Stata Tips

Volume I: Tips 1–119

Fourth Edition

NICHOLAS J. COX, Editor
Durham University
Department of Geography

A Stata Press Publication
StataCorp LLC
College Station, Texas

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in $\LaTeX\,2_\varepsilon$
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Print ISBN-10: 1-59718-405-5 (volumes I and II)
Print ISBN-10: 1-59718-407-1 (volume I)
Print ISBN-10: 1-59718-409-8 (volume II)
Print ISBN-13: 978-1-59718-405-2 (volumes I and II)
Print ISBN-13: 978-1-59718-407-6 (volume I)
Print ISBN-13: 978-1-59718-409-0 (volume II)
ePub ISBN-10: 1-59718-406-3 (volumes I and II)
ePub ISBN-10: 1-59718-408-X (volumes I)
ePub ISBN-10: 1-59718-410-1 (volumes II)
ePub ISBN-13: 978-1-59718-406-9 (volumes I and II)
ePub ISBN-13: 978-1-59718-408-3 (volumes I)
ePub ISBN-13: 978-1-59718-410-6 (volumes II)

Library of Congress Control Number: 2023948738

Stata Tips

Volume II: Tips 120–152

Fourth Edition

NICHOLAS J. COX, Editor
Durham University
Department of Geography

A Stata Press Publication
StataCorp LLC
College Station, Texas

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in $\LaTeX\,2_\varepsilon$
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Print ISBN-10: 1-59718-405-5 (volumes I and II)
Print ISBN-10: 1-59718-407-1 (volume I)
Print ISBN-10: 1-59718-409-8 (volume II)
Print ISBN-13: 978-1-59718-405-2 (volumes I and II)
Print ISBN-13: 978-1-59718-407-6 (volume I)
Print ISBN-13: 978-1-59718-409-0 (volume II)
ePub ISBN-10: 1-59718-406-3 (volumes I and II)
ePub ISBN-10: 1-59718-408-X (volumes I)
ePub ISBN-10: 1-59718-410-1 (volumes II)
ePub ISBN-13: 978-1-59718-406-9 (volumes I and II)
ePub ISBN-13: 978-1-59718-408-3 (volumes I)
ePub ISBN-13: 978-1-59718-410-6 (volumes II)

Library of Congress Control Number: 2023948738

Contents

Contents

Editor's preface

The book you are reading reprints the first 119 Stata Tips from the *Stata Journal*, with thanks to their original authors. It is a reissue of *One Hundred Nineteen Stata Tips* from 2014. The *Journal* began publishing tips in 2003, beginning with volume 3, number 4. The Editors are now pleased to reprint this selection in this book. Among past and present Editors, Nicholas J. Cox has overseen the production of these Tips from the beginning, with continued support and encouragement from H. Joseph Newton and Stephen P. Jenkins.

The *Stata Journal* publishes substantive and peer-reviewed articles ranging from reports of original work to tutorials on statistical methods and models implemented in Stata, and indeed on Stata itself. Other features include regular columns such as "Speaking Stata", book reviews, and announcements.

We are pleased by the external recognition that the *Journal* has achieved. The *Stata Journal* is indexed and abstracted by CompuMath Citation Index, Current Contents/Social and Behavioral Sciences, RePEc: Research Papers in Economics, Science Citation Index Expanded (also known as SciSearch), Scopus, and Social Sciences Citation Index.

But back to the Tips. There was little need for tips in the early days. Stata 1.0 was released in 1985. The original program had 44 commands, and its documentation totaled 175 pages. The current version, on the other hand, has hundreds if not thousands of commands—including an embedded matrix language called Mata—and Stata's official documentation now totals more than 18,000 pages. Beyond that, the user community has added several hundred more commands and many more pages explaining them or the official commands.

The pluses and the minuses of this growth are evident. As Stata expands, it is increasingly likely that users' needs can be met by available code. But at the same time, learning how to use Stata and even learning what is available become larger and larger tasks.

The Tips are intended to help. The ground rules for Stata Tips, as found in the original 2003 statement, are laid out as the next item in this book. We have violated one original rule in the letter, if not the spirit: some Stata Tips have been much longer than three pages. However, the intention of producing concise tips that are easy to pick up remains as it was.

The Tips grew from many discussions and postings on Statalist, at Stata conferences, meetings, and workshops, and elsewhere, which underscores a simple fact: Stata is now so big that it is easy to miss even simple features that can streamline and enhance your sessions with Stata. This applies not just to new users, who understandably may quake nervously before the manual mountain, but also to longtime users, who too are faced with a mass of new features in every release.

Tips have come from Stata users as well as from StataCorp employees. Many discuss new features of Stata, or features not documented fully or even at all. We hope that you enjoy the Stata Tips reprinted here and can share them with your fellow Stata users. If you have tips that you would like to write, or comments on the kinds of tips that are helpful, do get in touch with us, as we are eager to continue the series.

Among many complementary resources, and beyond the all-important help files and manual volumes, I want to flag two features of the StataCorp website, https://www.stata.com, namely, the FAQs ("Frequently asked questions on using Stata") and the Stata Blog, *Not Elsewhere Classified*. Both share the primary aims of alerting you to features of Stata and how to use them easily and effectively. They also include many contributions from the user community.

Nicholas J. Cox, Editor
October 2023

Editor's preface

The book you are reading reprints 33 Stata Tips from the *Stata Journal* from 2014 to 2023, with thanks to their original authors. It is a sequel to *One Hundred Nineteen Stata Tips* from 2014, reissued together with this volume. The *Journal* began publishing tips in 2003, beginning with volume 3, number 4. The Editors are now pleased to reprint this selection in this book. Among past and present Editors, Nicholas J. Cox has overseen the production of these Tips from the beginning, with continued support and encouragement from H. Joseph Newton and Stephen P. Jenkins.

The *Stata Journal* publishes substantive and peer-reviewed articles ranging from reports of original work to tutorials on statistical methods and models implemented in Stata, and indeed on Stata itself. Other features include regular columns such as "Speaking Stata", book reviews, and announcements.

We are pleased by the external recognition that the *Journal* has achieved. The *Stata Journal* is indexed and abstracted by CompuMath Citation Index, Current Contents/Social and Behavioral Sciences, RePEc: Research Papers in Economics, Science Citation Index Expanded (also known as SciSearch), Scopus, and Social Sciences Citation Index.

