Title

stata.com

frames intro — Introduction to frames

Description Remarks and examples Also see

Description

Frames, also known as data frames, allow you to simultaneously store multiple datasets in memory. The datasets in memory are stored in frames, and Stata allows multiple frames. You can switch between them and even link data in them to data in other frames. How this works is presented below.

Remarks and examples

stata.com

Remarks are presented under the following headings:

What frames can do for you

Frames let you have multiple datasets in memory simultaneously. Here are a few ways you can use them.

Use frames to multitask

You can create a new frame, load another dataset into it, perform some task, switch back, and discard the frame.

You are working. The phone rings. Something has to be handled right now.

```
. frame create interruption          // you create new frame ...
. frame change interruption          // and switch to it

. use another_dataset                // you load a dataset
.                                    // you do what needs doing

. frame change default               // you switch back
. frame drop interruption            // you delete the new frame
```

You are back to work just as if you had never been interrupted.

Use frames to perform tasks integral to your work

You need to calculate a value from the data and add it to the data. This is troublesome because making the calculation requires modifying the data, the same data that need to be unmodified and have the result added to them.

You have loaded `yourdata.dta` into memory and have already made some updates to it. You have not yet saved those changes. You set about calculating the troublesome value.

```
. frame copy default subtask         // create & copy current data to new frame
. frame change subtask               // switch to the new frame

. sort weight foreign                // begin result calculation
. omitted steps
. keep if mark1 | mark2              // drop observations!
. omitted steps
. regress dmpg dw if mod(_n,2)       // calculate troublesome value

. frame change default               // switch back to previous frame
. gen dwc = cond(foreign,_b[dw],0)   // save result in yourdata.dta
. frame drop subtask                 // drop new frame
```

You could have used `preserve` and `restore` to perform this task. Using frames, however, is usually more convenient, if for no other reason than you can switch back and forth between them. You cannot do that with a preserved dataset and the modified copy in memory.

If you look carefully at the code above, you will notice that the troublesome value we needed to calculate and store was `_b[dw]`. `_b[dw]` was calculated from data in frame `subtask` and stored in Stata for subsequent use no matter which frame is current.

It is dataset values that are stored in frames. Programmatic values such as `_b[]`, `r()`, `e()`, and `s()` are stored in Stata and available across frames.

Use frames to work with separate datasets simultaneously

When we say working with datasets simultaneously, we mean datasets that are linked. Linked datasets are an alternative to merged datasets.

You have two datasets. `persons.dta` contains data on people. `uscounties.dta` contains data on counties. You want to analyze the people in `persons.dta` and the counties in which they live. There are issues in combining the two datasets:

1. Some of the people in `persons.dta` live in the same county.
2. There are counties in `uscounties.dta` that are irrelevant to your analysis because nobody in `persons.dta` lives in them.

3. You are not certain that `uscounties.dta` is complete. There might be some people in `persons.dta` that live in counties not recorded in `uscounties.dta`.
4. And beyond that, only some of the variables in `uscounties.dta` are needed for your analysis.

The frames solution to all of these problems is to link the two datasets. You start by loading `persons.dta` into one frame and `uscounties.dta` into another:

```
. use persons
. frame create uscounties
. frame uscounties: use uscounties
```

To link the datasets in the two frames, you type

```
. frlink m:1 countyid, frame(uscounties)
```

This matches the observations in `persons.dta` to those in `uscounties.dta` based on equal values of variable `countyid`. The data are not merged, they are linked. No variables from `uscounties.dta` are copied to `persons.dta`, but how the variables would be copied has been worked out.

You copy variables to the person data as you need them, one at a time, or in groups, using the `frget` command:

```
. frget med_income nschools, from(uscounties)
```

You can perform the desired analysis using `persons.dta`, the dataset in the current frame:

```
. regress income med_income n_schools educ age
```

Use frames to record statistics gathered from simulations

Simulations involve repeating a task—performing a simulation—each step of which produces statistics that are somehow recorded. After that, you analyze the recorded statistics.

