, hyperbolic, gamma, density functions, and so forth, it is mostly just a matter of finding out the syntax. An essay on functions (Cox 2002) was one of the earliest *Speaking Stata* columns, but much remains to be said.

If you are a new Stata user, please note that in Stata, commands are not considered to be functions. `list` and `regress`, say, are to be called Stata commands, not functions, regardless of terminology elsewhere.

2 The big picture

2.1 Functions in Stata

A few broad general comments will sketch out the terrain.

Functions can be more useful than you think. Users often seek commands or imagine they need programs when a few functions will crack the problem.

Stata is not knee-deep in functions. The intent is rather to provide a core of really useful functions. Often you need to combine functions to get what you want. This simplicity is really a feature!

Arguments and results of functions can be variables, calculated observation by observation. This is sometimes overlooked; people plan loops over observations when `generate` or `replace` will do that automatically.

Stata functions, in the strict sense, cannot be written by users. You can write functions in Mata, and you can write `egen` functions, although those are functions in different senses of the term.

Functions cannot be called by themselves; you must use commands to assign or display their results. Writing `ln(42)` by itself is not recognized in Stata, but you can type `display ln(42)` to see the result.

In learning functions, `display` and `graph` are your friends. If you are unclear what a function does, use `display` on a few examples.

Is `log()` logarithm to base 10 or to base e? A quick trial settles the matter:

```
. display log(10)
2.3025851
```

Is the square root function `sqr()` or `sqrt()`?

```
. di sqr(10)
Unknown function sqr()
r(133);

. di sqrt(10)
3.1622777
```

Functions unfamiliar to you often make more sense if you draw a graph using `twoway function` (Cox 2004a):

```
. twoway function clip(x,0.2,0.8)
. twoway function chop(x,1), ra(0 10)
. twoway function sin(x)/x , ra(0 `=20 * _pi´)
```

2.2 Strategies and tactics

- *Divide and conquer.* Often you need one function used repeatedly, or two or more functions working together. So hypergeometric probabilities do not need a separate function, just repeated calls to `comb()`:

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

  Combining functions is particularly common with string problems.

- *Nest function calls.* Feed the results of one function to another, as with `exp(rnormal())` to get a random sample from a lognormal distribution or with

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

to get a beta function (beta integral, some say). Cut down on middle macros or middle variables.

Remember that just as in algebra, every left parenthesis, `(`, is a promise to write its match, `)`, sooner or later.

- *Help yourself with layout.* Spaces after commas and around operators often make your code much more readable.
- *Use the documentation.* `help` *fname*`()` gets you straight to the help file for a function *fname*`()`.

3 Some functions with wide scope

Most functions we will look at in this column concern numeric problems only or string problems only, and that will be our main subdivision. To begin, however, we will look at six functions that transcend the numeric or string distinction.

3.1 Numbers trapped in strings, and vice versa

`real()` and `string()` are the workhorse functions when you know you want to change from one to the other form. Always remember that `string()` can take a format argument, too.

Some users use `destring` or `tostring` (see [D] **destring**) with the `force` option to change single variables. That will work but is backward: `destring` and `tostring` are just elaborate wrappers for those functions. If you know you want to force the conversion, you should call the appropriate function directly.

3.2 Depending on conditions

`cond(`*a*`, `*b*`, `*c*`)` returns *b* if *a* is true (nonzero) and *c* if *a* is false (zero). Results can be either numeric or string. For a tutorial, see Kantor and Cox (2005).

Is a given year a leap year? This calls for, or shows off, how calls to `cond()` can be nested:

```
cond(mod(year, 400) == 0, 1,
cond(mod(year, 100) == 0, 0,
cond(mod(year, 4)   == 0, 1,
                                0)))
```

If an expression involving `cond()` is complicated, use a text editor that checks matching parentheses. For this problem, there are other solutions, such as `missing(mdy(2, 29, year))`. Stata's date functions naturally understand leap years.

3.3 In list or in range?

`inlist()` and `inrange()` are a useful pair of functions (Cox 2006b).

`inlist(`*z*`,` *a*`,` *b*`,...)` returns `0` or `1`: `1` if `z ==` *a* | `z ==` *b* |... and `0` otherwise.

There are limits on how many arguments can be supplied, but `inlist()` can be a convenient shortcut for moderately simple problems—such as whether `rep78` is 3, 4, or 5, which is `inlist(rep78, 3, 4, 5)`.