But back to the Tips. There was little need for tips in the early days. Stata 1.0 was released in 1985. The original program had 44 commands, and its documentation totaled 175 pages. The current version, on the other hand, has hundreds if not thousands of commands—including an embedded matrix language called Mata—and Stata's official documentation now totals more than 18,000 pages. Beyond that, the user community has added several hundred more commands and many more pages explaining them or the official commands.

The pluses and the minuses of this growth are evident. As Stata expands, it is increasingly likely that users' needs can be met by available code. But at the same time, learning how to use Stata and even learning what is available become larger and larger tasks.

The Tips are intended to help. The ground rules for Stata Tips, as found in the original 2003 statement, are laid out as the next item in this book. We have violated one original rule in the letter, if not the spirit: some Stata Tips have been much longer than three pages. However, the intention of producing concise tips that are easy to pick up remains as it was.

The Tips grew from many discussions and postings on Statalist, at Stata conferences, meetings, and workshops, and elsewhere, which underscores a simple fact: Stata is now so big that it is easy to miss even simple features that can streamline and enhance your sessions with Stata. This applies not just to new users, who understandably may quake nervously before the manual mountain, but also to longtime users, who too are faced with a mass of new features in every release.

Tips have come from Stata users as well as from StataCorp employees. Many discuss new features of Stata, or features not documented fully or even at all. We hope that you enjoy the Stata Tips reprinted here and can share them with your fellow Stata users. If you have tips that you would like to write, or comments on the kinds of tips that are helpful, do get in touch with us, as we are eager to continue the series.

Among many complementary resources, and beyond the all-important help files and manual volumes, I want to flag two features of the StataCorp website, https://www.stata.com, namely, the FAQs ("Frequently asked questions on using Stata") and the Stata Blog, *Not Elsewhere Classified*. Both share the primary aims of alerting you to features of Stata and how to use them easily and effectively. They also include many contributions from the user community.

Nicholas J. Cox, Editor
October 2023

(Pages omitted)

DOI: 10.1177/1536867X0400300412

The Stata Journal (2003)
3, Number 4, p. 445

Stata tip 1: The eform() option of regress

Roger Newson, King's College London, UK
roger.newson@kcl.ac.uk

Did you know about the `eform()` option of `regress`? It is very useful for calculating confidence intervals for geometric means and their ratios. These are frequently used with skewed Y-variables, such as house prices and serum viral loads in HIV patients, as approximations for medians and their ratios. In Stata, I usually do this by using the `regress` command on the logs of the Y-values, with the `eform()` and `noconstant` options. For instance, in the `auto` dataset, we might compare prices between non-US and US cars as follows:

```
. sysuse auto, clear
(1978 Automobile Data)
. generate logprice = log(price)
. generate byte baseline = 1
. regress logprice foreign baseline, noconstant eform(GM/Ratio) robust
Regression with robust standard errors                 Number of obs =      74
                                                       F(  2,    72) =18043.56
                                                       Prob > F      =  0.0000
                                                       R-squared     =  0.9980
                                                       Root MSE      =  .39332

------------------------------------------------------------------------------
             |               Robust
    logprice |   GM/Ratio   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     foreign |    1.07697    .103165     0.77   0.441     .8897576    1.303573
    baseline |   5533.565   310.8747   153.41   0.000     4947.289    6189.316
------------------------------------------------------------------------------
```

We see from the `baseline` parameter that US-made cars had a geometric mean price of 5534 dollars (95% CI from 4947 to 6189 dollars), and we see from the `foreign` parameter that non-US cars were 108% as expensive (95% CI, 89% to 130% as expensive). An important point is that, if you want to see the baseline geometric mean, then you must define the constant variable, here `baseline`, and enter it into the model with the `noconstant` option. Stata usually suppresses the display of the intercept when we specify the `eform()` option, and this trick will fool Stata into thinking that there is no intercept for it to hide. The same trick can be used with `logit` using the `or` option, if you want to see the baseline odds as well as the odds ratios.

My nonstatistical colleagues understand regression models for log-transformed data a lot better this way than any other way. Continuous X-variables can also be included, in which case the parameter for each X-variable is a ratio of Y-values per unit change in X, assuming an exponential relationship—or assuming a power relationship, if X is itself log-transformed.

86 DOI: 10.1177/1536867X0700700211

The Stata Journal (2007)
7, Number 2, pp. 268–271

Stata tip 45: Getting those data into shape[1]

Christopher F. Baum
Department of Economics
Boston College
Chestnut Hill, MA 02467
baum@bc.edu

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

Are your data in shape? That is, are they in the structure that you need to conduct the analysis you have in mind? Data sources often provide the data in a structure that is suitable for presentation but clumsy for statistical analysis. One of the key data management tools that Stata provides is `reshape`; see [D] **reshape**. If you need to modify the structure of your data, you should be familiar with `reshape` and its two functions: `reshape wide` and `reshape long`. In this tip, we discuss how two applications of `reshape` may be the solution to some knotty data management problems.

As a first example, consider this question posted on Statalist by an individual who has a dataset in the wide form:

`country`	`tradeflow`	`Yr1990`	`Yr1991`
Armenia	imports	105	120
Armenia	exports	90	100
Bolivia	imports	200	230
Bolivia	exports	80	115
Colombia	imports	100	105
Colombia	exports	70	71

He would like to reshape the data into long form:

`country`	`year`	`imports`	`exports`
Armenia	1990	105	90
Armenia	1991	120	100
Bolivia	1990	200	80
Bolivia	1991	230	115
Colombia	1990	100	70
Colombia	1991	105	71

1. This tip was updated to use the new command `import delimited` rather than `insheet`.—Ed.

We must exchange the roles of years and tradeflows in the original data to arrive at the desired structure, suitable for analysis as `xt` data. This exchange can be handled by two successive applications of `reshape`:

```
. reshape long Yr, i(country tradeflow)
(note: j = 1990 1991)
Data                               wide   ->   long
-----------------------------------------------------------------------------

Number of obs.                        6   ->      12
Number of variables                   4   ->       4
j variable (2 values)                     ->   _j
xij variables:
                          Yr1990 Yr1991   ->   Yr
-----------------------------------------------------------------------------
```

This transformation swings the data into long form with each observation identified by `country`, `tradeflow`, and the new variable `_j`, taking on the values of year. We now perform `reshape wide` to make imports and exports into separate variables:

```
. rename _j year
. reshape wide Yr, i(country year) j(tradeflow) string
(note: j = exports imports)
Data                               long   ->   wide
-----------------------------------------------------------------------------

Number of obs.                       12   ->       6
Number of variables                   4   ->       4
j variable (2 values)         tradeflow   ->   (dropped)
xij variables:
                                     Yr   ->   Yrexports Yrimports
-----------------------------------------------------------------------------
```

If we transform the data to wide form once again, the `i()` option contains `country` and `year`, as those are the desired identifiers on each observation of the target dataset. We specify that `tradeflow` is the `j()` variable for `reshape`, indicating that it is a `string` variable. The data now have the desired structure. Although we have illustrated this double-reshape transformation with only a few countries, years, and variables, the technique generalizes to any number of each.

