Stata Resources, Unit 4, Section 5:
Programs versus do files: What's the difference?

Friday, March 28, 2014 11:16 AM

By Meltem Odabaş

The first three sections of Stata Resources Unit 4 have already introduced loops: Loops can ease one's work through using the same code sequence with different variables. Here is one simple example:

```
gen NYdum=0
forvalues i=1(1)4 {
    replace NYdum=1 if state`i'=="NY"
}
tab NYdum
```

The example above uses dynamics of collective action dataset. For each collective action, there are four state variables, representing in which states the action took place. All four variables are string variables including two letter characters of accompanying state. In this example a dummy variable equal to zero is created for the state NY, and then replaced with one if at least one of the variables of state1, state2, state3 and state4 has the value "NY".

When we tabulate NYdum, we come up with the following result:

```
. tab NYdum
```

NYdum	Freq.	Percent	Cum.
0	14,978	63.42	63.42
1	8,638	36.58	100.00
Total	23,616	100.00	

For sure, one can use the following command to create "NYdum" dummy variable:

```
gen NYdum=0
replace NYdum=1 if state1=="NY" | state2=="NY" | state3=="NY" | state4=="NY"
```

However, as the number of elements/variables/values increase, the process gets more complicated, and necessitates long codes to be hand-written, which increases the probability of having typos.

As mentioned above, loops can ease one's work through using the same code sequence with

different variables. Program also has the same feature. Through creating a program, one may define a new command which captures a sequence of different commands, and use this program for different variables. The main difference between a loop and a program, however, is the following: In a loop, one needs to define the elements through which the coding sequence defined is expected to go over. A program is more flexible in that regard. Elements or values to be used in the program are entered in after the program is written. Therefore, the same program might be used over and over again with different elements and values with no necessary changes in the coding sequence.

The following simple example, also provided in this website in section 18.4, is useful in understanding the underlying logic of programs.

```
program list_args
        display "The whole argument you typed is: `0'"
        display "The 1st argument you typed is: `1'"
        display "The 2nd argument you typed is: `2'"
        display "The 3rd argument you typed is: `3'"
        display "The 4th argument you typed is: `4'"
end
```

The command program tells Stata to start define a new command, which is "list_args" in this example.
The following commands stated in the code represents what the program is expected to run in sequence.
The command "end" tells Stata to stop learning this new program called "list_args", and to end defining it.

Since the program is already defined, we can use it now. The results will be as in the following:

```
. list_args what do you do?
The whole argument you typed is: what do you do?
The 1st argument you typed is: what
The 2nd argument you typed is: do
The 3rd argument you typed is: you
The 4th argument you typed is: do?
```

As in loops, the entries in between the characters ` and ' represent the elements/values to be used in the program. In the example above, and in a program in general, `0' represents what is exactly typed by the user after the command "list_args". `1' represents the first, `2' represents the second, `3' represents the third and `4' represents the fourth argument. So, in other words, the arguments `1', 2',… break out the entry on the basis of blanks.

Let's define a new but similar program, "list_args2":

```
capture program drop list_args
program list_args2
        display "The 1st argument you typed is: `1'"
        display "The 2nd argument you typed is: `2'"
        display "The 3rd argument you typed is: `3'"
        display "The 4th argument you typed is: `4'"
end
```

The only difference is that this program does not include `0'. It we do not type anything, we will see that Stata will not give any error and run the program, but the arguments returned will be basically null arguments:

```
. list_args2
The 1st argument you typed is:
The 2nd argument you typed is:
The 3rd argument you typed is:
The 4th argument you typed is:
```

The following captions are some other trials with the similar sentence structure. As seen in these examples, one may also force the command to not to break out the arguments in basis of blanks by simply using quotation marks, if necessary:

```
. list_args2 what do you do?
The 1st argument you typed is: what
The 2nd argument you typed is: do
The 3rd argument you typed is: you
The 4th argument you typed is: do?

.
. list_args2 what "do you" do today?
The 1st argument you typed is: what
The 2nd argument you typed is: do you
The 3rd argument you typed is: do
The 4th argument you typed is: today?

.
. list_args2 "what do you" do today ?
The 1st argument you typed is: what do you
The 2nd argument you typed is: do
The 3rd argument you typed is: today
The 4th argument you typed is: ?
```

The user might also have loops in a program. Returning to the first example on loops in this module, we will write a program that enable us to create different dummies similar to "NYdum" at once with different claim codes. We will enter those claim codes after we have defined our program "states". Let's remind ourselves the initial loop example:

```
gen NYdum=0
forvalues i=1(1)4 {
    replace NYdum=1 if state`i'=="NY"
}
tab NYdum
```

As mentioned, a dummy variable equal to zero is created, and then replaced with one if at least one of the variables of state1, state2, state3 and state4 has the value "NY". Now, let's see our "states" program below:

```
capture program drop states
program states
    foreach var of local 0 {
        gen okay_state`var'=0
        forvalues i=1(1)4 {
            replace okay_state`var'=1 if state`i'=="`var'"

            }
        }
end

states AZ NY VA
```

The first line of the program begins to define a loop. "Local 0" represents the elements to be written after the program is defined, which are AZ, NY and VA in this example. Therefore, the "states" program gives the flexibility for entering the elements of the loop later on. Since we have more than one claim in this new example as compared to the NYdum example, we need to generate different dummy variables for each state, as given in the second line of the program. The rest of the lines (i.e., 3rd and 4th) of the program is just similar to NYdum example except the use of `var' rather than "Nydum" and "NY".

We can also show that "states" program is a generalization of the NYdum example by simply running the program for the value of "NY" and tabulate the generated dummy variable "okay_stateNY" with initially created "Nydum" dummy variable:

```
. states NY
(8613 real changes made)
(21 real changes made)
(4 real changes made)
(0 real changes made)

. tab NYdum okay_stateNY
```

NYdum	okay_stateNY 0	1	Total
0	14,978	0	14,978
1	0	8,638	8,638
Total	14,978	8,638	23,616