Title

stata.com

Styles — Dataset styles

Description Syntax Remarks and examples Also see

Description

The purpose of this entry is to familiarize you with the four styles in which `mi` data can be stored.

Syntax

There are four dataset styles available for storing `mi` data:

`wide`

`mlong`

`flong`

`flongsep`

Remarks and examples

stata.com

Remarks are presented under the following headings:

The four styles

We have highly artificial data, which we will first describe verbally and then show to you in each of the styles. The original data have two observations on two variables:

a	b
1	2
4	.

Variable `b` has a missing value. We have two imputed values for `b`, namely, 4.5 and 5.5. There will also be a third variable, `c`, in our dataset, where `c` = `a` + `b`.

Thus, in the jargon of mi, we have $M = 2$ imputations, and the datasets $m = 0$, $m = 1$, and $m = 2$ are

m=0:

a	b	c
1	2	3
4	.	.

m=1:

a	b	c
1	2	3
4	4.5	8.5

m=2:

a	b	c
1	2	3
4	5.5	9.5

Continuing with jargon, a is a regular variable, b is an imputed variable, and c is a passive variable.

Style wide

The above data have been stored in miproto.dta in the wide style.

```
. use https://www.stata-press.com/data/r17/miproto
(mi prototype)
. list
```

	a	b	c	_1_b	_2_b	_1_c	_2_c	_mi_miss
1.	1	2	3	2	2	3	3	0
2.	4	.	.	4.5	5.5	8.5	9.5	1

There is no significance to the order in which the variables appear.

On the left, under variables a, b, and c, you can see the original data.

The imputed values for b appear under the variables named _1_b and _2_b; $m = 1$ appears under _1_b, and $m = 2$ appears under _2_b. Note that in the first observation, the observed value of b is simply repeated in _1_b and _2_b. In the second observation, however, _1_b and _2_b show the replacement values for the missing value of b.

The passive values for c appear under the variables named _1_c and _2_c in the same way that the imputed values appeared under the variables named _1_b and _2_b.

Finally, one extra variable appears: _mi_miss. This is an example of an mi system variable. You are never to change mi system variables; they take care of themselves. The wide style has only one system variable. _mi_miss contains 0 for complete observations and 1 for incomplete observations.

Style flong

Let's convert this dataset to style flong:

```
. mi convert flong, clear
. list, separator(2)
```

	a	b	c	_mi_miss	_mi_m	_mi_id
1.	1	2	3	0	0	1
2.	4	.	.	1	0	2
3.	1	2	3	.	1	1
4.	4	4.5	8.5	.	1	2
5.	1	2	3	.	2	1
6.	4	5.5	9.5	.	2	2

We listed these data with a separator line after every two rows so that they would be easier to understand. Ignore the `mi` system variables and focus on variables `a`, `b`, and `c`. Observations 1 and 2 contain $m = 0$; observations 3 and 4 contain $m = 1$; observations 5 and 6 contain $m = 2$.

We will now explain the system variables, but you do not need to remember this.

1. We again see `_mi_miss`, just as we did in the wide style. It marks the incomplete observations in $m = 0$. It contains missing in $m > 0$.
2. `_mi_m` records m. The first two observations are $m = 0$; the next two, $m = 1$; and the last two, $m = 2$.
3. `_mi_id` records an arbitrarily coded observation-identification variable. It is 1 and 2 in $m = 0$, and then repeats in $m = 1$ and $m = 2$. Observations `_mi_id` $= 1$ correspond to each other for all m. The same applies to `_mi_id` $= 2$.

   Warning: Do not use `_mi_id` as your own ID variable. You might look one time, see that a particular observation has `_mi_id` $= 8$, and look a little later, and see that the observation has changed from `_mi_id` $= 8$ to `_mi_id` $= 5$. `_mi_id` belongs to `mi`. If you want your own ID variable, make your own. All that is true of `_mi_id` is that, at any instant, it uniquely identifies, and ties together, the observations.

There is no significance to the order of the variables or, for that matter, to the order of the observations.