As a second example of successive applications of `reshape`, consider the World Bank's World Development Indicators (WDI) dataset.[2] Their extract program generates a comma-separated value (CSV) database extract, readable by Excel or Stata, but the structure of those data hinders analysis as panel data. For a recent year, the header line of the CSV file is

```
"Series code","Country Code","Country Name","1960","1961","1962","1963",
"1964","1965","1966","1967","1968","1969","1970","1971","1972","1973",
"1974","1975","1976","1977","1978","1979","1980","1981","1982","1983",
"1984","1985","1986","1987","1988","1989","1990","1991","1992","1993",
"1994","1995","1996","1997","1998","1999","2000","2001","2002","2003","2004"
```

2. See http://econ.worldbank.org.

That is, each row of the CSV file contains a *variable* and *country* combination, with the columns representing the elements of the time series.[3]

Our target dataset structure is that appropriate for panel-data modeling, with the variables as columns and rows labeled by country and year. Two applications of `reshape` will again be needed to reach the target format. We first `import delimited` (see [D] **import delimited**) the data and transform the triliteral country code into a numeric code with the country codes as labels:

```
. import delimited using wdiex
. encode countrycode, generate(cc)
. drop countrycode
```

We then must address that the time-series variables are named `var4-var48`, as the header line provided invalid Stata variable names (numeric values) for those columns. We use `rename` (see [D] **rename**) to change `v4` to `d1960`, `v5` to `d1961`, and so on:

```
forvalues i=4/48 {
        rename v`i´ d`=1956+`i´´
}
```

We now are ready to carry out the first `reshape`. We want to identify the rows of the reshaped dataset by both country code (`cc`) and `seriescode`, the variable name. The `reshape long` will transform a fragment of the WDI dataset containing two series and four countries:

```
. reshape long d, i(cc seriescode) j(year)
(note: j = 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972
> 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987
> 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002
> 2003 2004)
Data                               wide   ->   long
-----------------------------------------------------------------------------
Number of obs.                        7   ->     315
Number of variables                  48   ->       5
j variable (45 values)                    ->   year
xij variables:
                  d1960 d1961 ... d2004   ->   d
-----------------------------------------------------------------------------
```

3. A variation occasionally encountered will resemble this structure, but with periods in reverse chronological order. The solution here can be used to deal with that problem as well.

```
. list in 1/15

     +--------------------------------------------------------+
     |  cc   seriesc~e   year   countryname                d  |
     |--------------------------------------------------------|
  1. | AFG   adjnetsav   1960   Afghanistan                .  |
  2. | AFG   adjnetsav   1961   Afghanistan                .  |
  3. | AFG   adjnetsav   1962   Afghanistan                .  |
  4. | AFG   adjnetsav   1963   Afghanistan                .  |
  5. | AFG   adjnetsav   1964   Afghanistan                .  |
     |--------------------------------------------------------|
  6. | AFG   adjnetsav   1965   Afghanistan                .  |
  7. | AFG   adjnetsav   1966   Afghanistan                .  |
  8. | AFG   adjnetsav   1967   Afghanistan                .  |
  9. | AFG   adjnetsav   1968   Afghanistan                .  |
 10. | AFG   adjnetsav   1969   Afghanistan                .  |
     |--------------------------------------------------------|
 11. | AFG   adjnetsav   1970   Afghanistan   -2.97129  |
 12. | AFG   adjnetsav   1971   Afghanistan   -5.54518  |
 13. | AFG   adjnetsav   1972   Afghanistan   -2.40726  |
 14. | AFG   adjnetsav   1973   Afghanistan   -.188281  |
 15. | AFG   adjnetsav   1974   Afghanistan    1.39753  |
     +--------------------------------------------------------+
```

The rows of the data are now labeled by year, but one problem remains: all variables for a given country are stacked vertically. To unstack the variables and put them in shape for `xtreg` (see [XT] **`xtreg`**), we must carry out a second `reshape` that spreads the variables across the columns, specifying `cc` and `year` as the *i* variables and `seriescode` as the *j* variable. Since that variable has string content, we use the `string` option.

```
. reshape wide d, i(cc year) j(seriescode) string
(note: j = adjnetsav adjsavC02)
Data                               long   ->   wide
-----------------------------------------------------------------------------
Number of obs.                      315   ->    180
Number of variables                   5   ->      5
j variable (2 values)        seriescode   ->   (dropped)
xij variables:
                                      d   ->   dadjnetsav dadjsavC02
-----------------------------------------------------------------------------

. order cc countryname

. tsset cc year
       panel variable:  cc (strongly balanced)
        time variable:  year, 1960 to 2004
```

After this transformation, the data are now in shape for `xt` modeling, tabulation, or graphics.

As illustrated here, the `reshape` command can transform even the most inconvenient data structure into the structure needed for your research. It may take more than one application of `reshape` to get there from here, but it can do the job.

The Stata Journal (2014)
14, Number 1, pp. 221–225 DOI: 10.1177/1536867X1401400115

Stata tip 117: graph combine—Combining graphs

Lars Ängquist
Institute of Preventive Medicine
Bispebjerg and Frederiksberg Hospitals—The Capital Region
Copenhagen, Denmark
lars.henrik.angquist@regionh.dk

1 Introduction

There are many different reasons for wanting to create multipanel graphs, presented in $r \geq 1$ rows and $c \geq 1$ columns: these reasons include making efficient use of restricted display space and enhancing the presentation of results. In basic Stata, the flexible approach to confidently handle these tasks is given by using the `graph combine` functionality (see `help graph combine`). For related discussions and examples, see the stimulating books *An Introduction to Stata for Health Researchers* (Juul and Frydenberg 2010) and *A Visual Guide to Stata Graphics* (Mitchell 2012).

