version — Version control

Description

At the top of every do-file or program that you create, type

```
version 19.5
```

Or, if you do not have StataNow, type

```
version 19
```

That single step ensures that your do-file or program will continue to run in all future versions of Stata, even if that future version has changes in the syntax of some of the commands or programming constructs that you use in your do-file or program. The few other ways to use `version` are in support of this functionality and are unimportant by comparison.

Syntax

Display version number to which command interpreter is set

```
version
```

Set command interpreter to version #

```
version #
```

```
version #: command
```

Set supplemental version number, rarely used

```
version #, user
```

```
version #, user: command
```

Option

`user` sets the version for a supplemental form of version control; see *User version* below.

Remarks and examples

Remarks are presented under the following headings:

Version

Stata is continually improving, and sometimes that means that commands or language elements in the interpreter need to change. `version` ensures that do-files, ado-files, and other programs continue to work. They will continue to work in all future versions of Stata regardless of the version of Stata in which they were written.

`version` # sets the interpretation of all language elements and commands to be the same as it was in version #. All do-files, ado-files, and other programs written for the current version of Stata should include as the first executable statement `version 19.5` or, if you do not have StataNow, `version 19`.

Here is what that looks like in a do-file:

```
begin mydofile.do
version 19.5      // (or version 19 if you do not have StataNow)
...
contents of your do-file
...
end mydofile.do
```

Here is what that looks like in an ado-file:

```
begin mycommand.ado
program
    version 19.5      // (or version 19 if you do not have StataNow)
    ...
    contents of your ado program
    ...
end
end mycommand.ado
```

For programs outside of ado-files, versioning looks just like it does for programs in ado-files.

When Stata first starts, it sets its version number to the current version of Stata, which is 19.5 in StataNow. Typing `version` without arguments displays the current setting of version:

```
. version
version 19.5
```

The `version` command simply sets this current version to a different value.

```
. version 8.0
. version
version 8.0
```

That is all Stata needs to ensure that your do-files, ado-files, and other programs continue to work in all future versions of Stata.

Version a single command

You can version individual commands by prefixing them with `version` #`:`. For example, you can use the old `anova` syntax by typing

```
. version 10: anova ...
```

This single command sets Stata's version to 10, runs the `anova` command, and then resets Stata's version to whatever it was before the command was executed. This usage is much less common.

User version

There is a supplement to version control called "user version". User version works alongside `version`, and you are unlikely to ever care about or need to separately control the user version. Very few people need to read this section.

Stata has always had version control. User version was created later to handle big improvements to some commands and functions that we wanted all existing ado-files and other programs to use, regardless of the `version` settings in those existing programs.

User version is not affected by `version` statements in ado-files and other programs. User version is only reset when `version` # is typed interactively or it appears in a do-file.

The argument is that users type commands and users write do-files to script sets of commands. If the user says they want version #, then they always get version #, even if there are some improvements. Programmers write programs that users then run to perform their tasks. Programmers primarily use version control to be sure their programs continue to run, even when Stata syntax is changed. If there is an improvement that does not affect syntax or stored results, then programmers will not care whether that improvement is included in their program.

For some features, the user should be in control of which version of that feature is run, not the programmer—thus, user version.

Let's consider some improvements that have been put under user-version control. To date, there have been only three:

1. The KISS random number was replaced by the 64-bit Mersenne Twister random number starting in version 14.
2. How Stata interprets factor-variable statements like `i(2 3).rep78` was improved in version 11.
3. A faster algorithm for `sort` was introduced in version 17.

We will ignore item 2 because it is difficult to explain succinctly, but be assured you are glad we made the change. And despite how things sound, it was not a syntactical change that affected programs.

Items 1 and 3 are simply improvements. The Mersenne Twister has even better statistical properties than the already great properties of the KISS random-number generator. And it has a whoppingly long period. Regarding item 3, a faster `sort` is just always better. Moreover, neither change has syntactical implications for previously written code.

Why not make these three changes new defaults regardless of the version set in do-files or interactively? The changes are not without consequence. The new random-number generator returns different sequences of numbers. The new sorter orders observations with tied sort keys differently.

A user cares about these consequences because the results will change. A programmer does not care because the program is simply grabbing the random numbers or performing the sort on behalf of the user. It is the version that the user wants to run that is important.

