The Stata Journal (2019)
19, Number 3, pp. 719–728 DOI: 10.1177/1536867X19874247

Speaking Stata: The last day of the month

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

Abstract. I discuss three related problems about getting the last day of the month in a new variable. Commentary ranges from the specifics of date and other functions to some generalities on developing code. Modular arithmetic belongs in every Stata user's coding toolbox.

Keywords: dm0100, dates, days, weeks, months, functions, modulus, remainders, rotations

1 Introduction

Given a monthly date variable in Stata, people sometimes want the last day of each month as a new daily date variable. This problem has been touched on in a previous tip (Samuels and Cox 2012), but the title of that tip may not make its relevance to this question sufficiently evident.

In this column, I will examine that problem and two related problems. I also attempt to distill some coding morals that lie behind the problems and their solutions. I assume you know enough about date variables to understand that monthly dates and daily dates are held in different ways. If you do not, reading `help datetime` now might be a good idea.

2 Have monthly dates; seek last daily date of each month

Scrutiny of `help datetime functions` does not reveal a dedicated function, so it seems that you may have to write code yourself. Your heart may sink at this as you ponder: some months have 31 days, some 30, and then there is February, which has 29 days in a leap year and 28 otherwise. So, do we have to code not just for those different month lengths, but also for whether a year is leap or not?

Let us make a sandbox dataset to work on:

```
. set obs 10
number of observations (_N) was 0, now 10
. generate mdate = cond(_n <= 5, _n - 1, _n + 6)
. format mdate %tm
. list

     ┌─────────┐
     │   mdate │
     ├─────────┤
  1. │  1960m1 │
  2. │  1960m2 │
  3. │  1960m3 │
  4. │  1960m4 │
  5. │  1960m5 │
     ├─────────┤
  6. │  1961m1 │
  7. │  1961m2 │
  8. │  1961m3 │
  9. │  1961m4 │
 10. │  1961m5 │
     └─────────┘
```

Do you see what we did there? The sandbox has some months for 1960 (a leap year) and some for 1961 (not a leap year), so we can test our solutions on different months and the two cases for February. Just typing some data into the Data Editor may be quicker than thinking up the small trickery here with _n. I often do that myself as well.

The first trick is this. The last day of the present month is the day before the first day of the next month. If someone mentioned that in conversation, you might wonder if they were being facetious in saying something so obvious, but this identity is the key to a one-line solution of the problem.

With our sandbox dataset, the next month is `mdate + 1`; pushing that through `dofm()` gives the first day of that month, and finally we subtract 1.

```
. generate lastddate = dofm(mdate + 1) - 1
. format lastddate %td
. list

     ┌─────────────────────┐
     │   mdate   lastddate │
     ├─────────────────────┤
  1. │  1960m1   31jan1960 │
  2. │  1960m2   29feb1960 │
  3. │  1960m3   31mar1960 │
  4. │  1960m4   30apr1960 │
  5. │  1960m5   31may1960 │
     ├─────────────────────┤
  6. │  1961m1   31jan1961 │
  7. │  1961m2   28feb1961 │
  8. │  1961m3   31mar1961 │
  9. │  1961m4   30apr1961 │
 10. │  1961m5   31may1961 │
     └─────────────────────┘
```

There is no completely free lunch here. We may still need to use the help for functions to find the function `dofm()`—and to appreciate that it will help. Read that function name as indicating the first day of the current month.

3 Have daily dates; seek last day of current month

A common extra twist is that we have some daily dates in a variable (perhaps meaningful dates, perhaps arbitrary dates) and also wish to have the last day of the corresponding month in a new variable. Given that problem, we just need to convert it to the previous problem, and then we are done. Let us add the 15th of each month to the sandbox. Now the inverse function `mofd()` yields a monthly date, and then we do the same trick:

```
. generate middate = dofm(mdate) + 14
. generate lastddate2 = dofm(mofd(middate) + 1) - 1
. format middate lastddate2 %td
. list middate lastddate*

     +------------------------------------+
     |   middate   lastddate   lastdda~2 |
     |------------------------------------|
  1. | 15jan1960   31jan1960   31jan1960 |
  2. | 15feb1960   29feb1960   29feb1960 |
  3. | 15mar1960   31mar1960   31mar1960 |
  4. | 15apr1960   30apr1960   30apr1960 |
  5. | 15may1960   31may1960   31may1960 |
     |------------------------------------|
  6. | 15jan1961   31jan1961   31jan1961 |
  7. | 15feb1961   28feb1961   28feb1961 |
  8. | 15mar1961   31mar1961   31mar1961 |
  9. | 15apr1961   30apr1961   30apr1961 |
 10. | 15may1961   31may1961   31may1961 |
     +------------------------------------+
```

4 But it must be a Friday (or some other day of the week)

Once a problem is solved, it often looks trivial, so let us try something harder.