2 Basic usage

First, we start with setting up seven simple, but quite artificial, linear relations disturbed by normally distributed noise based on simulated x and y variables (interpreted in the standard sense).

```
set obs 100
generate xvar=10*runiform()
forvalues i=1/7 {
   generate y`i´=xvar*`i´+runiform()*(`i´*3)
   label variable y`i´ "Outcome variable `i´"
}
```

Second, we simply fit linear regressions that correspond to these relations and then save the seven corresponding graphs in memory.

```
foreach yvar of varlist y* {
   local lbl: variable label `yvar´
   sort xvar
   reg `yvar´ xvar
   local b : display %3.2f _b[xvar]
   predict p, xb
   twoway (scatter `yvar´ xvar) (line p xvar),              ///
      ytitle("`lbl´") xtitle("Explanatory covariate")       ///
      yscale(range(0 80))                                   ///
      legend(off) note("{&beta}=`b´", position(4) ring(0))  ///
      name("graph_`yvar´", replace)
   drop p
}
```

(Here we use the `name(`*string*`)` option—unless we want to actually save the separate graphs to disk. In that case, we would replace this option with `saving(`*string*`)`.)

Finally, we intend to combine the graphs into a multipanel setup. Assuming that the graphs belong to two distinct groups—graphs 1–3 and 4–7, respectively—they are mirrored in the construction. This is achieved by the following:

1. Combine graphs 1–3 into panel 1.
2. Combine graphs 4–7 into panel 2.
3. Combine the resulting 1-row panels, panel 1 ($r \times c = 1 \times 3$) and panel 2 ($r \times c = 1 \times 4$), into a final 2-row panel ($r = 2$; see figure 1).

```
graph combine graph_y1 graph_y2 graph_y3,                                  ///
   name("firstset", replace) ycommon cols(3) title("First set of graphs")
graph combine graph_y4 graph_y5 graph_y6 graph_y7,                         ///
   name("secondset", replace) ycommon cols(4) title("Second set of graphs")
graph combine firstset secondset,                                          ///
   saving("sevenpanelgraph.gph", replace) ycommon cols(1)
graph export sevenpanelgraph.eps, replace
```

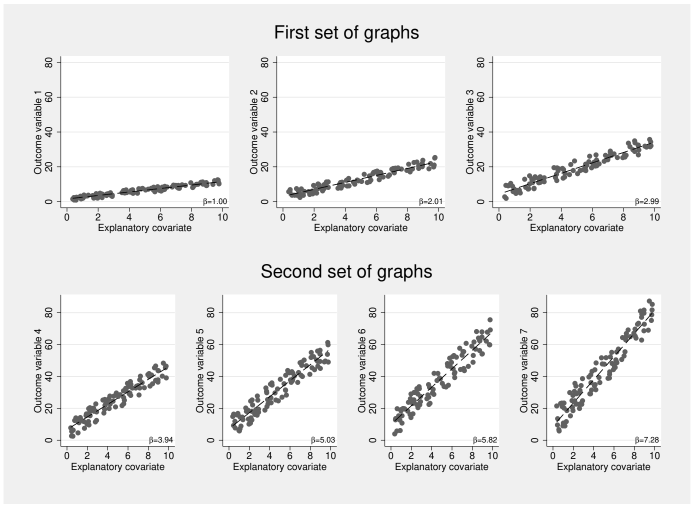

Figure 1. Multipanel graph—a combination of combined graphs

3 Some notes on options

The basic functionality facilitates an easy-to-use combination of graphs. A well-suited set of selected options might improve the display.

3.1 Axes

In many cases, keeping scales constant over panels might enhance the interpretability of the jointly graphed relations. Generally, this might prove to be a valid argument; however, it is imperative for the x axis and y axis when comparing vertically (the `xcommon` option) and horizontally (the `ycommon` option), respectively.

3.2 Margins

To keep the panels as tightly linked as possible—to increase overall comparability—it might be suitable to reduce margins through `imargin(zero)`; for other margin choices, see `help marginstyle`.

3.3 Panel pattern

The final number of panels to use is implicitly given by the stated list of panels in the actual program call. (Remember that each panel might in itself be a previously constructed multipanel. In the above example, a single column, $c = 1$, was used at the combination stage.) To define which $r \times c$ panel-matrix shape will be used, one may choose any of the following options (one is enough): `rows(`*integer*`)` or `cols(`*integer*`)`. To make the graph (distribution of panels) unique, select the `colfirst` option (or not). If the required number of panels is less than the available number $r{\cdot}c$, it may be useful to explicitly—given the unique order—tell Stata which panels should be left empty (instead of the default) by using `holes(`*numlist*`)`.

3.4 Scaling

Each panel is downscaled when using multipanels, text and markers, etc. It is possible to rescale the downscaling through the `iscale(`*scale*`)` option, where *scale* is either an absolute (positive) or a relative value. For example, the absolute value `1` means the original size, and the relative value `*1` implies the same size as the default selection; `0.75` and `*0.75` will adjust the size to the three-quarter size counterparts.

4 A second example

For our second example, we will play around with the individual panel sizes. For this, we will use one of the seven graphs (the sixth) from figure 1, which is inspired by the informative help file (see the end of the `help graph combine` post), to complement it with the two corresponding underlying histograms (see result in figure 2).

```
histogram xvar,                                     ///
    percent start(0) width(1)                       ///
    xscale(range(0 10) off)                         ///
    fxsize(100) fysize(25)                          ///
    yscale(range(0 15)) ytitle("")                  ///
    ylabel(0(5)15, angle(horizontal))               ///
    kdensity kdenopts(lpattern(dash))               ///
    plotregion(margin(zero))                        ///
    note("N (%)", ring(0) position(10))             ///
    name("hist_xvar", replace)
histogram y6,                                       ///
    percent start(0) width(10) horizontal           ///
    xtitle("") xlabel(0(10)20) xscale(rev)          ///
    fxsize(25) fysize(100)                          ///
    yscale(range(0 80) off)                         ///
    ylabel(10(20)70, angle(horizontal))             ///
    kdensity kdenopts(lpattern(dash))               ///
    plotregion(margin(zero))                        ///
    note("N (%)", ring(0) position(4))              ///
    name("hist_y6", replace)
```

In the next step, these three panels are combined (note that we use some of the options just discussed). The main point here is that the options `fxsize(`*number*`)` and `fysize(`*number*`)` govern the widths and heights of the panels; that is, in the example above, the thin sides are left at 25% of the original sizes.

```
graph combine hist_y6 graph_y6 hist_xvar,               ///
    holes(3) rows(2)                                    ///
    imargin(0 2 0 0)                                    ///
    title("  Twoway graph with histograms", ring(0))    ///
    saving(graphwithhistograms.gph, replace)
graph export graphwithhistograms.eps, replace
```