The frames solution to the simulation problem is to collect the statistics in another frame. We will name that frame `results`. You start by creating a new frame and the variables in it to record the statistics, such as `b1coverage` and `b2coverage`:

```
. frame create results b1coverage b2coverage
```

new frame's name — \ — `results`

`b1coverage b2coverage` — / — *new variables in it*

The new frame contains zero observations at this point.

You will next write a do-file to create the values to be stored after each iteration. At the end of each iteration, the do-file will contain the line

```
. frame post results (exp1) (exp2)
```

frame's name — \ — `results`

(*exp*$_1$) (*exp*$_2$) — / — *values for* `b1coverage` and `b2coverage`

`frame post` adds an observation to the data in `results`. *exp*$_1$ and *exp*$_2$ are expressions.

When the do-file finishes, the completed set of results will be found in frame `results`. You will want to save them:

```
. frame results: save filename
```

You will then switch to the frame and begin your analysis of the statistics:

```
. frame change results
. summarize
```

Frames make Stata (preserve/restore) faster

Many programs written in Stata use the commands `preserve` and `restore` to temporarily save and later restore the contents of the data in memory. Programs that use `preserve` and `restore` now run faster if you are using Stata/SE or Stata/MP. They run faster because Stata preserves data by copying them to hidden frames. Those hidden frames are stored in memory. Copying data to frames stored in memory takes a lot less time than copying data to disk.

More correctly, `preserve` copies data to hidden frames unless memory is in short supply. If it is, `preserve` resorts to storing them on disk. That is temporary because later, as datasets are restored, memory will again become available and `preserve` will return to preserving them in hidden frames.

This is all automatic, but you may want to reset the value of `max_preservemem`, which controls this behavior. When the amount stored in hidden frames would exceed `max_preservemem`, Stata preserves subsequent datasets on disk. Out of the box, `max_preservemem` is set to 1 gigabyte. Perhaps you or someone else has already changed that. To find out the current value of `max_preservemem`, type

```
. query memory
```

If you want to change `max_preservemem` to 2 gigabytes for the duration of the session, type

```
. set max_preservemem 2g
```

You can set the value up or down. You could set it to `4g` or `50m`. You could even set it to `0`, and then all datasets would be preserved to disk.

If you want to set `max_preservemem` to 2 gigabytes permanently, for this session and future Stata sessions, type

```
. set max_preservemem 2g, permanently
```

Other uses will occur to you that we should have listed

Frames make doing lots of tasks more convenient, and you will find your own uses for them. Frames make code faster too. Manipulating objects stored in memory takes less computer time than manipulating disk files.

Learning frames

Here is a tutorial on using frames. In the tutorial, we will sometimes show you a syntax diagram. For example, we might show you

`frame copy` *framename newframename*

When we show syntax diagrams in the tutorial, they are not always the full syntax diagrams. `frame copy`, for instance, also allows a `replace` option, and we might not only not show it in the syntax diagram but also not even mention it. You can click on the command to see the full syntax.

The current frame

Everything hinges on the *current frame*. Stata commands use the data in the current frame. When you load a dataset,

```
. sysuse auto
(1978 Automobile Data)
```

you are loading it into the current frame. Which frame is that? Type `frame` to discover its identity:

```
. frame
  (current frame is default)
```

You can type `frame` or type `pwf`, which is a synonym for `frame`. The letters stand for "print working frame". We will type `frame` in this tutorial, but you may prefer to type `pwf` because it is shorter. Other `frame` commands also have shorter synonyms. We will mention them as we go along.

We just discovered that the current frame is named `default`. When Stata is launched, that is what it names the frame it creates for you. You cannot change that, but `default` is just a name, and you can rename frames if you wish. You can create other frames too. You can create up to 100 of them.

