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

### 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, . . ., N, must already exist in the dataset.

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 \( x_1 \) and \( x_2 \) such that \( x_1 \) depends on its current value and the previous value of \( x_2 \), and \( x_2 \) depends on previous values of \( x_1 \) and \( x_2 \). We will be using these same values of \( x_1 \) and \( x_2 \) in subsequent examples, so we do not want to overwrite their values. We create a copy of each in the variables \( d_1 \) and \( d_2 \), 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
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td></td>
<td>5</td>
<td>3.8</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>d2</td>
<td></td>
<td>5</td>
<td>1.8</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Within the `dyngen` command, we specify an `update` statement for \( d_1 \) and \( d_2 \). 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*
```

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>d1</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3.8</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1.78</td>
<td>1.3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2.13</td>
<td>.746</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2.0746</td>
<td>.6498</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>.86498</td>
<td>.60986</td>
</tr>
</tbody>
</table>

In observation 1, `dyngen` has substituted 3.8 for \( d_1 \) and 1.8 for \( d_2 \), values that would otherwise be missing because there are no data preceding the first observation. In observation 2, the updated value of \( d_1 \) is \( 0.4 \times 4 + 0.1 \times 1.8 = 1.78 \) and that of \( d_2 \) 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.

```stata
. 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.

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

<table>
<thead>
<tr>
<th></th>
<th>x1</th>
<th>x2</th>
<th>d1</th>
<th>d2</th>
<th>r1</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3.8</td>
<td>1.8</td>
<td>3.8</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1.78</td>
<td>1.3</td>
<td>1.78</td>
<td>1.3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>2.13</td>
<td>.746</td>
<td>2.4</td>
<td>.746</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2.0746</td>
<td>.6498</td>
<td>2.2</td>
<td>.7038</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>.86498</td>
<td>.60986</td>
<td>.9</td>
<td>.65114</td>
</tr>
</tbody>
</table>
```

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 ≥ 3.

```stata
. replace d1=x1
   (5 real changes made)
. replace d2=x2
   (5 real changes made)
```
4 dyngen — Dynamically generate new values of variables

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

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3.</td>
<td>5</td>
<td>2</td>
<td>2.4</td>
</tr>
<tr>
<td>4.</td>
<td>5</td>
<td>1</td>
<td>2.2</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
<td>1</td>
<td>.908</td>
</tr>
</tbody>
</table>

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.

```stata
dyn gen {
    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] generate — Create or change contents of variable
[U] 12 Data
[U] 13 Functions and expressions