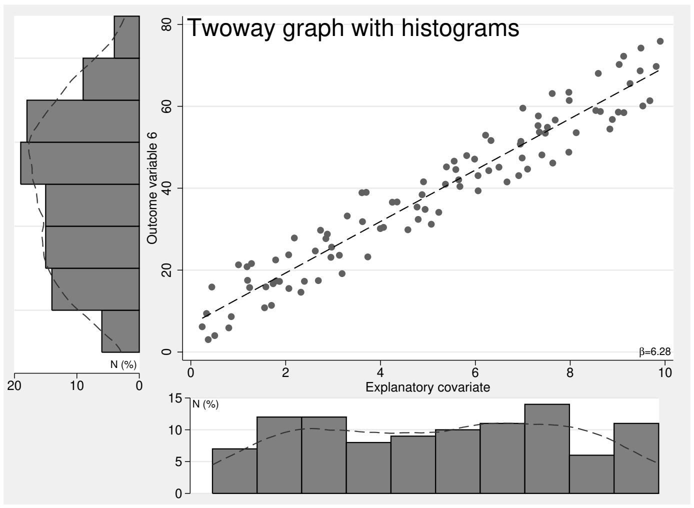

Figure 2. Multipanel graph—a scatterplot with a prediction line and two complementary histograms

5 Discussion and alternatives

In many situations where the subgraphs combine corresponding true data subsets of the present loaded data, a similarly performing alternative would be to use the `by()` option (see `help by_option`). Here the syntax `by(`*varlist*[`,` *options*]`)` allows combined graphing of the corresponding defined graph with respect to all present categories specified by the categorical variables given in *varlist*. In this setting, the options `total` and `missing` add panels based on the total dataset (over nonmissing groups) and missing data for individuals, respectively.

5.1 by() options

As noted above, the option `by()` allows for suboptions. Some suboptions are similar to the ones available for `graph combine`—for example, `colfirst`, `cols()`, `rows()`, `holes()`, `iscale()`, and `imargin()`. Similar functionality, but with different names and adapted settings, is given by `compact` (reduces margins between panels), `norescale` (uses the same scales over panels), and `noedgelabel` (restricts the number of displayed labels). Note that an option with `no`, such as `norescale`, generally has a counterpart, such as `rescale`, with a quite obvious implication.

Usually, this type of solution might be convenient in different cases; however, in most situations, this solution is less flexible and more restrictive by nature. Furthermore, graphing several subgroups within a single panel (together but separately marked) is an alternative solution that allows the smaller number of subgroups to be totally displayed while applying distinct colors and markers. For other cases, the multipanel design may be the best choice because one (or several) background groups can be added to each panel to enhance overall comparability. For example, see the discussion of overlaid graphs in Cox (2010), where subgroups are plotted against completely complementary data while using discrete gray-scaled backdrop markers for the background group. This is referred to as adopting a substrate, or subset, graphing design.

References

Cox, N. J. 2010. Speaking Stata: Graphing subsets. *Stata Journal* 10: 670–681. https://doi.org/10.1177/1536867X1101000408.

Juul, S., and M. Frydenberg. 2010. *An Introduction to Stata for Health Researchers*. 3rd ed. College Station, TX: Stata Press.

Mitchell, M. N. 2012. *A Visual Guide to Stata Graphics*. 3rd ed. College Station, TX: Stata Press.

150 DOI: 10.1177/1536867X221141068

The Stata Journal (2022)
22, Number 4, pp. 998–1003

Stata tip 148: Searching for words within strings

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

1 The problem: Looking for words

Searching for particular text within strings is a common data management problem. One frequent context is whenever various possible answers to a question are bundled together in values of a string variable. Suppose people are asked which sports they enjoy or something more interesting, like which statistical software they use routinely. To keep the matter simple, we will first imagine just lists of one or more numbers that are concise codes for distinct answers, say, `"42"` for `"cricket"` or `"1"` for `"Stata"`. Nonnumeric codes will also be considered in due course. For more on handling such questions, sometimes called multiple response, see Cox and Kohler (2003) or Jann (2005).

The precise problem discussed in this tip is finding text in strings whenever such text is a word in Stata's sense, or something close to that. This needs a little explanation.

Here is a tiny sandbox dataset that will be enough to show the problem and some devices that can yield solutions. By way of example, we will focus mainly on a goal of generating indicator variables, sometimes known as dummy variables. For one overview of generating such variables, see Cox and Schechter (2019). We will also touch on the problem of counting instances of a word.

```
. input str8 mytext
        mytext
  1. "1"
  2. "1 2"
  3. "1 2 11"
  4. "11 12 13"
  5. "11 12 13 111"
  6. end
```

Searching for `"1"` or `"2"`, say, starts with looking for either character with a string function. The function `strpos()` is useful for that. For a rapid personal survey of especially useful functions, see Cox (2011a).

Finding such single characters is easy and unproblematic if the possible answers are one character long at most. More generally, searches are easy if there is no ambiguity. Consider

```
. generate byte is1 = strpos(mytext, "1") > 0
```

The function `strpos()` looks for particular text within other text. It returns 0 if that particular text is not found and a positive number, the position of that particular text, if that text is found. Thus, the position of `"1"` in `"1 2"` is 1, the position of `"2"` in

"1 2" is 3, and so on. Hence, an indicator variable like `is1` will be returned as 1 if there are observations in which `strpos()` returns a positive result. Otherwise, the indicator variable will be returned as 0. If you are new to the idea that an expression like

```
strpos(mytext, "1") > 0
```

returns 1 if true and 0 if false, see Cox and Schechter (2019) or, more directly, Cox (2005, 2016).

If you look again at the sandbox, you should see what is coming next. Looking for "1" with

```
strpos("1 2 11", "1")
```

will still work, fortunately, but looking for "1" with

```
strpos("11 12 13", "1")
```

will yield a false positive. The problem is that we want to find "1" only if it occurs by itself, namely, as a separate word. Stata's primary sense of a word within a string is that words are separated by spaces.

In some Stata contexts, double quotation marks bind together more strongly than spaces separate, so "Stata is subtle" would be treated as a single word if the quotation marks were explicit. For present purposes, we will leave that complication aside.