To rename a frame, use the `frame rename` command:

`frame rename` *oldname newname*

To rename the frame `default` to `genesis`, type

```
. frame rename default genesis
. frame
  (current frame is genesis)
```

Frames can be renamed whether Stata created them or you did. They can be renamed whether they have data in them or they are empty. Renaming `default` will not break anything subsequently. Stata commands operate on the current frame, whatever its name.

Creating new frames

Create new frames using the `frame create` command:

`frame create` *newframename*

We will show you an example in a minute. First, however, if you are going to create a frame with a new name, you need to know how to find out the names of the frames that currently exist. You do that using the `frames dir` command:

`frames dir`

We recall that we renamed our default frame, but we cannot recall the name that we used. So what frames are in memory?

```
. frames dir
  genesis  74 x 12; 1978 Automobile Data
```

There is one frame in memory, named `genesis`. It contains a dataset that is 74×12, meaning 74 observations and 12 variables. The dataset has a dataset label "1978 Automobile Data", but if it did not, the dataset's name, `auto.dta`, would have appeared in its place in `frames dir`'s output, unless the data had never been saved to disk. In that case, nothing would have appeared where "1978 Automobile Data" appeared.

Now let's create a new frame named `second`:

```
. frame create second
. frame dir
  genesis  74 x 12; 1978 Automobile Data
  second   0 x 0
```

There are now two frames in memory. The new frame is 0×0. It is empty.

By the way, `frame create` has a shorter synonym, `mkf`. The letters stand for "make frame". We could have typed `mkf second` to make the new frame.

Type frame or frames, it does not matter

You probably did not notice, but we have used `frames dir` twice so far, but we typed it differently the second time. We typed

```
. frames dir
. frame dir
```

Stata does not care whether you type `frame` or `frames`. This indifference applies to all the `frames`/`frame` commands.

Switching frames

`frame change` (synonym: `cwf` for "change working frame") switches the identity of the current frame:

`frame change` *framename*

We could make `second` the current frame and switch back to `genesis` again:

```
. frames change second
. count
  0
. cwf genesis
. count
  74
```

We used Stata's `count` command to demonstrate that the current frame really switched. `count` without arguments displays the number of observations.

Copying frames

There are two commands for copying frames:

`frame copy` *framename newframename*

`frame put` *varlist*`, into(`*newframename*`)`

`frame put` *if*`, into(`*newframename*`)`

`frame copy` copies the entire dataset.

`frame put` copies subsets of the dataset.

In either case, the commands create the frame being copied to.

Dropping frames

To drop an existing frame, type

```
frame drop framename
```

Resetting frames

Resetting frames means the following:

1. Drop all the data in all the frames, even if the data have not been saved since they were last saved.
2. Drop (delete) all the frames.
3. Create a new frame named `default`, and make it the current frame.

Each of the following commands resets frames:

```
frames reset
clear frames

clear all
```

`frames reset` and `clear frames` are synonyms.

`clear all` resets the frames and does more. It returns Stata to as close to just-after-launch status as possible.

Frame prefix command

The `frame` prefix command is perhaps the most convenient of the `frame` commands. Its syntax command is

```
frame framename: stata_command
```

The `frame` prefix command 1) changes the current frame to the frame specified, 2) executes *stata_command*, and 3) changes the current frame back to what it was.

For instance, say the current frame is `default` and we have a second frame named `second`. We type

```
. frame second: sysuse census, clear
```

The result would be that frame `second` would contain `census.dta` and the current frame would still be `default`, just as if we had typed

```
. frame change second
. sysuse census, clear
. frame change default
```

Frame prefix has a second feature too. Imagine that in doing the above, we omitted the `clear` option when we use the data. Consider what would have happened if we set about typing the three commands but the data in `second` had changed since they were last saved:

```
. frame second
. sysuse census
no; dataset in memory has changed since last saved
r(4);
```

What is the current frame? It is `second`, of course, because we changed to it. Now consider making the same mistake using the `frame` prefix approach:

```
. frame second: sysuse census
no; dataset in memory has changed since last saved
r(4);
```

Even though an error occurred, the current frame is still `default`! To recover from the error, we do not have to change back to the original frame. The `frame` prefix command did that for us.