I am embarrassed at how long it took me to realize that

```
inlist(1, a, b, c, d, e)
```

is a solution for

```
a == 1 | b == 1 | c == 1 | d == 1 | e == 1
```

My guess is that this was harder to see because mathematical convention makes it much more likely that we write and think in terms of the line above rather than the exact equivalent

```
1 == a | 1 == b | 1 == c | 1 == d | 1 == e
```

Either way, the choices here are aesthetic or cosmetic: the long-winded ways of writing will work fine and may even be clearer to some tastes.

`inrange(`*z*`,` *a*`,` *b*`)` returns `0` or `1`: `1` if $a \leq z \leq b$ and `0` otherwise. There are special rules for missing arguments. Clearly, the inequalities must match your problem.

`inrange(`*char*`, "a", "z")` and `inrange(`*char*`, "A", "Z")` are tests of whether *char* is, respectively, a lowercase or uppercase letter. It is easy to overlook the fact that inequalities may be applied to string values. The associated order is naturally identical to the order yielded by the `sort` command.

Often it is easier in a problem to exclude subsets, which is simplest by logical negation, yielding expressions of the form `if !inlist(`*arguments*`)` or of the form `if !inrange(`*arguments*`)`. Learn to think in terms of "if not in list", and so forth. This is cleaner and arguably easier to decode than (say) `if inlist(`*arguments*`) == 0`.

3.4 Anything missing?

With a single numeric variable, `if x < .` is the cleanest way to exclude missings, as was pointed out by Lachenbruch (1992). `if x != .` is fine so long as there are no extended missing values `.a`, ..., `.z`.

For more complicated problems, you should turn to `missing()` (Rising 2010). `missing(`*x1*`,` *x2*`, ...,` *xn*`)` returns `1` if any of its arguments is missing and `0` otherwise. `missing()` covers numeric or string arguments or even a mixture. `!missing()` reverses results.

If your concern is with numeric variables only, a statistical shortcut is to use `regress` and then `e(sample)` or its negation. The estimation sample after `regress` will automatically be only those observations with nonmissing values on all the variables specified.

4 Functions for numeric problems

Let's turn now to essentially numeric problems.

4.1 abs(), sign(), and mod()

`abs(`x`)` is often overlooked when $|x|$ is wanted. People might write

```
if t > 2 | t < -2
```

when they could write more cleanly, and with less risk of error,

```
if abs(t) > 2
```

`sign(`x`)` returns `-1`, `0`, `1`, and `.` for negative, zero, positive, and missing arguments.

`mod(`x`, `y`)` is a very versatile function. By a standard abuse of terminology, `mod()` returns the remainder, not the modulus as mathematicians know it. Thus, what is leftover on division of x by y? Such abuse has been common in programming since the 1950s and goes back at least as far as the first version of FORTRAN. A simple example is whether the observation number is odd: `if mod(_n, 2) == 1` or `if mod(_n, 2)`.

`mod()` has already received its own small song of praise (Cox 2007a). That article omitted rotations, such as a rotation 90° clockwise, whose result is given by `mod(angle + 90, 360)`.

4.2 logit() and invlogit()

`logit(`x`)` and `invlogit(`x`)` are also often neglected. You might prefer to write out the definitions from scratch, but that would create horrible bugs if your solutions were legal but incorrect.

4.3 Rounding

Stata has a bundle of functions for rounding up and down. I have an irrational fondness for the first two here (Cox 2003).

`ceil(`x`)` rounds up always (think: "ceiling"), so `ceil(1.2)` is 2. A neat way to get uniformly distributed integers, 1(1)10, is `ceil(10 * runiform())`. Compare `1 + int(10 * runiform())`. Apply "for any value of 10".

`floor(`x`)` rounds down always, so `floor(1.8)` is 1. `if x == floor(x)` is one of several tests of whether `x` is integer.

`int(`x`)` rounds toward zero always; that is, it rounds up for negative numbers. `int(1.2)` is 1 and `int(-1.2)` is -1. `trunc(`x`)` is a synonym for `int(`x`)`.

`round(`x`)` rounds to the nearest integer. `round(x, y)` rounds to the nearest multiple of `y`.