Suppose there is an extra constraint—that we want the last Friday in a month. This will stand for all other such problems in which we insist on the last day identified in each month being a particular day of the week. In many fields, specific things happen on particular days of the week. You may know examples in your own field, whether it is economics, epidemiology, ecclesiology, eschatology, or something quite different. Even if you have never met this problem, keep reading for a little extra technique that will help if you do encounter it in the future.

If the code so far does yield a Friday as the last day in that month, then we are done for that month. If it is a Thursday, we need 6 days before; Wednesday, 5 days before; Tuesday, 4 days; Monday, 3 days; Sunday, 2 days; and Saturday, 1 day.

Writing out code for all seven cases would solve the problem. We might hope to find a simpler solution if we can. Done slowly or quickly, we need a function to find the day of the week. Scrutiny of `help datetime functions` finds `dow()`. You may know about that function already. `dow()` yields 0 for Sundays, 1 for Mondays, and so on, until 6 for Saturdays.

Across cultures, countries, and professions, there are many variations on which day of the week is considered first, or equivalently which day is last. Given any particular rule, we can still use `dow()`; we may just need to rotate the results. Calendar-related problems are splendidly variable and capricious. Two encyclopedic references are Blackburn and Holford-Strevens (1999) and Reingold and Dershowitz (2018).

Let us do this slowly and then see if we can find a pattern we can exploit to simplify the code. If the last day of the month is Friday, `dow()` will return 5 and that is good. If the last day of the month is Saturday, `dow()` will return 6 and we need the day before, so subtract 1. If it is Thursday, we need 6 days before; if it is Wednesday, we need 5 days before; and so on.

```
. generate lastfri = lastddate if dow(lastddate) == 5
(9 missing values generated)
. replace lastfri = lastddate - 1 if dow(lastddate) == 6
(1 real change made)
. replace lastfri = lastddate - 6 if dow(lastddate) == 4
(1 real change made)
. replace lastfri = lastddate - 5 if dow(lastddate) == 3
(1 real change made)
. replace lastfri = lastddate - 4 if dow(lastddate) == 2
(3 real changes made)
. replace lastfri = lastddate - 3 if dow(lastddate) == 1
(1 real change made)
. replace lastfri = lastddate - 2 if dow(lastddate) == 0
(2 real changes made)
```

Now if you stare at the code, you should see first that seven lines can be reduced to three lines. If the last day is Friday, we are done; if it is a Saturday, we just subtract 1; otherwise, we can subtract the day of the week as given by `dow()` plus 2.

```
. generate lastfri2 = lastddate if dow(lastddate) == 5
(9 missing values generated)
. replace lastfri2 = lastddate - 1 if dow(lastddate) == 6
(1 real change made)
. replace lastfri2 = lastddate - dow(lastddate) - 2 if lastfri2 == .
(8 real changes made)
```

It is good to be cautious about whether that reduction is correct, so we have tried it both ways. Shortly, we will check to see that we get the same result.

We can reduce the code all the way down to one line. The subtracted correction can be thought of as the combination of two rules into one using `cond()` (Kantor and Cox 2005): if the day of the week is 5 or 6, we subtract day of the week minus 5; otherwise, we subtract the day of the week plus 2. Differently put, the branching comes from a split: either the day of the week is greater than or equal to that desired, or it is less than that desired.

```
. generate lastfri3 = lastddate - cond(dow(lastddate) >= 5, dow(lastddate) - 5,
> dow(lastddate) + 2)
```

Even cleaner—at least if you are comfortable with remainders—is a formulation using `mod()` (Cox 2007):

```
. generate lastfri4 = lastddate - mod(dow(lastddate) - 5, 7)
```

We can also test that using the equivalent function in Mata:

```
. mata: dow = (6 \ 5 \ 4 \ 3 \ 2 \ 1 \ 0)
. mata: dow, mod(dow :- 5, 7)
       1   2
    +---------+
  1 |  6   1  |
  2 |  5   0  |
  3 |  4   6  |
  4 |  3   5  |
  5 |  2   4  |
  6 |  1   3  |
  7 |  0   2  |
    +---------+
```

If you are new to Mata, what you see is creation of a column vector with values descending from 6 to 0, followed by use of `mod()` to create the term to be subtracted, and further followed by their display side by side. The operator `:-` gives elementwise subtraction. (There are shorter ways to yield the vector `dow`. Those so minded can treat that as an exercise.)