`frame` prefix has another syntax when you have more than one command to be executed:

```
frame framename {
        stata_command
        stata_command
        .
        .
}
```

This syntax is especially useful in programs.

Linking frames

When we say linking, we mean linking as shown in the earlier example when we had separate datasets on people and counties and combined them in a merged-data kind of way. Linking can do a lot more than we showed you.

In [D] **frlink**, we show you how to create a nested linkage to link students (one dataset) to the schools they attend (a second dataset) and to the counties (a third dataset) in which their schools are located. We show you an example of linking a generational dataset with itself, so that adult children are linked to their parents and grandparents, a total of six simultaneous linkages!

Linkages are created by using the `frlink` command. Its simplest syntaxes are

`frlink m:1` *varlist*`, frame(`*framename*`)`

`frlink 1:1` *varlist*`, frame(`*framename*`)`

These syntaxes create an `m:1` or `1:1` link between the current frame and *framename* based on observations having equal values of *varlist*.

Once a link is created, you can use the `frget` command to copy the appropriate values of variables from *framename* to the current frame. Its syntaxes are

`frget` *varlist*`, from(`*linkagename*`)`

`frget` *newvar* `=` *varname*`, from(`*linkagename*`)`

You can use the `frval()` function in expressions to access appropriate observations of variables in the linked data. Its syntax is

`... frval(`*linkagename*`,`*varname*`) ...`

Ignore the _frval() function

While we are on the subject of the `frval()` function, we should warn you. Also available in [FN] **Programming functions** is `_frval()`. Ignore it. `frval()` is better. `_frval()` is for use by programmers.

Posting new observations to frames

We used posting to perform simulations in an example earlier. That is one use of it. More generally, posting solves problems that require transferring data or values from one frame to a new observation in another.

First, you prepare the other frame to receive the data. `frame create`, which we already discussed, has a syntax for doing this. We showed you its first syntax, which is

`frame create` *newframename*

The second syntax is

`frame create` *newframename* *newvarlist*

This syntax creates the new frame and creates in it a zero-observation dataset of the new variables specified. *newvarlist* really is a new varlist, and that means that you can specify variables types and variable names. You could type

```
. frame create results strL(rngstate) double(b1coverage b2coverage)
```

Alternatively, you can use `frame create`'s first syntax to create the frame, use `frame change` to switch to it, and create the zero-observation dataset yourself. Then, you can switch back to what was the current frame.

`frame post` adds observations to the second frame. Its syntax is

`frame post` *framename* (*exp*) (*exp*) ... (*exp*)

The expressions are in the same order as the variables in the second frame.

Programming with frames

Below we discuss writing Stata programs that deal with multiple frames.

If you are not interested in writing such programs, stop reading.

What follows is not a tutorial. What follows are numbered lists detailing everything you need to know to write programs that use more than the current frame. That program could implement a command that does something with frames specified by users. Or it could do something that, as far as users are concerned, uses only the current frame and hidden from them is that your program uses frames to accomplish certain internal tasks.

We also want to emphasize there still exists a place for programs written in Stata that do not use frames at all. Perhaps most programs are like that.

Ado-programming with frames

1. `tempnames`.

   Frames with names created by `tempname` are automatically dropped (deleted) when the program generating the temporary name ends.

   If the program you write is to create a new frame for the user, give the frame a `tempname` in your program, and, at the end, use `frame rename` to change its name. This way, if an error occurs, the frame the program may have been in the midst of creating will be dropped automatically.