A common misconception, however, is confusing numeric rounding (a calculation matter) with display to so many decimal places (a presentation matter). Although `round(1.23, 0.1)` may look like 1.2, it cannot be 1.2 exactly; as with most decimal numbers with fractional parts, 1.2 has no exact binary equivalent. If you want to show so many decimal places, use the correct format, not `round()`. For much fuller explanations, see Cox (2006a), Linhart (2008), and especially Gould (2006, 2011a, 2011b, 2011c, 2011d).

4.4 Binning

`autocode()`, `irecode()`, and `recode()` are functions for binning a range into contiguous intervals. See also `egen`'s `cut()` function. Some people use `recode` (see [D] **recode**) for binning continuous variables. In all cases, there is an overarching caution: you need to check boundary rules carefully. If a value is exactly on some boundary, is it assigned to the lower or upper bin?

`2 * floor(myvar/2)` illustrates a simpler device. In this example, `myvar` is binned with a width of 2, and the lower bound is inclusive, so bins would be like $[0, 2)$ and $[2, 4)$. Using `ceil()` instead means that the upper bounds are inclusive. I find it easy to remember what happens with these rules and also find that these two functions solve almost all binning problems, but naturally your experience may differ.

That said, such binning might be called histogram style, in which we are careful to specify lower and upper limits for each. Occasionally, you want what might be called binning scatterplot style, in which bins are represented by their centers. For this, `round()` is helpful. `round(x, 2)` will produce bins with width 2 and with centers like 0 or 2. Inspection shows that these bins are like $[-1, 1)$ or $[1, 3)$ so that lower limits are inclusive.

4.5 Going to extremes

The extreme functions `max()` and `min()` are, by and large, what you would expect. But a key detail is that missings are ignored unless all arguments are missing. This is usually a feature.

There are work-arounds when you want any missing to trump nonmissings. One is

```
cond(missing(a, b), ., max(a, b))
```

To spell this out: if either `a` or `b` is missing, return missing; otherwise, return the larger of `a` and `b`.

If you have several arguments, it can be easier to use a loop to get the maximum or minimum. Suppose you want a rowwise maximum across a bundle of variables whose names are held in a local called `varlist`, and set aside the fact that there is a dedicated `egen` function for this problem.

```
generate max = .
foreach v of local varlist {
        replace max = max(max, `v´)
}
```

The initialization to missing is safe, even though numeric missing (`.`) is in most contexts treated as larger than any nonmissing value. As said, `max()` returns the largest nonmissing argument supplied, unless all arguments are missing. For a broader review of working rowwise, see Cox (2009).

Naturally, the existence of the `max()` and `min()` functions does not affect the fact that if you have a variable, you should use `summarize, meanonly` to get the extremes (Cox 2007b).

What about records, such as the maximum so far, and within panels too? As in the loop just given, the maximum is, recursively, the largest of the previous maximum and the present value, but calculated with respect to observations, not variables. Similar code will get you the minimum, and `by:` makes it easy to do this within groups.

```
. generate record = .
. by id (time), sort: replace record = max(record[_n-1], y)
```

4.6 Any or all?

There is a "Yes, of course" relationship between `max()` and `min()` and any or all problems. "Is any?" and "Are all?" questions are often easily solved through the corresponding function.

To spell this out: if any variable argument *arg* is 1 (true) or 0 (false), then there is a correspondence:

`min(`*arg*`) == 0`	some false
`min(`*arg*`) == 1`	all true
`max(`*arg*`) == 1`	some true
`max(`*arg*`) == 0`	all false

These relations are often useful. Note in passing that `egen`'s `min()` and `max()` functions give easy ways to apply this principle in conjunction with `by:` or `by()` to panels or other groups, such as families. See http://www.stata.com/support/faqs/data/anyall.html for discussion.

4.7 Some sums

Stata's `sum()` function returns running or cumulative sums. Perhaps it should have been called, say, `cusum()`. Like `max()` and `min()`, `sum()` ignores missings. However, unlike those functions, `sum()` returns `0` if all of its arguments are missing.

Using `egen`'s `total()` function is more direct—but less efficient—to put group totals in a variable.

Make sure you know this two-step process:

```
. by id, sort: gen mysum = sum(myvar)
. by id: replace mysum = mysum[_N]
```

After the first command, the last observation in each group contains its group total. In the second command, this is copied to all observations in each group. This two-step process is essentially what `egen`'s `total()` does too.

