mark — Mark observations for inclusion

Description

`marksample`, `mark`, and `markout` are for use in Stata programs. `marksample` and `mark` are alternatives; `marksample` links to information left behind by `syntax`, and `mark` is seldom used. Both create a 0/1 to-use variable that records which observations are to be used in subsequent code. `markout` sets the to-use variable to 0 if any variables in `varlist` contain missing and is used to further restrict observations.

`markin` is for use after `marksample`, `mark`, and `markout` and, sometimes, provides a more efficient encoding of the observations to be used in subsequent code. `markin` is rarely used.

`svymarkout` sets the to-use variable to 0 wherever any of the survey-characteristic variables contain missing values; it is discussed in [SVY] `svymarkout` and is not further discussed here.

Syntax

Create marker variable after `syntax`

```
marksample hname [, novarlist strok zeroweight noby]
```

Create marker variable

```
mark newmarkvar [if] [in] [weight] [, zeroweight noby]
```

Modify marker variable

```
markout markvar [varlist] [, strok sysmissok]
```

Find range containing selected observations

```
markin [if] [in] [, name(lclname) noby]
```

Modify marker variable based on survey-characteristic variables

```
svymarkout markvar
```

`aweights`, `fweights`, `iweights`, and `pweights` are allowed; see [U] 11.1.6 `weight`. `varlist` may contain time-series operators; see [U] 11.4.4 `Time-series varlists`. 
Options

novarlist is for use with marksample. It specifies that missing values among variables in \textit{varlist} not cause the marker variable to be set to 0. Specify novarlist if you previously specified

\begin{verbatim}
syntax newvarlist ...
\end{verbatim}
or

\begin{verbatim}
syntax newvarname ...
\end{verbatim}

You should also specify novarlist when missing values are not to cause observations to be excluded (perhaps you are analyzing the pattern of missing values).

strok is used with marksample or markout. Specify this option if string variables in \textit{varlist} are to be allowed. strok changes rule 6 in Remarks and examples below to read

`````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
marksample, mark, and markout assist in this.

Program ...
    (parse the arguments)
    (use mark* to create temporary variable ‘touse’ containing 0 or 1)
    rest of code ... if ‘touse’
end

marksample is for use in programs where the arguments are parsed using the syntax command; see [P] syntax. marksample creates a temporary byte variable, stores the name of the temporary variable in lmacname, and fills in the temporary variable with 0s and 1s according to whether the observation should be used. This determination is made by accessing information stored by syntax concerning the varlist, if exp, etc., allowed by the program. Its typical use is

Program ...
    syntax ...
    marksample touse
    rest of code ... if ‘touse’
end

mark starts with an already created temporary variable name. It fills in newmarkvar with 0s and 1s according to whether the observation should be used according to the weight, if exp, and in range specified. markout modifies the variable created by mark by resetting it to contain 0 in observations that have missing values recorded for any of the variables in varlist. These commands are typically used as

Program ...
    (parse the arguments)
    tempvar touse
    mark ‘touse’ ...
    markout ‘touse’ ...
    rest of code ... if ‘touse’
end

marksample is better than mark because there is less chance that you will forget to include some part of the sample restriction. markout can be used after mark or marksample when there are variables other than the varlist and when observations that contain missing values of those variables are also to be excluded. For instance, the following code is common:

Program ...
    syntax ... [, Denom(varname) ... ]
    marksample touse
    markout ‘touse’ ‘denom’
    rest of code ... if ‘touse’
end

Regardless of whether you use mark or marksample, followed or not by markout, the following rules apply:

1. The marker variable is set to 0 in observations for which weight is 0 (but see the zeroweight option).
2. The appropriate error message is issued, and everything stops if weight is invalid (such as being less than 0 in some observation or being a noninteger for frequency weights).
3. The marker variable is set to 0 in observations for which \textit{if exp} is not satisfied.

4. The marker variable is set to 0 in observations outside \textit{in range}.

5. The marker variable is set to 0 in observations for which any of the numeric variables in \textit{varlist} contain a numeric missing value.

6. The marker variable is set to 0 in all observations if any of the variables in \textit{varlist} are strings; see the \texttt{strook} option for an exception.

7. The marker variable is set to 1 in the remaining observations.

Using the name \texttt{touse} is a convention, not a rule, but it is recommended for consistency between programs.

\section*{Technical note}

\texttt{markin} is for use after \texttt{marksample}, \texttt{mark}, and \texttt{markout} and should be used only with extreme caution. Its use is never necessary, but when it is known that the specified \texttt{if exp} will select a small subset of the observations (small being, for example, 6 of 750,000), using \texttt{markin} can result in code that executes more quickly. \texttt{markin} creates local macro ‘\texttt{lclname}’ (or ‘\texttt{in}’ if \texttt{name()} is not specified) containing the smallest \texttt{in range} that contains the \texttt{if exp}.

By far the most common programming error—made by us at StataCorp and others—is to use different samples in different parts of a Stata program. We strongly recommend that programmers identify the sample at the outset. This is easy with \texttt{marksample} (or alternatively, \texttt{mark} and \texttt{markout}). Consider a Stata program that begins

\begin{verbatim}
program myprog
    version 16.1
    syntax varlist [if] [in]
    ... 
end
\end{verbatim}

Pretend that this program makes a statistical calculation based on the observations specified in \texttt{varlist} that do not contain missing values (such as a linear regression). The program must identify the observations that it will use. Moreover, because the user can specify \texttt{if exp} or \texttt{in range}, these restrictions must also be taken into account. \texttt{marksample} makes this easy:

\begin{verbatim}
version 16.1
syntax varlist [if] [in]
marksample touse
...
end
\end{verbatim}

To produce the same result, we could create the temporary variable \texttt{touse} and then use \texttt{mark} and \texttt{markout} as follows:

\begin{verbatim}
program myprog
    version 16.1
    syntax varlist [if] [in]
tempvar touse
    mark ‘touse’ ‘if’ ‘in’
    markout ‘touse’ ‘varlist’
    ...
end
\end{verbatim}

The result will be the same.
The `mark` command creates temporary variable ‘touse’ (temporary because of the preceding `tempvar`; see [P] macro) based on the if exp and in range. If there is no if exp or in range, ‘touse’ will contain 1 for every observation in the data. If if `price>1000` was specified by the user, only observations for which `price` is greater than 1,000 will have `touse` set to 1; the remaining observations will have `touse` set to 0.

The `markout` command updates the ‘touse’ marker created by `mark`. For observations where ‘touse’ is 1—observations that might potentially be used—the variables in `varlist` are checked for missing values. If such an observation has any variables equal to missing, the observation’s ‘touse’ value is reset to 0.

Thus observations to be used all have ‘touse’ set to 1. Including if ‘touse’ at the end of statistical or data management commands will restrict the command to operate on the appropriate sample.

Example 1

Let’s write a program to do the same thing as `summarize`, except that our program will also engage in casewise deletion—if an observation has a missing value in any of the variables, it is to be excluded from all the calculations.