2. Current frame.

   Stata provides the name of the current frame in creturn result c(frame). You can obtain the name of the current frame by coding

```
local curframe = c(frame)
```

   Programs that use frames invariably change frames during their execution. Programs need to ensure the appropriate frame is the current one at the time the program exits. This includes when the program is successful and when it exits with error.

   The successful case is easy enough to handle. At the point your program exits, set the current frame appropriately. In general, the current frame should be the same as the current frame was when the program started.

   Error cases can be more difficult. Who knows when the user will press break or when the bug buried in your code will bite? The code could be doing literally anything. Even so, your program needs to ensure that the current frame is set appropriately. There is a style of programming that does this.

   Case 1: You are writing new command foo. foo uses frames but in all cases is to leave the current frame the same as it was initially. The code reads as follows:

```
program foo
        version ...
            local curframe = c(frame)
            frame `curframe' {
                 foo_cmd `0'
            }
end
```

   Write foo_cmd as you usually would. As you write foo_cmd, you can ignore the current-frame problem. You can use frame change freely in foo_cmd and its subroutines. No matter what happens, error or success, the program will end with the current frame unchanged.

   Case 2: You are writing new command foo. If foo is successful, the new frame will change. The code reads as follows:

```
program foo
        version ...
            local curframe = c(frame)
            frame `curframe' {
                 foo_cmd `0'
            }
      frame change `s(frame)'
end
```

   Write foo_cmd as you usually would. If execution is successful, however, foo_cmd must sreturn in s(frame) the name of the frame that is to be the current frame. As with case 1, you can use frame change freely in foo_cmd and all of its subroutines.

3. preserve and restore.

   For end users, using frames is sometimes a better alternative to using preserve and restore. Programmers should not, however, interpret that as preserve and restore are out of date and not to be used in frame programming. preserve and restore in programming have the same valid use they have always had.

   Before frames existed in Stata, a single program could have at most one active preserve in it. Active means not canceled by restore or restore, not. A program could preserve, later restore or restore, not, and then preserve again. It would be odd but allowed.

Nowadays, a single program can have up to one active `preserve` for each frame. If a program deals with frames `'one'` and `'two'` and it is necessary, it can `preserve` both of them. `preserve` preserves the current frame. To preserve frames `'one'` and `'two'`, code,

```
frame 'one': preserve
frame 'two': preserve
```

When frames are automatically restored at the end of the program, both frames will be restored.

If you wish to restore frame `'one'` early and cancel its automatic restoration when the program ends, code

```
frame 'one': restore
```

If you instead wish to restore frame `'one'` now and still have it restored when the program ends, code

```
frame 'one': restore, preserve
```

If you instead wish simply to cancel the restoration of frame `'one'` when the program ends, code

```
frame 'one': restore, not
```

In all three cases, frame `'two'` will still be restored when the program ends.

Any uncanceled automatic restorations when the program ends will re-create any frames that have been dropped (deleted). Automatic restoration does not change the identity of the current frame.

Mata programming with frames

1. `st_frame*()` functions.

   Mata provides a suite of frame-related functions. They can change frames, create frames, drop frames, etc.

2. `st_data()`, `st_sdata()`, `_st_data()`, and `_st_sdata()` functions.

   Calls to `st_data()` and its associated functions return the data from the current frame. If you want data from other frames, change to the other frame first using `st_framecurrent()`.

3. `st_view()` and `st_sview()` functions.

   Views are views onto the frame that was current at the time the view was created by `st_view()` or `st_sview()`, and they remain that after creation even when the identity of the current frame changes. If `X` is a view onto frame `default`, it remains a view onto frame `default` even if the current frame changes.

   Views are how data can be copied between frames. Create a view onto the data in one frame. Create another view onto the data in the other. Use one view to update the other.

Also see

[D] **frames** — Data frames

[D] **frget** — Copy variables from linked frame

[D] **frlink** — Link frames

[FN] **Programming functions**

[M-5] **st_frame*( )** — Data frame manipulation