That last solution leads to an easy guess about the same solution for any day of the week to be the last reported date: just change 5 to the result from `dow()` for the chosen day. You might like to test this for yourself.

Let us check that our different solutions match.

```
. format lastfri* %td

. list lastddate lastfri*

     +-----------------------------------------------------------+
     | lastddate     lastfri    lastfri2    lastfri3    lastfri4 |
     |-----------------------------------------------------------|
  1. | 31jan1960   29jan1960   29jan1960   29jan1960   29jan1960 |
  2. | 29feb1960   26feb1960   26feb1960   26feb1960   26feb1960 |
  3. | 31mar1960   25mar1960   25mar1960   25mar1960   25mar1960 |
  4. | 30apr1960   29apr1960   29apr1960   29apr1960   29apr1960 |
  5. | 31may1960   27may1960   27may1960   27may1960   27may1960 |
     |-----------------------------------------------------------|
  6. | 31jan1961   27jan1961   27jan1961   27jan1961   27jan1961 |
  7. | 28feb1961   24feb1961   24feb1961   24feb1961   24feb1961 |
  8. | 31mar1961   31mar1961   31mar1961   31mar1961   31mar1961 |
  9. | 30apr1961   28apr1961   28apr1961   28apr1961   28apr1961 |
 10. | 31may1961   26may1961   26may1961   26may1961   26may1961 |
     +-----------------------------------------------------------+
```

In this case, it is easy enough to scan the data to see that all is well. Programmatically, it is better to use [D] **assert** in a test that all variables are equal. See also Gould (2003).

Other problems with weekly dates, including day of the week, were discussed in previous tips (Cox 2010, 2012a,b).

5 Merits of modular arithmetic

In the movie *Peggy Sue Got Married* (1986), the 43-year-old Peggy Sue (played by Kathleen Turner) wakes up to find herself back in her past, just before she left high school. Faced with an algebra test, she tells her teacher: "I happen to know that in the future I will not have the slightest use for algebra, and I speak from experience." (You can find a video clip at https://www.youtube.com/watch?v=-3eKzmozvrI.)

Usually, I dislike jokes against mathematics, but this one always makes me chuckle when I remember it. The serious point for us: what is the algebra that we will find use for in our work? It certainly includes modular arithmetic. Cox (2007) gave the following as general references: Graham, Knuth, and Patashnik (1994); Knuth (1997); and Biggs (2002). To those, I will add Stewart (1975), Conway and Guy (1996), and Gardner (1997) at the light, entertaining, or introductory end, and Boute (1992), Leijen (2001), and Dershowitz and Reingold (2012) if this is all standard stuff and you want to go deeper or further.

In the tip on uses of the modulus function (really, remainder or residue function), keywords were selections, sequences, and extractions. A keyword that deserves as much if not more prominence is rotations. In the third problem, the day of the week is returned as integers 0 to 6, and the last day of the month can be 0 to 6 days later than the last one acceptable, so the correction is a rotation of integers 0 to 6. More generally, whenever there is a rotation, it is likely that `mod()` could be part of the solution, and this is the point to carry forward. That may seem intuitive, and if it does, that is because it is familiar.

Another example of rotation involving dates is wanting to plot seasonal data that are centered on Northern Hemisphere winters or Southern Hemisphere summers rather than on months of the conventional calendar year running from January to December (Cox 2006, 2015). If the response of interest is snow in Switzerland or sunshine in Sydney, the peak of interest will be around the turn of the calendar year, which would be better in the middle of your graph, not split between two ends. Given, say, a variable `month` that starts at 1 for January, we might want to start the time axis at, say, July. That is a rotation such as given by the Stata code

```
1 + mod(month - 7,12)
```

Let us use Mata to think that through. Opening up with a `mata` command, we follow with

```
: month = (1..12)

: month
          1    2    3    4    5    6    7    8    9   10   11   12
      +-------------------------------------------------------------+
    1 |   1    2    3    4    5    6    7    8    9   10   11   12  |
      +-------------------------------------------------------------+

: month :- 7
          1    2    3    4    5    6    7    8    9   10   11   12
      +-------------------------------------------------------------+
    1 |  -6   -5   -4   -3   -2   -1    0    1    2    3    4    5  |
      +-------------------------------------------------------------+

: mod(month :- 7, 12)
          1    2    3    4    5    6    7    8    9   10   11   12
      +-------------------------------------------------------------+
    1 |   6    7    8    9   10   11    0    1    2    3    4    5  |
      +-------------------------------------------------------------+

: 1 :+ mod(month :- 7, 12)
          1    2    3    4    5    6    7    8    9   10   11   12
      +-------------------------------------------------------------+
    1 |   7    8    9   10   11   12    1    2    3    4    5    6  |
      +-------------------------------------------------------------+
```

As before, `:+` and `:-` are elementwise operators for addition and subtraction.

