The Stata Journal

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

The Stata Journal (2005)
5, Number 3, pp. 413–420

Depending on conditions: a tutorial on the cond() function

David Kantor
kantor.d@att.net

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

Abstract. This is a tutorial on the `cond()` function, giving explanations and examples and assessing its advantages and limitations.

Keywords: pr0016, cond(), functions, if command, if qualifier, generate, replace

1 Introduction

Stata functions, like functions in any similar language, fall on a continuum, from those you know you want to those you do not know you need. If you want a logarithm, a square root, or some probability function, the only small difficulty is likely to be checking the exact syntax Stata uses: Is `log()` equivalent to `ln()` or `log10()`? How do I get tail probabilities for a Gaussian? What is more problematic is finding out about functions that might be useful and should thus be added to your personal toolkit. The help files and the manual entry [D] **functions** are admirably terse and precise but lack detailed examples making clear how functions can be exploited in practice. In several cases, the issue is not so much understanding the definition, but more appreciating how a particular function might be helpful in the future.

The function `cond()` is a case in point. The online help gives its formal definition. Here only the simpler of the two forms allowed is examined. In this tutorial, we will not ever get to the more complicated form, as we can do plenty without it.

`cond(`*x*`,`*a*`,`*b*`)` returns *a* if *x* evaluates to true (not 0) and *b* if *x* evaluates to false (0).

In abstraction, the idea is that of producing different results—*a* or *b*—depending on whether a specified condition—*x*—is true or false. The results can be both numeric or both string, and—depending on context—variables or single values. As with many formal definitions, this may not impart a strong sense of precisely how useful `cond()` could be. The purpose of this tutorial is to provide such a sense. On the way, we will make comparisons with how else various problems might be solved.

2 Simple examples

Let us make that definition concrete immediately with some simple examples.

1. If `a` and `b` are numeric variables, `cond(a > b, a, b)` returns the larger of `a` and `b`. Given a little knowledge of Stata's functions, your reaction is likely to be that

`max(a,b)` will do exactly the same. But you are not quite right. If one of `a` or `b` is missing, let us say `b`, then `max(a,b)` returns `a`, but `cond(a > b, a, b)` returns missing. Either might be precisely what you want.

2. With the same variables, `cond(a >= 42, 42, a)` returns the smaller of 42 and `a`. This time there is no trap: `min(a,42)` will do that job, too, regardless of missing values.

3. With `a` again numeric, and `a` not missing, `cond(a >= 0, a, 0)` and `cond(a < 0, -a, 0)` return the positive and negative parts of `a`, that is, a_+ and a_- such that $a = a_+ - a_-$. This too you could achieve via `max(a,0)` and `-min(a,0)`.

4. With probabilities `p` and a desire to calculate entropy, defined as the sum of terms $-p \ln p$, you need to tell Stata to respect the convention that $-0 \ln 0$ is evaluated as 0. This could be done as `cond(p == 0, 0, -p * ln(p))` to override Stata's understandable belief that `ln(0)` is indeterminate and so must be reckoned as missing.

5. With `a` and `b` now string variables, `cond(a < b, a, b)` returns whichever of the strings is earlier in alphanumeric sort order. You cannot do that so easily with other functions.

3 The sales pitch

The advantages of `cond()` include

Conciseness. `cond()` allows the use of one line for what might otherwise take two (or more, as we shall see later). Consider how to do (5) to produce a new string variable using the `if` qualifier. We need two commands, one for each `if` condition. The first must be a `generate` command, and the second must be a `replace` command:

```
generate first = a if a < b
replace first  = b if a >= b
```

That is more long-winded—and a little more error-prone, as the case of the two strings being identical must be included. If you were to omit the case of `a` equal to `b`, the corresponding observations would end up with missing values (empty strings) in the new variable `first`. You may or may not feel more comfortable with this solution than with

```
generate first = cond(a < b, a, b)
```

We should perhaps stress that assignment using a `generate` command for one subset followed by `replace` commands for other subsets is inevitably much more general than a call to `cond()`. `generate` and `replace` can involve calls to other functions, and a `replace` statement can revisit observations modified earlier.

Either way, and in other situations, there is an overarching question: when you revisit your own code, or when someone else has to maintain, modify, and debug your code, which version would be preferred?

Admittedly, `cond()` is not always more concise than alternatives, as is shown by (2) and (3).

Generality. Other functions tend to do precisely one kind of thing, but `cond()` has some generality. Note, in particular, how `cond()` can produce string results, as well as numeric. A general tool that you can use in different problems should appeal as well as specific tools more restricted in application.

Control. You are in charge and get to say precisely what the results are. This is usually attractive, and particularly so when a standard function is not what you want, or even not quite what you want. (4) and (5) are simple examples.

That is the sales pitch. Let us push it harder by giving several more examples.

4 Do-it-yourself categorization

A common application of `cond()` is categorization of a variable into a fixed number of categories, based on your own class definitions.

