Title

stata.com

set seed — Specify random-number seed and state

Description

`set seed #` specifies the initial value of the random-number seed used by the random-number functions, such as `runiform()` and `rnormal()`.

`set rngstate` *statecode* resets the state of the random-number generator to the value specified, which is a state previously obtained from `creturn` value `c(rngstate)`.

`set seed #` and `set rngstate` *statecode* apply to the current random-number generator. Every random-number generator in Stata has its own seed and state encoding.

Syntax

```
set seed #

set rngstate statecode
```

is any number between 0 and $2^{31} - 1$ (or 2,147,483,647).

statecode is a random-number state previously obtained from `creturn` value `c(rngstate)`.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Examples

1. Specify initial value of random-number seed

```
. set seed 339487731
```

2. Create variable `u` containing uniformly distributed pseudorandom numbers on the interval $(0, 1)$

```
. generate u = runiform()
```

3. Create variable `z` containing normally distributed random numbers with mean 0 and standard deviation 1

```
. generate z = rnormal()
```

4. Obtain state of pseudorandom-number generator and store it in a local macro named `state`

```
. local state = c(rngstate)
```

5. Restore pseudorandom-number generator state to that previously stored in local macro named `state`

```
. set rngstate 'state'
```

Setting the seed

Stata's random-number generation functions, such as `runiform()` and `rnormal()`, do not really produce random numbers. These functions are deterministic algorithms that produce numbers that can pass for random. `runiform()` produces numbers that can pass for independent draws from a rectangular distribution over $(0, 1)$; `rnormal()` produces numbers that can pass for independent draws from $N(0, 1)$. Stata's random-number functions are formally called pseudorandom-number functions. The default pseudorandom-number generator introduced in Stata 14 is the 64-bit Mersenne Twister. See Matsumoto and Nishimura (1998) and *Random-number generators in Stata* in [R] **set rng** for more details.

The sequences the random-number functions produce are determined by the seed, which is just a number and which is set to 123456789 every time Stata is launched. This means that `runiform()` produces the same sequence each time you start Stata. The first time you use `runiform()` after Stata is launched, `runiform()` returns 0.348871704556195. The second time you use it, `runiform()` returns 0.266885709753138. The third time you use it, ...

To obtain different sequences, you must specify different seeds using the `set seed` command. You might specify the seed 472195:

```
. set seed 472195
```

If you were now to use `runiform()`, the first call would return 0.713028143573182, the second call would return 0.920524469911484, and so on. Whenever you `set seed 472195`, `runiform()` will return those numbers the first two times you use it.

Thus you set the seed to obtain different pseudorandom sequences from the pseudorandom-number functions.

If you record the seed you set, pseudorandom results such as results from a simulation or imputed values from `mi impute` can be reproduced later. Whatever you do after setting the seed, if you set the seed to the same value and repeat what you did, you will obtain the same results.

How to choose a seed

Your best choice for the seed is an element chosen randomly from the set $\{0, 1, \ldots, 2^{31} - 1\}$ (where $2^{31} - 1 = 2{,}147{,}483{,}647$). We recommend that, but that is difficult to achieve because finding easy-to-access, truly random sources is difficult.

One person we know uses digits from the serial numbers from dollar bills he finds in his wallet. Of course, the numbers he obtains are not really random, but they are good enough, and they are probably a good deal more random than the seeds most people choose. Some people use dates and times, although we recommend against that because, over the day, it just gets later and later, and that is a pattern. Others try to make up a random number, figuring if they include enough digits, the result just has to be random. This is a variation on the five-second rule for dropped food, and we admit to using both of these rules.

It does not really matter how you set the seed, as long as there is no obvious pattern in the seeds that you set and as long as you do not set the seed too often during a session.

Nonetheless, here are two methods that we have seen used but you should not use:

1. The first time you set the seed, you set the number 1. The next time, you set 2, and then 3, and so on. Variations on this included setting 1001, 1002, 1003, . . . , or setting 1001, 2001, 3001, and so on.

   Do not follow any of these procedures. The seeds you set must not exhibit a pattern.

2. To set the seed, you obtain a pseudorandom number from `runiform()` and then use the digits from that to form the seed.

   This is a bad idea because the pseudorandom-number generator can converge to a cycle. If you obtained the pseudorandom-number generator unrelated to those in Stata, this would work well, but then you would have to find a rule to set the first generator's seed.

Choosing seeds that do not exhibit a pattern is of great importance. That the seeds satisfy the other properties of randomness is minor by comparison.

Do not set the seed too often

We cannot emphasize this enough: Do not set the seed too often.

To see why this is such a bad idea, consider the limiting case: You set the seed, draw one pseudorandom number, reset the seed, draw again, and so continue. The pseudorandom numbers you obtain will be nothing more than the seeds you run through a mathematical function. The results you obtain will not pass for random unless the seeds you choose pass for random. If you already had such numbers, why are you even bothering to use the pseudorandom-number generator?

The definition of too often is more than once per problem.