For example, changing the random-number generator clearly affects the results of `bootstrap` and `mi impute` because those commands draw random numbers. The commands do not, however, truly care how those numbers are drawn so long as they have good properties. The user, on the other hand, needs to know exactly how and what is drawing the random numbers—first, so they can be confident that the random-number generator is fit for their purpose, and second, because they may need to reproduce the results.

The dichotomy between the role of user and programmer is not perfect. Often, they are the same person. More importantly, sometimes we write programs as part of our analyses and sometimes we write do-files that are really more like tools or programs.

We can handle those cases. If you have a program and you want it to use the random-number generator, the factor-variables interpreter, and the sorter that were the defaults as of version 12, then put

```
version 12, user
```

at the top of your program. You are probably wanting to set everything to version 12, so you will need two version statements at the top of your program:

```
version 12
version 12, user
```

That sets both version and user version to 12. `version 12, user` does not set the normal version.

If you want version 12 to remain the version for the entirety of your program, and you likely do, then you do not have to do anything more. When your program concludes, for any reason, Stata will reset the version and user version to their state prior to running your program.

If you ever want to know the current user version, it is stored in `c(userversion)`.

If you have an old do-file and it is, say, version 12, but you want it to use a modern sort, you will need to change each `sort` command. This is an uncommon desire. Regardless, find each `sort` command and prepend it with `version 19:` as follows:

```
version 19: sort ...
```

Note that the do-file will now run only in Stata 19 and beyond. Stata 12 cannot run version 19 commands. How could it? That is fine; you have obviously already moved to Stata 19. And you did not put that version statement at the top of your do-file to help it run in Stata 12. You put it there so that all the commands in the file would continue to work using Stata 12 syntax no matter which future version of Stata you were running.

Version and random numbers

As of Stata 14, Stata's RNGs were improved, renamed, and restructured; see [R] **set seed**. The default RNG in modern Stata is the 64-bit Mersenne Twister (`mt64`). Before Stata 14, the RNG was the 32-bit KISS (`kiss32`). If user version is 14 or higher, RNG results are based on `mt64`. If user version is less than 14, RNG results are based on `kiss32`. This also affects the results of commands that use the RNGs, such as `bootstrap`, `bsample`, and the `mi` suite of commands.

Version control within a RNG is specified at the time the `set seed` command is given, not at the time the random-number generation function such as `rnormal()` is used. For instance, typing

```
. (assume version is set to be 11.2)
. set seed 123456789
. any_command ...
```

causes *any_command* to use the Stata 11.2 version of `rnormal()` even if *any_command* is an ado-file containing an explicit `version` statement setting the version to less than 11.2. This occurs because the version of `rnormal()` that is used was determined at the time the seed was set, and the seed was set under version 11.2 or later.

This works in both directions. Consider

```
. version 11.1: set seed 123456789
. any_command ...
```

In this case, *any_command* uses the older version of `rnormal()` because the seed was set under version 11.1, before `rnormal()` was updated. *any_command* uses the older version of `rnormal()` even if *any_command* itself includes an explicit `version` statement setting the version to 11.2 or later.

Thus both older and newer ado-files can use the newer or older `rnormal()`, and they can do so without modification. The only case in which you need to modify a do-file or ado-file is when it is older, it contains `set seed`, and you now want it to use the new `rnormal()`. In that case, find the `set seed` command in the do-file or ado-file,

```
version 10              // for example
...
set seed 123456789
...
```

and change it to read

```
version 10              // for example
...
version 11.2: set seed 123456789
...
```

You need to change only the one line.

Everything written above about prefixing `set seed` with a `version` is irrelevant if you are restoring the seed to a state previously obtained from `c(rngstate)`:

```
set rngstate X075bcd151f123bb5159a55e50022865700023e53
```

The string state `X075bcd151f123bb5159a55e50022865700023e53` includes the version number at the time the seed was set. Prefixing the above with `version`, whether older or newer, will do no harm but is unnecessary.

For an up-to-date summary of version changes, see `help version`.

Reference

Correia, S., and M. P. Seay. 2024. require: Package dependencies for reproducible research. *Stata Journal* 24: 599–613.

Also see

[U] 18.11.1 Version