Suppose that with the `auto` data, you decide to categorize `weight` into three classes: low, medium, and high. A glance at the data suggests divisions at 2,500 pounds and 4,000 pounds. Three or indeed more classes is no problem with `cond()`, as the trick is to nest function calls. The expression to feed to a `generate` command could be

```
cond(weight < 2500, 1,
cond(weight < 4000, 2,
cond(weight < .,    3,
                    .
                     )))
```

This is far from the only way to nest calls to `cond()`, but it is simple once understood and widely applicable. We have laid the code out in one way that shows the structure more clearly than a more compressed display. There are others, but some tidy layout is strongly recommended.

As in elementary algebra, and in other situations in Stata, putting down a left parenthesis is a promise that you will put down a matching right parenthesis later. If you break your promise, Stata will complain. To make sure you have balanced parentheses in such expressions, exploit the pertinent function in a decent text editor. In the Stata do-file editor, it is `Balance`, *Ctrl-B*; in Vim, it is the `%` key; and so forth. If your text editor has no facility to find and show the matching parenthesis, it is not decent. You need something better.

In practice this kind of layout would mean, especially if the code appeared in a do-file or program, either using semicolons as line delimiters or commenting out end-of-lines:

```
#delimit ;
cond(weight < 2500, 1,
cond(weight < 4000, 2,
cond(weight < .,    3,
                    .
                     ))) ;
#delimit cr
```

or

```
cond(weight < 2500, 1, ///
cond(weight < 4000, 2, ///
cond(weight < .,    3, ///
                    .  ///
                     )))
```

Some favor one style; some favor the other. But experience does teach, sometimes bitterly, that identifying a neat layout that works for you and then following it is much better technique than jamming all the code into the smallest possible space.

A detail here worth flagging is that we took care of missing values explicitly. There are other ways to do that, but a little thought on how missing values are handled can save much puzzlement later. But that is true more generally.

There are, not surprisingly, other ways of carrying out multiple categorization. If your subdivision is into regular intervals, `floor()` or `ceil()` provides a more systematic approach (Cox 2003). Other customized approaches are possible through the `recode()` and `irecode()` functions and the `cut()` function of `egen`.

A solution using `cond()` has some simple advantages. Complete control can be maintained over what is done. A program or log file with the categorization code is pretty well self-documenting so that, in particular, equalities and inequalities are plain for all to see. Using any of the other functions means that you may have to look up the documentation to see what happens if values equal cutpoints (are values mapped upwards or downwards?), what happens in end classes, and what happens to missing values.

5 cond() in terms of if and else conditions

The main idea of this kind of example, using a series of nested calls to `cond()`, can be spelled out in another way with an unStataish pseudocode, closer to some programming languages. The result is

```
IF weight < 2500      THEN 1
ELSE IF weight < 4000 THEN 2
ELSE IF weight < .    THEN 3
ELSE .
```

The effect is thus one of a cascade: at each step, a subset of cases is peeled off and dealt with (and note that once dealt with, those cases are not revisited within the same command). Anything that falls through all the tests gets caught at the end. In this case, observations that fail all the tests including `weight < .` evidently are missing on `weight`. The last branch insists that missing values of `weight` are mapped to missing in whatever is produced by this code fragment.

5.1 A digression on the if command

We should note in passing that code such as

```
if weight < 2500  {
...
}
else if weight < 4000 {
...
}
else if weight < . {
...
}
```

is legal in Stata, but almost never will it be what you want whenever your conditions involve variables. Stata's `if` command, in short, is quite different from Stata's `if` qualifier. Code like those statements will get interpreted as referring to the very first observation in the current sort order, that is, as if you had written

```
if weight[1] < 2500
```

and so forth. Not realizing this will produce, almost always, a puzzling bug. On the other hand, if you really do want testing in terms of the first observation, then it is better style to make that explicit, as even experienced Stata users can be puzzled by such code. If you want testing in terms of any other observation, say the last, then you really must make that explicit.

To put it another way, Stata's `if` command does not support vectorized calculations according to which observations yield true and which yield false. It supports only branching according to single values, or more precisely expressions yielding single values, evaluated as true or false. Other software does support vectorized calculations with their equivalent of the `if` command, which naturally creates expectations when their users first make the transition to Stata. Right or wrong, this lack of support does provide an extra rationale for `cond()`.

6 Leap years

How do you determine if a year is a leap year using Stata syntax? You know that 2004 was a leap year, and 2008 will be one, but how would you handle leap years generally? The rules in the Gregorian calendar for a year to be a leap year are

```
YES if year divisible by 4
(but NO if year divisible by 100
(but YES if year divisible by 400))
and NO otherwise.
```

Note the nesting of rules, as shown by our parentheses. If a leap year is a first-order correction, the third rule is an example of a third-order correction. Scientists and engineers often use such ideas, but they seem in shorter supply in the everyday world.