2 A solution: Looking for spaces too

Let's carry forward the idea that we need to look for spaces too. At first sight, this is a beautiful idea that just does not work very well because there are too many possibilities to catch. Thus, looking for "1 " catches "1"—as part of "1 "—and not "11" within "1 2 11", which is as intended. But it catches the first "1 "—as part of "11 "—within "11 12 13", which is not what we want. Other way round, looking for " 1" catches correctly sometimes and incorrectly other times. Looking for " 1 "—with spaces before and after—will not work if "1" is the first word or the last word, so without a previous space or a following space, respectively.

But that last idea can be made to work with a simple twist. Congratulations if you thought of this directly!

```
. generate byte is1 = strpos(" " + mytext + " ", " 1 ")
. list

     ┌────────────────────┐
     │   mytext     is1   │
     ├────────────────────┤
  1. │        1       1   │
  2. │      1 2       1   │
  3. │   1 2 11       1   │
  4. │ 11 12 13       0   │
  5. │ 11 12 13       0   │
     └────────────────────┘
```

So we solve the problem of initial and following spaces by supplying them on the fly. Note that we do not need to `generate` a new variable or `replace` an existing variable; we just get Stata to work with a version of the variable with extra spaces. Extra spaces that go beyond our need are harmless, because `"  1  "`, in which "1" has two spaces before it and two after it, is treated the same way as `" 1 "`, in which "1" has one space before it and one after it.

3 What about other separators?

Suppose our string variable used another separator, say, commas, which could just be a different convention or a good idea anyway if spaces occur naturally. Someone's favorite sport might be `"water polo"` or `"debugging code"`. Then whatever the commas separate are not words in Stata's technical sense, but they are still words for us or atoms we wish to seek as such.

We could still use a similar idea of looking for `",1,"` within `"," + mytext + ","`. We just need to watch for gratuitous extra spaces so that `"1 ,"` is not missed. If strings could be moderately complicated, we might need a different method. More positively, if spaces have no meaning and we have values like `"1,2 ,3"`, then changing all commas to spaces allows the method of the previous section to be used.

4 A solution: What would change if we deleted words?

Here is another solution. This time around, an example comes before the explanation.

```
. generate byte IS1 = strlen(mytext) > strlen(subinword(mytext, "1", "", 1))
. list

     ┌──────────────────────────┐
     │   mytext   is1   IS1     │
     ├──────────────────────────┤
  1. │        1     1     1     │
  2. │      1 2     1     1     │
  3. │   1 2 11     1     1     │
  4. │ 11 12 13     0     0     │
  5. │ 11 12 13     0     0     │
     └──────────────────────────┘
```

We get the same answer, so how did that work?

The function `strlen()` measures the length of strings by counting characters. Although no longer documented, the older name `length()` still works if you remember or prefer that.

The function `subinword()` replaces text with other text if and only if that text occurs as a word in Stata's primary sense. The function knows how to handle words at the beginning and end of strings. However, `subinword()` does not follow Stata's extended sense that a word can be defined (meaning, delimited) by explicit double quotation marks.

But how does replacing text help? We do not want to change text; we are just searching for it. Yet, if the result of replacing text by an empty string (deleting it, to put it plainly) would be to reduce the length of the string, then evidently we did find that text.

Notice "would be". As before, we do not have to `generate` a new variable or `replace` an existing variable. We just get Stata to tell us what the result would be if the text existed and so would be deleted.

Whether the length of the string is greater than the length of the string with the word removed is a true or false question. Either the first length is greater because there is at least one instance of the word or the two lengths are the same because there is no such instance. If the expression is true, 1 is returned; and if it is false, 0 is returned, giving us an indicator variable.

This method is of interest for another reason: you may want to count instances of a word. We could have written

```
. generate byte IS1 = strlen(mytext) > strlen(subinword(mytext, "1", "", .))
```

The difference is in the last argument fed to `subinword()`, namely, system missing `.` rather than 1. That different syntax instructs Stata to delete all instances of the word `"1"` rather than the first only. For detecting whether the word exists, you need know only that it exists at least once.

If the problem is counting instances instead of checking for existence, then the difference in lengths

```
. generate count1 = strlen(mytext) - strlen(subinword(mytext, "1", "", .))
```

is precisely the number of times `"1"` occurs as a word. If you are looking for instances of `"11"` or `"111"`, remember to divide by 2 or 3—the lengths of the words in question, respectively—or you will get the number of characters notionally deleted, not the number of words.

For more on counting substrings, see Cox (2011b).

5 Nonnumeric words

Datasets may include one or more nonnumeric words bundled in a string variable. Suppose there was a survey question about which programming languages are routine for Stata users, with possible answers such as one or more of `Python`, `Julia`, `C++`, and `C`.

Handling such nonnumeric words can be both easier and more difficult than handling numeric words. The possibility of ambiguity is less but still present, as witness checking for mentions of `C` and finding them within mentions of `C++`. Hence, insisting on searching for a word, and not just a substring, can be necessary using one of the devices just explained.

Greater difficulty can arise because of variations in spelling and punctuation, depending sensitively on how such data were entered and collated. Suppose that `none` was expected as an answer when true but that there are also instances of `None`, `NONE`, and so forth. This particular variability is easily handled by looking for `none` within `strlower()` or—according to taste—looking for `NONE` within `strupper()`. The older function names `lower()` and `upper()` are equivalent and still work. Other variations in spelling may be harder to handle, but the first step is always to find out exactly which names were used.

6 A list of tricks

We have covered two main ideas:

- Words are separated by spaces, so look for a word together with previous and following spaces, remembering how to catch words at the beginning or the end of a string (sections 2 and 3).
- If we ask Stata to tell us whether and how the length of a string would change if we were to delete a word, we have ways to detect the occurrence of that word, either yes or no, or the number of occurrences if that is what we seek (section 4).

That is not a complete treatise, even on this small topic. A longer account might mention other possibilities, complications that may arise, or possible solutions.

First, I will mention other problems:

- I have focused on plain ASCII characters, but searching for Unicode needs more care and different functions.
- I have mentioned but not fully solved the complication of "words" that include spaces. But the more complicated the string we are searching for, the less likely ambiguity is to bite.
- I have focused on simple searching of string variables, but string manipulation is needed in other contexts, such as parsing user input if you are writing Stata programs.