Style mlong

Let's convert this dataset to the mlong style:

```
. mi convert mlong, clear
. list
```

	a	b	c	_mi_miss	_mi_m	_mi_id
1.	1	2	3	0	0	1
2.	4	.	.	1	0	2
3.	4	4.5	8.5	.	1	2
4.	4	5.5	9.5	.	2	2

This listing will be easier to read if we add some carefully chosen blank lines:

	a	b	c	_mi_miss	_mi_m	_mi_id
1.	1	2	3	0	0	1
2.	4	.	.	1	0	2
3.	4	4.5	8.5	.	1	2
4.	4	5.5	9.5	.	2	2

The mlong style is just like flong except that the complete observations—observations for which `_mi_miss` $= 0$ in $m = 0$—are omitted in $m > 0$.

Observations 1 and 2 are the original, $m = 0$ data.

Observation 3 is the $m = 1$ replacement observation for observation 2.

Observation 4 is the $m = 2$ replacement observation for observation 2.

Style flongsep

Let's look at these data in the flongsep style:

```
. mi convert flongsep example, clear
(files example.dta _1_example.dta _2_example.dta created)
. list
```

	a	b	c	_mi_miss	_mi_id
1.	1	2	3	0	1
2.	4	.	.	1	2

The flongsep style stores $m = 0$, $m = 1$, and $m = 2$ in separate files. When we converted to the flongsep style, we had to specify a name for these files, and we chose `example`. This resulted in $m = 0$ being stored in `example.dta`, $m = 1$ being stored in `_1_example.dta`, and $m = 2$ being stored in `_2_example.dta`.

In the listing above, we see the original, $m = 0$ data.

After conversion, $m = 0$ (`example.dta`) was left in memory. When working with flongsep data, you always work with $m = 0$ in memory. Nothing can stop us, however, from taking a brief peek:

```
. save example, replace
file example.dta saved
. use _1_example, clear
(mi prototype)
. list
```

	a	b	c	_mi_id
1.	1	2	3	1
2.	4	4.5	8.5	2

There are the data for $m = 1$. As previously, system variable `_mi_id` ties together observations. In the $m = 1$ data, however, `_mi_miss` is not repeated.

Let's now look at _2_example.dta:

```
. use _2_example, clear
(mi prototype)
. list
```

	a	b	c	_mi_id
1.	1	2	3	1
2.	4	5.5	9.5	2

And there are the data for $m = 2$.

We have an aside, but an important one. Review the commands we just gave, stripped of their output:

```
. mi convert flongsep example, clear
. list
. save example, replace
. use _1_example, clear
. list
. use _2_example, clear
. list
```

What we want you to notice is the line `save example, replace`. After converting to flongsep, for some reason we felt obligated to save the dataset. We will explain below. Now look farther down the history. After using _1_example.dta, we did not feel obligated to resave that dataset before using _2_example.dta. We will explain that below, too.

The flongsep style data are a matched set of datasets. You work with the $m = 0$ dataset in memory. It is your responsibility to save that dataset. Sometimes `mi` will have already saved the dataset for you. That was true here after `mi convert`, but it is impossible for you to know that in general, and it is your responsibility to save the dataset just as you would save any other dataset.

The $m > 0$ datasets, _#_*name*.dta, are `mi`'s responsibility. We do not have to concern ourselves with saving them. Obviously, it was not necessary to save them here because we had just used the data and made no changes. The point is that, in general, the $m > 0$ datasets are not our responsibility. The $m = 0$ dataset, however, is our responsibility.

We are done with the demonstration:

```
. drop _all
. mi erase example
(files example.dta _1_example.dta _2_example.dta erased)
```

How we constructed this example

You might be curious as to how we constructed miproto.dta. Here is what we did:

```
. drop _all
. input a b
             a          b
  1. 1 2
  2. 4 .
  3. end
. mi set wide
. mi set M = 2
(2 imputations added; M = 2)
```

```
. mi register regular a
. mi register imputed b
. replace _1_b = 4.5 in 2
(1 real change made)
. replace _2_b = 5.5 in 2
(1 real change made)
. mi passive: generate c = a + b
m=0:
(1 missing value generated)
m=1:
m=2:
. order a b c _1_b _2_b _1_c _2_c _mi_miss
```

Using mi system variables

You can use `mi`'s system variables to make some tasks easier. For instance, if you wanted to know the overall number of complete and incomplete observations, you could type

```
. tabulate _mi_miss
```

because in all styles, the `_mi_miss` variable is created in $m = 0$ containing 0 if complete and 1 if incomplete.

