**dyngen** — Dynamically generate new values of variables

Description Menu Syntax Option Remarks and examples Also see

## Description

`dyngen` replaces the value of variables when two or more variables depend on each other's lagged values. Use `dyngen` when the values for the whole set of variables must be computed for an observation before moving to the next observation.

## Menu

Data > Create or change data > Dynamically generate new values

## Syntax

```
dyngen {
    update varname_1 = exp [ if ] [ , missval(#) ]
    ⋮
    update varname_N = exp [ if ] [ , missval(#) ]
} [ if ] [ in ]
```

$varname_n$, $n = 1, \ldots, N$, must already exist in the dataset and cannot be an alias variable; see [D] **frunalias**.
*exp* must be a valid expression and may include time-series operators; see **[U] 11.4.4 Time-series varlists**.

## Option

`missval(#)` specifies the value to use in place of missing values when performing calculations. This option is particularly useful when referring to lags that exist prior to the data.

## Remarks and examples

Like `replace`, `dyngen` modifies the contents of existing variables. However, `dyngen` works observation by observation. If you are doing a computation only on a single variable that relies only on its own lagged values or those of other variables, you do not need `dyngen` because `generate` and `replace` work their way through the data sequentially. Use `dyngen` when you need to modify two or more variables at the same time.

The examples in this entry use the following data:

```
. input time x1 x2
         time         x1         x2
  1. 1   3   1
  2. 2   4   4
  3. 3   5   2
  4. 4   5   1
  5. 5   2   1
  6. end
```

## Example 1: Using dyngen

We want to update our values of x1 and x2 such that x1 depends on its current value and the previous value of x2, and x2 depends on previous values of x1 and x2. We will be using these same values of x1 and x2 in subsequent examples, so we do not want to overwrite their values. We create a copy of each in the variables d1 and d2, where the d prefix is used to remind us that these variables contain dynamically updated values.

```
. generate d1=x1
. generate d2=x2
```

Because we are using previous values, we need to specify a value for dyngen to substitute in place of missings; in this case, we use the means.

```
. summarize d1 d2
```

| Variable | Obs | Mean | Std. dev. | Min | Max |
|---|---|---|---|---|---|
| d1 | 5 | 3.8 | 1.30384 | 2 | 5 |
| d2 | 5 | 1.8 | 1.30384 | 1 | 4 |

Within the dyngen command, we specify an update statement for d1 and d2. We also use observation subscripts to indicate the previous values as needed; see **[U] 13.7 Explicit subscripting**. With time-series data, we could also use time-series operators; see example 3 for an illustration.

```
. dyngen {
.     update d1 = .4*d1 + .1*d2[_n-1], missval(3.8)
.     update d2 = .2*d1[_n-1] + .3*d2[_n-1], missval(1.8)
. }
. list x1 x2 d*
```

| | x1 | x2 | d1 | d2 |
|---|---|---|---|---|
| 1. | 3 | 1 | 3.8 | 1.8 |
| 2. | 4 | 4 | 1.78 | 1.3 |
| 3. | 5 | 2 | 2.13 | .746 |
| 4. | 5 | 1 | 2.0746 | .6498 |
| 5. | 2 | 1 | .86498 | .60986 |

In observation 1, dyngen has substituted 3.8 for d1 and 1.8 for d2, values that would otherwise be missing because there are no data preceding the first observation. In observation 2, the updated value of d1 is $0.4 \times 4 + 0.1 \times 1.8 = 1.78$ and that of d2 is $0.2 \times 3.8 + 0.3 \times 1.8 = 1.3$, and so on.

## Example 2: Distinction between dyngen and replace

We can compare the results from example 1 with those from `replace` to see how `dyngen` operates differently.

As in example 1, we create two new variables, `r1` and `r2`, that will hold values we update using `replace`. There is no automatic way to handle missing values with `replace`, so we need to set the first values to the means "by hand" to avoid missing values later. We then have a `replace` command for each variable, restricted to observations 2 through 5.

```
. generate r1=x1
. generate r2=x2
. replace r1 = 3.8 in 1
(1 real change made)
. replace r2 = 1.8 in 1
(1 real change made)
. replace r1 = .4*r1 + .1*r2[_n-1] in 2/5
(4 real changes made)
. replace r2 = .2*r1[_n-1] + .3*r2[_n-1] in 2/5
(4 real changes made)
```

Now, we can compare the results side by side.

```
. list x* d* r*
```

| | x1 | x2 | d1 | d2 | r1 | r2 |
|---|---|---|---|---|---|---|
| 1. | 3 | 1 | 3.8 | 1.8 | 3.8 | 1.8 |
| 2. | 4 | 4 | 1.78 | 1.3 | 1.78 | 1.3 |
| 3. | 5 | 2 | 2.13 | .746 | 2.4 | .746 |
| 4. | 5 | 1 | 2.0746 | .6498 | 2.2 | .7038 |
| 5. | 2 | 1 | .86498 | .60986 | .9 | .65114 |

For the first two observations, the inputs are exactly the same, so there is no difference in the outcome. We see differences starting in the third row.

At the time that `replace` is updating the value of `r1` in observation 3, it is making the calculation

$$0.4 \times 5 + 0.1 \times 4 = 2.4$$

because the value of `r2` is still 4, the original value of `x2`. Compare this with the results of `dyngen`, which uses

$$0.4 \times 5 + 0.1 \times 1.3 = 2.13$$

That is, the key distinction is `dyngen` has fully updated observation 2 before moving on to observation 3. `replace` will make a full pass through `r1` before moving on to `r2`.

◁

### ▷ Example 3: Processing if conditions

Each update statement within the dyngen command can take an if condition. To illustrate, we replace d1 and d2 with the original values of x1 and x2 and update them again, this time restricting the updated observations to just those observations where time $\geq 3$.

```
. replace d1=x1
(5 real changes made)
. replace d2=x2
(5 real changes made)
```

Here, we tsset the data and use the lag operator instead of subscripting observations, but that is not required.

```
. tsset time
Time variable: time, 1 to 5
        Delta: 1 unit
. dyngen {
.     update d1 = .4*d1 + .1*L.d2 if time>=3
.     update d2 = .2*L.d1 + .3*L.d2 if time>=3
. }
. list x* d*
```

| | x1 | x2 | d1 | d2 |
|---|---|---|---|---|
| 1. | 3 | 1 | 3 | 1 |
| 2. | 4 | 4 | 4 | 4 |
| 3. | 5 | 2 | 2.4 | 2 |
| 4. | 5 | 1 | 2.2 | 1.08 |
| 5. | 2 | 1 | .908 | .764 |

When the same if condition is specified on all update statements, the results are equivalent to specifying one if condition on the entire dyngen block. We used the same if statement on both update statements above, so typing the following produces the same results as the code above.

```
dyngen {
    update d1 = .4*d1 + .1*L.d2
    update d2 = .2*L.d1 + .3*L.d2
} if time>=3
```

You may also specify an in qualifier with the dyngen command. If you specify an if or in qualifier, dyngen loops over the observations that meet the if condition or in range but will reference values outside that range if needed.

◁

## Also see

[D] **frunalias** — Change storage type of alias variables

[D] **generate** — Create or change contents of variable

**[U] 12 Data**

**[U] 13 Functions and expressions**

For suggested citations, see the FAQ on citing Stata documentation.