```stata
cwsumm
program cwsumm
version 16.1
syntax [varlist(fv ts)] [if] [in] [aweight fweight] [, Detail noFormat]
marksample touse
markout ‘touse’ ...
summarize ‘varlist’ [‘weight’ ‘exp’] if ‘touse’, ‘detail’ ‘format’
end
```

Technical note

Let’s now turn to `markin`, which is for use in those rare instances where you, as a programmer, know that only a few of the observations are going to be selected, that those small number of observations probably occur close together in terms of observation number, and that speed is important. That is, the use of `markin` is never required, and a certain caution is required in its use, so it is usually best to avoid it. On the other hand, when the requirements are met, `markin` can speed programs considerably.

The safe way to use `markin` is to first write the program without it and then splice in its use. Form a `touse` variable in the usual way by using `marksample`, `mark`, and `markout`. Once you have identified the `touse` sample, use `markin` to construct an in range from it. Then add ‘in’ on every command in which if ‘touse’ appears, without removing the if ‘touse’.

That is, pretend that our original code reads like the following:

```stata
program ...
syntax ...
marksample touse
markout ‘touse’ ...
// touse now fully set
generate ... if ‘touse’
replace ... if ‘touse’
summarize ... if ‘touse’
replace ... if ‘touse’
...
end
```
We now change our code to read as follows:

    program ...
    syntax ...
    marksample toused
    markout 'touse' ...       // touse now fully set
    markin if 'touse'        // <- new
                           // we add 'in':
    generate ... if 'touse' 'in'
    replace ... if 'touse' 'in'
    summarize ... if 'touse' 'in'
    replace ... if 'touse' 'in'
    ...
    end

This new version will, under certain conditions, run faster. Why? Consider the case when the program is called and there are 750,000 observations in memory. Let’s imagine that the 750,000 observations are a panel dataset containing 20 observations each on 37,500 individuals. Let’s further imagine that the dataset is sorted by subjectid, the individual identifier, and that the user calls our program and includes the restriction if subjectid==4225.

Thus our program must select 20 observations from the 750,000. That’s fine, but think about the work that generate, replace, summarize, and replace must each go to in our original program. Each must thumb through 750,000 observations, asking themselves whether ‘touse’ is true, and 749,980 times, the answer is no. That will happen four times.

markin will save Stata work here. It creates a macro named ‘in’ of the form “in \( j_1/j_2 \)”, where \( j_1 \) to \( j_2 \) is the narrowest range that contains all the ‘touse’ \( \neq 0 \) values. Under the assumptions we made, that range will be exactly 20 long; perhaps it will be in 84500/84520. Now the generate, replace, summarize, and replace commands will each restrict themselves to those 20 observations. This will save them much work and the user much time.

Because there is a speed advantage, why not always use markin in our programs? Assume that between the summarize and the replace there was a sort command in our program. The in range constructed by markin would be inappropriate for our last replace; we would break our program. If we use markin, we must make sure that the in range constructed continues to be valid throughout our program (we construct a new one when it changes). So that is the first answer: you cannot add markin without thinking. The second answer is that markin takes time to execute, albeit just a little, and that time is usually wasted because in range will not improve performance because the data are not ordered as required. Taking the two reasons together, adding markin to most programs is simply not worth the effort.

When it is worth the effort, you may wonder why, when we added ‘in’ to the subsequent commands, we did not simultaneously remove if ‘touse’. The answer is that ‘in’ is not a guaranteed substitute for if. In our example, under the assumptions made, the ‘in’ happens to substitute perfectly, but that was just an assumption, and we have no guarantees that the user happens to have his or her data sorted in the desired way. If, in our program, we sorted the data, and then we used markin to produce the range, we could omit if ‘touse’, but even then, we do not recommend it. We always recommend programming defensively, and the cost of evaluating if ‘touse’, when ‘in’ really does restrict the sample to the relevant observations, is barely measurable.
Reference


Also see

[P] byable — Make programs byable
[P] syntax — Parse Stata syntax
[SVY] svymarkout — Mark observations for exclusion on the basis of survey characteristics
[U] 18 Programming Stata