The trick here, as in many other problems, is that the remainder 0 upward is literally one step away from what you want. Adding 1 finally gets you a rotation from 1 to 12 to a new 1 to 12.

As in the previous section, there are solutions using other functions, such as `cond()`, which is all fine. Knowing several ways to solve a problem always beats knowing none.

6 Counsel for coders

We can find simple morals in this tale that extend to many more problems.

Use sandbox datasets to find solutions.

People wanting this kind of calculation with dates often have large datasets, perhaps with many panels, irregularly spaced dates, and even yet other complications. Set your real dataset aside and make up a simple dataset for which you can check solutions. I started with one based on the observation number. With dates, it may be as or more convenient to use very recent dates so that you know the correct answer or can glance at an accessible calendar (say, on your phone or laptop) to check that the code is correct. If I had chosen for an example some dates that are very recent as I write, which is in April 2019, that would become less convenient during the time that this column may remain useful. The examples show that Mata too can be useful for play within sandboxes.

Know about functions and use the help to look for others. Be prepared to combine functions, typically by feeding the results of one function to another.

Stata has many functions. I would be surprised at any user outside StataCorp who had much need for more than a few of them. But it is worth occasionally scanning the help to find out about functions in your territory that might be useful. Alternatively, see Cox (2011) for a rapid survey of some personal favorites. Naturally, you might have seen quickly which functions to use in these problems.

Knowing several functions often means that you will know more than one way to solve a problem, which is always good news.

Experiment. Sometimes you may need to write out longer code before you can see how it can be shortened.

Solutions to the third problem using `mod()` are clean and generalize further, exactly as would be wished. That said, never be embarrassed by writing simple code that is clear and easy to understand. That is a virtue too. Code is not just written for Stata to execute. It is written to be read—to be debugged, extended, or borrowed as necessary. The reader may be you at a later date, someone in your team, or someone else benefiting from your work if the code is made public. You do not have to be selfless, still less saintly, to be motivated to write clear code; the most natural beneficiary is you.

7 Programs and supplemental materials

To install a snapshot of the corresponding software files as they existed at the time of publication of this article, type

```
. net sj 19-3
. net install dm0100      (to install program files, if available)
. net get dm0100          (to install ancillary files, if available)
```

8 References

Biggs, N. L. 2002. *Discrete Mathematics*. Oxford: Oxford University Press.

Blackburn, B., and L. Holford-Strevens. 1999. *The Oxford Companion to the Year*. Oxford: Oxford University Press.

Boute, R. T. 1992. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems* 14: 127–144.

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

Cox, N. J. 2006. Speaking Stata: Graphs for all seasons. *Stata Journal* 6: 397–419.

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

———. 2010. Stata tip 68: Week assumptions. *Stata Journal* 10: 682–685.

———. 2011. Speaking Stata: Fun and fluency with functions. *Stata Journal* 11: 460–471.

———. 2012a. Stata tip 111: More on working with weeks. *Stata Journal* 12: 565–569.

———. 2012b. Stata tip 111: More on working with weeks, erratum. *Stata Journal* 12: 765.

———. 2015. Speaking Stata: Species of origin. *Stata Journal* 15: 574–587.

Dershowitz, N., and E. M. Reingold. 2012. Modulo intervals: A proposed notation. *ACM SIGACT News* 43: 60–64.

Gardner, M. 1997. *The Last Recreations: Hydras, Eggs, and Other Mathematical Mystifications*. New York: Springer.

Gould, W. 2003. Stata tip 3: How to be assertive. *Stata Journal* 3: 448.

Graham, R. L., D. E. Knuth, and O. Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science*. 2nd ed. Reading, MA: Addison–Wesley.

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

Knuth, D. E. 1997. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd ed. Reading, MA: Addison–Wesley.

Leijen, D. 2001. Division and modulus for computer scientists. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/divmodnote-letter.pdf.

Reingold, N. M., and N. Dershowitz. 2018. *Calendrical Calculations: The Ultimate Edition*. Cambridge: Cambridge University Press.

Samuels, S. J., and N. J. Cox. 2012. Stata tip 105: Daily dates with missing days. *Stata Journal* 12: 159–161.

Stewart, I. 1975. *Concepts of Modern Mathematics*. Harmondsworth: Penguin.

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 16 commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*. His "Speaking Stata" articles on graphics from 2004 to 2013 have been collected as *Speaking Stata Graphics* (2014, College Station, TX: Stata Press).