You will be familiar with the divisibility-by-4 rule. The other rules are occasionally forgotten, although the recent millennium year 2000 provided an example. For example, Excel has 1900 as a leap year; it is documented that this was to provide compatibility with Lotus 1-2-3.

In Stata, suppose that `year` is a variable. You could cheat by exploiting the fact that Stata's programmers have solved this problem already, and for completeness we will mention some such solutions, but we should look at ways of doing it from first principles.

An indicator containing 1 for leap year and 0 otherwise is given by

```
(mod(year,4) == 0 & mod(year,100) != 0) | mod(year,400) == 0
```

as `mod()` provides the remainder left over from division. You will expect a solution with `cond()`, and it shows once more how `cond()` expressions can be nested:

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

Note that we have to do it backwards: starting with

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

would not allow us to peel off the observations for which `mod(year,100) == 0`: all those are already dealt with by `mod(year, 4) == 0`.

Note that if (for example) `mod(year,400)` is equal to 0, then its logical negation will have value 1 and so evaluate as true. Thus we can abbreviate to

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

with the price, for some readers, of making the code more cryptic. `!` is now the approved official Stata way of indicating logical negation, but using the tilde remains perfectly legal if that is more to your taste.

Note, however, that missing values of `year` would fall through this set of calls with the result that the years would be declared nonleap years. This is more likely correct than not, but not necessarily what you would want. Trapping with an overall `if year < .` is the safest recommendation.

This is another example in which laying out code carefully helps understanding. Stata is indifferent to layout, so long as your code is legal, so this is entirely a matter of human communication. However, one of these humans could easily be yourself at various different times. Although neither is explicit here, use of semicolons as delimiters or of commenting out end-of-lines would again be a good idea.

6.1 How to do it with date functions

How to cheat? Given daily dates, a feature of a leap year is clearly that there is a February 29, so

```
mdy(2,29,year) < .
```

or

```
mdy(2,29,year) != .
```

is true. That is, `mdy(2,29,year)` is missing if `year` is not a leap year but not otherwise; in the latter case, it is less than missing. You could also do that as

```
!mi(mdy(2,29,year))
```

Similarly, there are 366 days in a leap year, so

```
doy(mdy(12,31,year)) == 366
```

is true whenever `year` is a leap year. What works with variables will also work with individual years, such as `mdy(2,29,2008) < .` or `doy(mdy(12,31,2008)) == 366`.

7 Duplicate dyads

In our initial set of examples, where did (5) come from? It grew out of a problem on Statalist. A user had interaction data of some kind for pairs of objects: people in some fields talk of 'dyads'. The rules of the game meant that an interaction between an object with name a and one with name b is equivalent to one the other way round. In short, the relation is symmetric. Thus an observation with values `"a"` and `"b"` on two variables is considered as equivalent to another with values `"b"` and `"a"`. `duplicates` (see [D] **duplicates**) and other ways of dealing with duplicates would have a hard time dealing with this.

One solution is to sort each pair of values observation-wise:

```
gen first = cond(a < b, a, b)
gen second = cond(a < b, b, a)
```

and then look at duplicates, or groups, in terms of `first` and `second`. In this way, an awkward problem is rendered trivial by exploiting two Stata facts: support for string results from `cond()` and the fact that inequalities make sense for strings.

A more neurotic solution would check for the use of upper and lower case or of leading and trailing spaces in strings, and for equalities (`a` equal to `b`), whether or not acceptable, but such points are secondary to the theme here.

8 The other side

Earlier you had the sales pitch for `cond()`. To be fair, we should underline what else should have been said in broad terms.

Alternatives. There are often different ways to say things, some involving `cond()` and others avoiding it. Stata is not a minimalist language dedicated to the idea that there is only one way to say something, and it allows different solutions to the same problem and a variety of styles. Although the multiplicity of functions can be confusing to beginners, it is ultimately a source of strength.

Readability. This is the bigger issue. `cond()` can feature in very complicated expressions, which has given it a reputation for difficulty. Supporters of `cond()` argue that complicated problems tend to require complicated solutions and urge that much depends on layout and learning to read examples. Detractors of `cond()` tend to avoid it. It is often true that a `generate` statement followed by one or more `replace`s is easier to read, to understand, to debug, and to maintain, and such code avoids lots of irritating parentheses.

Without dogmatically insisting on `cond()` whenever possible, we remain very positive about its merits.

9 Conclusion

`cond()` is versatile and not as tricky as you might fear from some of the more extreme examples of its use. Consider adding it to your Stata repertoire.

10 References

Cox, N. J. 2003. Stata tip 2: Building with floors and ceilings. *Stata Journal* 3(4): 446–447.

About the Authors

David Kantor worked for some years as a data analyst at the Institute for Policy Studies at Johns Hopkins University with a team that researched the effects of housing policy. He is now an independent worker. He has placed several of his Stata programs in the public domain, contributed code to the official Stata command `cf`, and made many postings to Statalist.

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 co-authored fifteen commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is an Editor of the *Stata Journal*.