Now, I will signal other solutions:

- Many readers will already know about regular expression syntax.
- Sometimes, we cannot solve a problem with one command line. We may need to use the `gettoken` (see [P] **gettoken**) command or the `split` (see [D] **split**) command. We may need to loop over words with a construct like `foreach` or `forvalues` (see [P] **foreach** or [P] **forvalues**).

All of these matters deserve detailed treatment, which is left to other accounts.

7 Acknowledgment

William Lisowski made helpful comments on a draft.

References

Cox, N. J. 2005. FAQ: What is true or false in Stata? https://www.stata.com/support/faqs/data-management/true-and-false/.

———. 2011a. Speaking Stata: Fun and fluency with functions. *Stata Journal* 11: 460–471. https://doi.org/10.1177/1536867X1101100308.

———. 2011b. Stata tip 98: Counting substrings within strings. *Stata Journal* 11: 318–320. https://doi.org/10.1177/1536867X1101100212.

———. 2016. Speaking Stata: Truth, falsity, indication, and negation. *Stata Journal* 16: 229–236. https://doi.org/10.1177/1536867X1601600117.

Cox, N. J., and U. Kohler. 2003. Speaking Stata: On structure and shape: The case of multiple responses. *Stata Journal* 3: 81–99. https://doi.org/10.1177/1536867X0300300106.

Cox, N. J., and C. B. Schechter. 2019. Speaking Stata: How best to generate indicator or dummy variables. *Stata Journal* 19: 246–259. https://doi.org/10.1177/1536867X19830921.

Jann, B. 2005. Tabulation of multiple responses. *Stata Journal* 5: 92–122. https://doi.org/10.1177/1536867X0500500113.

The Stata Journal (2023)
178 DOI: 10.1177/1536867X231175349 **23**, Number 2, pp. 589–594

Stata tip 152: if and if: When to use the if qualifier and when to use the if command

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

Clyde B. Schechter
Albert Einstein College of Medicine
Bronx, NY
clyde.schechter@einsteinmed.edu

1 Introduction

Stata has an `if` qualifier and an `if` command. Here we discuss generally when you should use either and specifically flag a common pitfall in using the `if` command. In a nutshell, the pitfall arises from confusing the two constructs: the `if` command does not loop over the data but, at most, looks in the first observation of a dataset. There has long been a StataCorp FAQ on this topic (Wernow 2005), but we and others have usually tried to explain matters otherwise. This tip is intended as a more durable version of the story that should be easier to find than occasional Statalist postings that are vivid when read but hard to find later.

2 The if qualifier

The `if` qualifier is met by most users early in their Stata experience. Its purpose is to select observations (cases, records, or rows in the dataset) for some action. Thus, you could run the following commands to read in a dataset and first `summarize` a variable and then `summarize` that variable again for a subset of observations. Here we suppress the results, but if you are new to Stata and unfamiliar with `summarize`, it would be worth your time to run the code yourself to find out about a valuable command.

```
. sysuse auto
. summarize mpg
. summarize mpg if foreign == 1
```

When the `if` qualifier is used (or, in other words, when an `if` condition is specified), Stata tests the expression given—here `foreign == 1`—in each observation to see whether it is satisfied (is true) in that observation. Observations for which the expression is true are selected for the action. In this example, `foreign` is an indicator variable that is 1 if a car is foreign (made outside the United States) and 0 if a car is domestic (made inside the United States). The operator `==` tests for equality, noting that in Stata the `=` operator typically indicates assignment of a value or values, say, to a variable. Out of 74 cars, 22 qualify as being foreign, so their observations will be `summarize`d for the variable `mpg`.

Stata follows a very widely used convention, running across statistics, mathematics, and computing, that in logical tests, a value of 1 means true and a value of 0 means false. In fact, Stata's rule is more general: Any numeric value that is not 0 means true, while only the numeric value 0 means false. Watch out with missing values because any numeric value that represents missing (whether system missing, `.`, or extended missing values from `.a` to `.z`) is certainly not 0 and so yields true in a logical test.

Logical tests in Stata take two forms. First, and more commonly, some logical operator is used in an expression. Tests for equality, using the `==` operator, may be what you need; otherwise, some test for inequality may be needed. See the help for operators to see the complete list. Thus, in `auto.dta` you could select cars with high `mpg` by, say, `mpg > 25`. Logical tests can combine two or more conditions, but even so the keyword `if` appears only once in any comparison.

Second, you can ask Stata to look inside a numeric variable and check whether its values are 0 or not. In `auto.dta`, `foreign` is only ever 1 or 0 and never missing. So the test `if foreign` is precisely the same test in practice as `if foreign == 1`. Presented with `if foreign`, Stata looks inside the variable and selects those observations for which it is not 0, which in practice is the same subset of observations as those for which the condition `if foreign == 1` is true.

There are positive and negative sides to this flexibility. The positive side is that we can write Stata code that may appeal to readers as idiomatic in their own language and in Stata too. "Let's focus on the cars that are foreign" becomes the condition `if foreign`. Such coding works best if you follow a convention, which we strongly recommend, of naming an indicator variable for the condition coded as 1. That is precisely what the developers of Stata did at the very beginning when coding up the auto data.

The negative side is that the inclusiveness here could bite if there are nonzero values that the condition `if foreign` would catch too, even though that is not what you intend. As said, nonzero values include any numeric missing values. So you might well prefer to be safe rather than succinct and always spell out, say, `if foreign == 1`.

For more on truth and falsity in Stata, see Cox (2005, 2016). For more on indicator variables, see Cox and Schechter (2019), especially if you have been thinking "Don't you mean dummy variables?" (Yes, we do.)

3 The if command

The previous section may have strengthened your understanding of the `if` qualifier, say, by spelling out some nuances. At this point in the story, the most important detail about the `if` command is that it is emphatically not a way to do the same thing differently. Oddly, or otherwise, a misunderstanding that the two are equivalent (or at least overlap in what they do) seems to arise most often with people new to Stata who are accustomed to programming in some other language. Such programmers may guess or hope that Stata's `if` command is similar to, or an extension of, what they know already.

Whatever the explanation, constructs using `if` or some equivalent keyword have been present in many programming languages over several decades. Examples can be found in Sammet (1969), Kernighan and Plauger (1978), and Bal and Grune (1994).

We will pursue this negative theme before turning to when and why the `if` command is appropriate or useful. Otherwise, there would be no point to including it within Stata.