If you wanted to know the summary statistics for `weight` in $m = 1$, the general solution is

```
. mi xeq 1: summarize weight
```

If you were using wide data, however, you could instead type

```
. summarize _1_weight
```

If you were using flong data, you could type

```
. summarize weight if _mi_m==1
```

If you were using mlong data, you could type

```
. summarize weight if (_mi_m==0 & !_mi_miss) | _mi_m==1
```

Well, that last is not so convenient.

What is convenient to do directly depends on the style you are using. Remember, however, you can always switch between styles by using `mi convert` (see [MI] **mi convert**). If you were using mlong data and wanted to compare summary statistics of the `weight` variable in the original data and in all imputations, you could type

```
. mi convert wide
. summarize *weight
```

Advice for using flongsep

Use the flongsep style when your data are too big to fit into any of the other styles. If you already have flongsep data, you can try to convert it to another style. If you get the error "no room to add more observations" or "no room to add more variables", then you need to increase the amount of memory Stata is allowed to use (see [D] **memory**) or resign yourself to using the flongsep style.

There is nothing wrong with the flongsep style except that you need to learn some new habits. Usually, in Stata, you work with a copy of the data in memory, and the changes you make are not reflected in the underlying disk file until and unless you explicitly save the data. If you want to change the name of the data, you merely save them in a file of a different name. None of that is true when working with flongsep data. Flongsep data are a collection of datasets; you work with the one corresponding to $m = 0$ in memory, and `mi` handles keeping the others in sync. As you make changes, the datasets on disk change.

Think of the collection of datasets as having one name. That name is established when the flongsep data are created. There are three ways that can happen. You might start with a non-`mi` dataset in memory and `mi set` it; you might import a dataset into Stata and the result be flongsep; or you might convert another `mi` dataset to flongsep. Here are all the corresponding commands:

```
. mi set flongsep name                    (1)
. mi import flongsep name                 (2)
. mi import nhanes1 name
. mi convert flongsep name                (3)
```

In each command, you specify a name and that name becomes the name of the flongsep dataset collection. In particular, *name*`.dta` becomes $m = 0$, `_1_`*name*`.dta` becomes $m = 1$, `_2_`*name*`.dta` becomes $m = 2$, and so on. You `use` flongsep data by typing `use` *name*, just as you would any other dataset. As we said, you work with $m = 0$ in memory and `mi` handles the rest.

Flongsep data are stored in the current (working) directory. Learn about `pwd` to find out where you are and about `cd` to change that; see [D] **cd**.

As you work with flongsep data, it is your responsibility to `save` *name*`.dta` almost as it would be with any Stata dataset. The difference is that `mi` might and probably has saved *name*`.dta` along the way without mentioning the fact, and `mi` has doubtlessly updated the `_#_`*name*`.dta` datasets, too. Nevertheless, it is still your responsibility to `save` *name*`.dta` when you are done because you do not know whether `mi` has saved *name*`.dta` recently enough. It is not your responsibility to worry about `_#_`*name*`.dta`.

It is a wonderful feature of Stata that you can usually work with a dataset in memory without modifying the original copy on disk except when you intend to update it. It is an unpleasant feature of flongsep that the same is not true. We therefore recommend working with a copy of the data, and `mi` provides an `mi copy` command (see [MI] **mi copy**) for just that purpose:

```
. mi copy newname
```

With flongsep data in memory, when you type `mi copy` *newname*, the current flongsep files are saved in their existing name (this is one case where you are not responsible for saving *name*`.dta`), and then the files are copied to *newname*, meaning that $m = 0$ is copied to *newname*`.dta`, $m = 1$ is copied to `_1_`*newname*`.dta`, and so on. You are now working with the same data, but with the new name *newname*.

As you work, you may reach a point where you would like to save the data collection under *name* and continue working with *newname*. Do the following:

```
. mi copy name, replace
. use newname
```

When you are done for the day, if you want your data saved, do not forget to save them by using `mi copy`. It is also a good idea to erase the flongsep *newname* dataset collection:

```
. mi copy name, replace
. mi erase newname
```

By the way, *name*`.dta`, `_1_`*name*`.dta`, ... are just ordinary Stata datasets. By using general (non-`mi`) Stata commands, you can look at them and even make changes to them. Be careful about doing the latter; see [MI] **Technical**.

See [MI] **mi copy** to learn more about `mi copy`.

Also see

[MI] **Intro** — Introduction to mi

[MI] **mi copy** — Copy mi flongsep data

[MI] **mi erase** — Erase mi datasets

[MI] **Technical** — Details for programmers