If you are running a simulation of 10,000 replications, set the seed at the start of the simulation and do not reset it until the 10,000th replication is finished. The pseudorandom-number generators provided by Stata have long periods. The longer you go between setting the seed, the more random-like are the numbers produced.

It is sometimes useful later to be able to reproduce in isolation any one of the replications, and so you might be tempted to set the seed to a known value for each of the replications. We negatively mentioned setting the seed to 1, 2, . . . , and it is in exactly such situations that we have seen this done. The advantage, however, is that you could reproduce the fifth replication merely by setting the seed to 5 and then repeating whatever it is that is to be replicated. If this is your goal, you do not need to reset the seed. You can record the state of the random-number generator, save the state with your replication results, and then use the recorded states later to reproduce whichever of the replications that you wish. This will be discussed in *Preserving and restoring the random-number generator state*.

There is another reason you might be tempted to set the seed more than once per problem. It sometimes happens that you run a simulation, let's say for 5,000 replications, and then you decide you should have run it for 10,000 replications. Instead of running all 10,000 replications afresh, you decide to save time by running another 5,000 replications and then combining those results with your previous 5,000 results. That is okay. We at StataCorp do this kind of thing. If you do this, it is important that you set the seed especially well, particularly if you repeat this process to add yet another 5,000 replications. It is also important that in each run there be a large enough number of replications, which is say thousands of them.

Even so, do not do this: You want 500,000 replications. To obtain them, you run in batches of 1,000, setting the seed 500 times. Unless you have a truly random source for the seeds, it is unlikely

you can produce a patternless sequence of 500 seeds. The fact that you ran 1,000 replications in between choosing the seeds does not mitigate the requirement that there be no pattern to the seeds you set.

In all cases, the best solution is to set the seed only once and then use the method we suggest in the next section.

Preserving and restoring the random-number generator state

In the previous section, we discussed the case in which you might be tempted to set the seed more frequently than otherwise necessary, either to save time or to be able to rerun any one of the replications. In such cases, there is an alternative to setting a new seed: recording the state of the pseudorandom-number generator and then restoring the state later should the need arise.

The state of the default random-number generator in Stata, the 64-bit Mersenne Twister, is a string of about 5,000 characters. The state can be displayed by typing `display c(rngstate)`. It is more practical to save the state, say, in a local macro named `state`:

```
. local state = c(rngstate)
```

The state can later be restored by typing

```
. set rngstate 'state'
```

The state string specifies an entry point into the sequence produced by the pseudorandom-number generator. Let us explain.

The best way to use a pseudorandom-number generator would be to choose a seed once, draw random numbers until you use up the generator, and then get a new generator and choose a new key. Pseudorandom-number generators have a period, after which they repeat the original sequence. That is what we mean by using up a generator. The period of the 64-bit Mersenne Twister, the default pseudorandom-number generator in Stata, is $2^{19937} - 1$. This is roughly 10^{6000}. It is difficult to imagine that you could ever use up this generator.

The string reported by `c(rngstate)` is an encoded form of the information necessary for Stata to reestablish exactly where it is located in the pseudorandom-number generator's sequence.

We are not seriously suggesting you choose only one seed over your entire lifetime, but let's look at how you might do that. Sometime after birth, when you needed your first random number, you would set your seed,

```
. set seed 1073741823
```

On that day, you would draw, say, 10,000 pseudorandom numbers, perhaps to impute some missing values. Being done for the day, you can save the state, and then later restore it.

When you type `set rngstate` followed by a saved state string, Stata reestablishes the previous state. Thus the next time you draw a pseudorandom number, Stata will produce the 10,001st result after setting seed 1073741823. Let's assume that you draw 100,000 numbers this day. Done for the day, you save the state string.

On the third day, after setting the state to the saved state string above, you will be in a position to draw the 110,001st pseudorandom number.

In this way, you would eat your way through the $2^{19937} - 1$ random numbers, but you would be unlikely ever to make it to the end.

We do not expect you to set the seed just once in your life, but using the state string makes it easy to set the seed just once for a problem.

When we do simulations at StataCorp, we record `c(rngstate)` for each replication. Just like everybody else, we record results from replications as observations in datasets; we just happen to have an extra variable in the dataset, namely, a string variable named `state`. That string is filled in observation by observation from the then-current values of `c(rngstate)`, which is a function and so can be used in any context that a function can be used in Stata.

Anytime we want to reproduce a particular replication, we thus have the information we need to reset the pseudorandom-number generator, and having it in the dataset is convenient because we had to go there anyway to determine which replication we wanted to reproduce. If we want to add more replications later, we have a state string that we can use to continue from where we left off.

Reference

Matsumoto, M., and T. Nishimura. 1998. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* 8: 3–30.

Also see

[R] **set** — Overview of system parameters

[R] **set rng** — Set which random-number generator (RNG) to use

[R] **set rngstream** — Specify the stream for the stream random-number generator

[FN] **Random-number functions**

[P] **version** — Version control