Any puzzlement is intensified whenever Stata allows use of the `if` command in a way that seems equivalent to use of the `if` qualifier. It then gives results that occasionally are what you want but more often just seem bizarre. As examples, consider these two statements and their results:

```
. if foreign == 1 summarize mpg
. if foreign == 0 summarize mpg
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         74     21.2973    5.785503         12         41
```

Stata complains about neither statement, so each is perfectly legal. But you might even wonder whether you have unearthed a bug. The first statement yields no results, whereas we already know that there are observations for which `foreign == 1`. Other way round, the second statement yields results, but if you look carefully, you will see that the results are for the entire dataset and so include both foreign and domestic cars.

The explanation is immediate given one extra piece of information. When an `if` command refers to a variable (or variables) in the dataset, Stata looks only in the first observation. It is exactly as if you wrote `if foreign[1] == 1` or `if foreign[1] == 0`. It so happens that the first statement is false and the second statement is true, as can be checked independently by looking at the data with, say, `list in 1` or `edit in 1` or `display foreign[1]`. Because the first statement was false, Stata did not execute the next command, `summarize mpg`. Because the second statement was true, Stata did execute the (same) next command. In both cases, the subset of observations specified was not part of the syntax for the next command.

We could make that plainer by writing the same syntax using curly brackets or braces:

```
if foreign[1] == 0 {
    summarize mpg
}
```

Backing up slightly: Here a so-called subscript such as `[1]` attached to a variable name indicates an observation number, so in another example `foreign[42]` would be the value of `foreign` in observation 42. We say "subscript" as an allusion to mathematical notation such as y_1 or y_{42}, but naturally writing *sub scriptum*, below the line, is not strictly possible in Stata.

A more general point to emphasize is that there is no sense in Stata in which the `if` command iterates or loops over the observations in the dataset. (Here we are assuming that there are data in memory; it is perfectly possible to use Stata with no variables in memory, and you may wish to think through what could be done depending on what else is allowed.) Positively put, the `if` command makes one and only one decision, depending on whether the condition specified is true.

The `if` command is very widely used within do-files and within programs, including within programs that define other commands.

There are many examples within Stata programs. Options are typically implemented in this way. In many commands, there are optional choices, either for extra actions or to vary some action from the default. Inside the command code, there is typically a switch for each option whereby different code is executed. The `summarize` command has options, such as `meanonly` (to do less than the default) or `detail` (to do more). That command is built in, so users may not see the internal code, but very many commands are implemented through ado-code, so much of or all the code is visible. If you are curious, you can look inside ado-code with, say,

```
. viewsource tabstat.ado
```

and you will immediately see a series of switches all using the `if` command to set up calculations according to whatever a user did (or did not) specify when issuing the `tabstat` command.

Another common sequence within ado-code is something like this.

```
. marksample touse
. count if `touse´
  74
. if r(N) == 0 error 2000
```

Here `marksample` has the job of creating a temporary indicator variable `` `touse' `` that is 1 when observations are to be used and 0 otherwise. (If the name `touse` looks odd to you, think "to use".) Exclusions arise for one of two reasons: whenever missing values make the use of observations impossible or whenever an `if` qualifier (there it is again) or an `in` qualifier excludes observations by implication. We then `count` the observations to be used. The result is left in `r(N)`. If that result is 0, then there are no observations to use, which here and usually is regarded as an error. If, as it were, no news is good news, such as when we are checking for something bad but fail to find it, then the syntax would be different. We might well condition on, say, `r(N) > 0`.

There are other vital differences between the `if` qualifier and the `if` command, beyond the cosmetic (but still crucial) difference that the first follows and the second precedes associated code.

The `if` command can be associated with code following `else` to indicate what should be done if the condition specified is false. Indeed, a more or less complicated series of branching decisions may be needed depending on a menu of possible choices. Again, if you are curious, look at the results of

```
. viewsource duplicates.ado
```

which show a series of branches aimed at identifying the subcommand that a user specified after the command itself, such as `duplicates report` or `duplicates list`.

Lest you think that the `if` command is primarily of interest to Stata programmers, let's look at an example of its use in a common situation that arises in data analysis. Suppose you want to analyze some panel data, performing some specific calculations separately in each panel but only in those panels that offer a minimum sample size. Here we assume for simplicity that firms have distinct numeric identifiers. The code in your do-file might look like this:

```
generate abnormal_return = .
levelsof firm, local(firms)
foreach f of local firms {
    count if firm == `f´                            // N.B. if qualifier
    if r(N) >= 30 {                                 // N.B. if command
        regress return market_return if firm == `f´    // if qualifier
        predict resid, resid
        replace abnormal_return = resid if firm == `f´ // if qualifier
        drop resid
    }
}
```

Notice that both the `if` command and the `if` qualifier are used in this code, with very different effects. The `if` qualifier applies only to the single command in which it appears, and it restricts those commands to the observations for which `firm == `f'`. The `if` command appears only once in the code, but it controls execution of the following four commands; they are executed only if the result of the preceding `count` command is at least 30. Note, in particular, that this `if` command does not examine any observations in the data in memory: it refers only to the result returned by the preceding `count` command. Note also the use of curly braces to apply the single `if` command to an entire block of commands. Those four commands are all executed, or none are, depending on the available sample size for the firm.

You may be thinking of refinements, such as counting observations with nonmissing values, because observations with missing values are of no use for any regression. You may also know of community-contributed commands in this area, but discussing those is beyond our scope here. Even if you have access to such commands, understanding the principles in this last example is valuable in many contexts.

References

Bal, H. E., and D. Grune. 1994. *Programming Language Essentials*. Wokingham: Addison–Wesley.

Cox, N. J. 2005. FAQ: What is true or false in Stata? https://www.stata.com/support/faqs/data-management/true-and-false/.

———. 2016. Speaking Stata: Truth, falsity, indication, and negation. *Stata Journal* 16: 229–236. https://doi.org/10.1177/1536867X1601600117.

Cox, N. J., and C. B. Schechter. 2019. Speaking Stata: How best to generate indicator or dummy variables. *Stata Journal* 19: 246–259. https://doi.org/10.1177/1536867X19830921.

Kernighan, B. W., and P. J. Plauger. 1978. *The Elements of Programming Style*. New York: McGraw–Hill.

Sammet, J. E. 1969. *Programming Languages: History and Fundamentals*. Englewood Cliffs, NJ: Prentice-Hall.

Wernow, J. 2005. FAQ: I have an if or while command in my program that only seems to evaluate the first observation. What's going on? http://www.stata.com/support/faqs/programming/if-command-versus-if-qualifier/.