The `sum()` function can be used for many other problems, including problems that at first sight are counting problems rather than summation problems. The number of distinct values of `x` seen so far within panels is given by

```
. by id x (time), sort: gen distinct = _n == 1
. by id (time), sort: replace distinct = sum(distinct)
```

Here the technique uses an indicator variable: we tag each distinct occurrence by 1 and then sum the 1s. The trickiest detail is getting the correct sort order first.

5 Functions for string problems

We will now turn to string problems.

5.1 An outstanding string quartet

Many data-management questions—including clean-ups of data—need string functions, often in combination. My top four string functions—those which every experienced Stata user should know—are `strpos()`, `substr()`, `subinstr()`, and `length()`.

But before we get to those, there is a key principle: Stata's string operations are utterly literal, so many tests should be phrased in terms of one case or consistent leading, trailing, and internal spaces. You can do that in one phrase with, say,

```
lower(trim(itrim(myvar)))
```

`strpos(`*s1*`,` *s2*`)` tells you where *s2* occurs in *s1* and tells you 0 if it does not occur. Think `string position`. An immediate corollary is that `if strpos(`*s1*`,` *s2*`) > 0` or (in brief) `if strpos(`*s1*`,` *s2*`)` is a true-or-false test of whether *s1* contains *s2*. Thus

```
    strpos("this", "is") = 3
    strpos("this", "it") = 0
    strpos("haystack", "needle") = 0
```

In older versions of Stata, this function was called `index()`.

Commonly, *s1* is a string variable name. Less commonly, *s2* is also a string variable name. In either case, the value of the variable is used, not the literal name.

`substr(`*s*`,` *pos*`,` *len*`)` gives the substring of *s* starting at position *pos* and of length *len*. *pos* can be negative, in which case it indicates position counted backward from the end of the string. *len* can be numeric missing (.), in which case it indicates everything else in the string.

```
    substr("abcdef", 2, 3) = "bcd"
    substr("abcdef", -3, 2) = "de"
    substr("abcdef", 2, .) = "bcdef"
```

`subinstr(`*s1*`,` *s2*`,` *s3*`,` *n*`)` changes the first *n* occurrences in *s1* of *s2* to *s3*. As an important special case, if *s3* is the empty string `""`, then occurrences of *s2* are deleted.

```
    subinstr("this is this", "is", "X", 1) = "thX is this"
    subinstr("this is this", "is", "X", 2) = "thX X this"
    subinstr("this is this", "is", "X", .) = "thX X thX"
```

`length(`*s*`)` returns the length of *s*. *s* can be a string variable name, in which case you get the length of its contents. Whereas `length("ab")` is 2, `length(myvar)` could vary between observations. `length("myvar")` is just 5.

5.2 Other leading string players

Some other string functions worth flagging are `char()`, `reverse()`, and the regular expression (regex) functions.

`char(`*n*`)` returns ASCII character *n* and so is one way of displaying otherwise unprintable characters (Cox 2004b). For a convenient display, download `asciiplot` from the Statistical Software Components (SSC) archive by using the `ssc` command.

To work backward, for example, to change the last occurrence of a substring, consider reversing and finally reversing back with `reverse(`...`(reverse(`...`)`...`)`.

Stata has a suite of regular expression functions: `regexm()`, `regexr()`, `regexs()`, and `strmatch()`. Be aware that it is often the case that people imagine a regex solution is required when one or more of the basic functions would suffice. The most complete documentation is at http://www.stata.com/support/faqs/data/regex.html.

5.3 Counting occurrences of substrings

Let's switch to a problem rather than a function: counting (disjoint) occurrences of strings (Cox 2011). For example, how many occurrences of `X` are there in

```
    "OOOOXXXOOXXX"
```

Many users store short histories (244 periods or less) in string variables, say, for holding recent histories of employment or obstetric status. Here is one way:

```
length(myvar) - length(subinstr(myvar, "X", "", .))
```

Take this in steps:

1. Get the length of `myvar`.
2. Get the length of `myvar` with all `X` deleted. (You don't have to carry out that deletion, just to find out what the resulting length would be.)
3. The difference is what you want.

This generalizes easily to longer substrings; you just need to remember also to divide by length of substring, because you want to count occurrences.

Remember that some operations on substrings are easier after `split` (see [D] **split**).

5.4 Removing the first word

There is more than one way to remove the first word. First, words are separated by spaces, so look for the first space:

```
trim(substr(myvar, strpos(myvar, " "), .))
```

This works, perhaps fortuitously, if there is no space present, because `strpos()` then returns `0` and `substr()` then returns `""`.

Alternatively, use a dedicated function. `word()` selects individual words:

```
trim(subinstr(myvar, word(myvar, 1), "", 1))
```

In both cases, we applied `trim()` last.

Also, `egen`'s `ends()` function can do this with its `tail` option.

5.5 Cleaning up species names (binominals)

The proper form for species names is that genus is capitalized, but not species: for example, *Homo sapiens*, *Homo economicus*, and *Troglodytes troglodytes*.

```
. generate species2 = upper(substr(species, 1, 1)) + lower(substr(species, 2, .))
```

The `proper()` function capitalizes each word, which is not what we want.

Suppose we want the first two words only, ignoring taxonomic detail like `(Linnaeus, 1758)`:

```
. replace species2 = word(species2, 1) + " " + word(species2, 2)
```

5.6 Filler text

Stata lacks a function quite like `rep("X", 80)` to replicate strings, but there are other ways to do it. For example,

```
. local text : di _dup(80) "X"
. mata : st_local("text", 80 * "X")
```

are two possibilities. Alternatively, with a supply of filler, you can add as much as you want with

```
substr("`text´", 1, length)
```

where `length` could vary between observations. Finally, you could always type repeated text yourself.

6 Conclusion

Commands are the leading characters in Stata dramas, but commands depend on functions to do their work. Stata's documentation of functions in help files and manuals is perhaps its driest part. This rapid tour of some key functions has aimed to publicize tools likely to be useful in your day-to-day work with Stata, whether in interactive sessions, do-file writing, or programming.

7 Acknowledgments

Stephen Jenkins and Roger Newson made helpful comments on an earlier version of this column. William Gould underlined the use of `round()` for binning.

8 References

Cox, N. J. 2002. Speaking Stata: On getting functions to do the work. *Stata Journal* 2: 411–427.

———. 2003. Stata tip 2: Building with floors and ceilings. *Stata Journal* 3: 446–447.

———. 2004a. Stata tip 15: Function graphs on the fly. *Stata Journal* 4: 488–489.

———. 2004b. Stata tip 6: Inserting awkward characters in the plot. *Stata Journal* 4: 95–96.

———. 2006a. Stata tip 33: Sweet sixteen: Hexadecimal formats and precision problems. *Stata Journal* 6: 282–283.

———. 2006b. Stata tip 39: In a list or out? In a range or out? *Stata Journal* 6: 593–595.

———. 2007a. Stata tip 43: Remainders, selections, sequences, extractions: Uses of the modulus. *Stata Journal* 7: 143–145.

———. 2007b. Stata tip 50: Efficient use of summarize. *Stata Journal* 7: 438–439.

———. 2009. Speaking Stata: Rowwise. *Stata Journal* 9: 137–157.

———. 2011. Stata tip 98: Counting substrings within strings. *Stata Journal* 11: 318–320.

Gould, W. 2006. Mata Matters: Precision. *Stata Journal* 6: 550–560.

———. 2011a. How to read the %21x format. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/02/02/how-to-read-the-percent-21x-format/.

———. 2011b. How to read the %21x format, part 2. The Stata Blog: Not Elsewhere Classified.
http://blog.stata.com/2011/02/10/how-to-read-the-percent-21x-format-part-2/.

———. 2011c. Precision (yet again), Part I. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/06/17/precision-yet-again-part-i/.

———. 2011d. Precision (yet again), Part II. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/06/23/precision-yet-again-part-ii/.

Kantor, D., and N. J. Cox. 2005. Depending on conditions: A tutorial on the cond() function. *Stata Journal* 5: 413–420.

Lachenbruch, P. A. 1992. ip2: A keyboard shortcut. *Stata Technical Bulletin* 9: 9. Reprinted in *Stata Technical Bulletin Reprints*, vol. 2, p. 46. College Station, TX: Stata Press.

Linhart, J. M. 2008. Mata Matters: Overflow, underflow and the IEEE floating-point format. *Stata Journal* 8: 255–268.

Rising, B. 2010. Stata tip 86: The missing() function. *Stata Journal* 10: 303–304.

About the author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also coauthored 15 commands in official Stata. He wrote several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.