Introduction to Stata- A. chevalier

Lecture 4: Programming

Content of Lecture 4:

-looping (while, foreach)
-branching (if/else)
-Example: the bootstrap
- saving results

A] Looping

First, make sure you have read and understood the use of macros (lecture 3).
While is one of the most useful programming command in stata. It allows you to execute (part of) your dofile, more than once. In fact, while executes a command as long as the initial condition is true.

There are two cases:

* you know how many replications of the loop are needed.

The while loop is going to have the following structure:

```
define a local that is going to be your loop increment
while condition on `local’ true {
            do something            /* codes are in terms of `local’
      */
            add an incremental step to the local
            close the loop }
```

```
1.local i=1
2.while `i' <=10 {
3.   display `i'
4.   local i=`i'+1
5.   }
```

line 1: i can be seen as a counter, there is nothing magical about fixing it to 1. In fact, in the following example it is fixed to 95.
Line2: this is your statement on how many times the commands within the while loop are going to be replicated. This statement basically says do until counter reaches a limit (here 10). Remember to open { at the end of this line.
Line 3, all commands between { and } will be executed each time you go through the while loop. Note that your code is in terms of `i’.
Line 4: you have to increase you counter by 1 unit, otherwise, your initial statement will be always true and your loop will be permanent. The increase in the counter does not have to be unitary. For example, if you are using bi-annual data, you may want to fix your step to 2.
Note the syntax of this line, you are defining the local counter (i) as the previous value of the local counter (`i’) plus a step.

Example. You want to download some datasets from the library and do the same analysis for each dataset. Download set95 and set96 from my webpage, put them on your current directory and pretend that these two files are large datasets.
You could load each dataset individually, do the analysis, then paste and copy your dofile and do the analysis on the next dataset, and so on…if your analysis is quite long, the number of dataset large, and you realise that you made a mistake in your original dofile, you may be pasting and coping a large number of times. Here is where loops are useful (among other examples).

```
local i=95
while `i'<=96 {
      use set`i'              <- code in terms of `i'
      reg prob gender         <- your analysis is much longer
      local i=`i'+1
      }
```

* you do not know how many replications of the loop are needed.

Remember what was described at the end of lecture 3. Stata allocates locals `1’, `2’, …to the arguments typed after a command.

We are going to use this characteristic here, by stating that we want the while loop to run until the local `1’ is empty. We also need to know about another programmer’s command mac shift (macro shifter)

Digression on mac shift

mac shift is a stata command, that shifts the content of local numeric macros to their precedent (to the left), so `2’ becomes `1’, etc…

remember the program tester from lecture 3, here we are just going to modify it in order to show how mac shift works

```
capture program drop tester2
program define tester2

mac shift                    <- this in the new line
display “The variable in local 1 is: `1’ “
display “The variable in local 2 is: `2’ “
display “The variable in local 3 is: `3’ “
display “The variable in local 4 is: `4’ “

end

tester1 age sex educ lny
tester2 age sex educ lny
```

so initially `1’ contained age, then after using mac shift once it contain sex, if we were to carry on using mac shift, `1’ will contain educ, then lny and then eventually be empty. This is the strategy we are going to use, to stop the loop.

When the number of replications is unknown, a loop has the following form:

```
while “ `1’ ” ~= “” {
                  do something , the codes are written in terms of
`1’
                  mac shift
                  }
```

Note the differences with the structure of the loop when the number of replications is known. This time, the contain of the loop is in terms of `1’ not `i'. We do not need a counter, instead, we have a mac shift statement. Otherwise, the structure is rather similar.

The first line of the loop is particularly tricky this time: `1’ is the argument, the double quote are needed because, at this stage, we want our loop to treat `1’ as a string, so that we can have the statement on whether this string is empty “” or not.
If you do not put the “`1’” and type `1’, then the statement, when `1’ is empty, will become:
while ~=””
which is meaningless and will cause stata to crash.

Even when `1’ is not empty, it will typically be a variable name, so having as in our example bellow while age ~=”” will also be meaningless, all we want at the moment is to know that the local `1’ is not empty, and the best way to check this is to compare the name associated with `1’ and the string empty (“”).

So when do I need to use this kind of loops

Example: write a program, that recodes missing values into -99 for a list of variables (when the number of variables in the list may vary and is therefore not known apriori).

Use the wage dataset (taken from my webpage)

```
capture program drop misto99
program define misto0

while “ `1′ “ ~= “” {      /* make sure there is no space
                          in “”  */
          recode `1′ .=-99
          mac shift
          }

end
```

misto99 age sex educ lny

* In stata 7, there is also a new and easier way to create loop: for each.

The syntax is the following

```
local varlist "educ-wage"
foreach var of local varlist {
      su `var'
      }
```

In the first line, I define a local containing all the variables that I want to loop over
Rather than your program being defined in terms of `1’ you have the choice of the name of the local in which your program is going to be defined, here I chose var.
Note, that varlist is a local but you do not need to put `varlist’ (this is an exception to the rule, and I find it confusing).
Then you white your program in terms of `var’
Your loop closes with }, no need to increase a counter or use mac shift.

B] branching if/else

If has a similar syntax to while, but it is easier to use. If does something or not depending on the validity of the initial statement.

Comment: It is important to distinguish between the if modifier that you all know and the branching if:

Regress lny age educ if sex==1 for each observation, stata assess whether sex=1, and if true, the observation is retained for the execution of the command.

If sex==1 { regress lny age edu } does not make a lot of sense but will or will not produce a result (in any case, this is not the one you want). This time, the condition sex==1 is assessed for the first observation only, if sex==1 in the first observation then regress lny age edu will be executed for all observations, if not, then all commands within the if {} will not be executed.

To summarise:

Conditional if: do something if condition true sum lny if sex==1 (check for each observation)
Programmer if: if (else) something true, then do something. (check only once)

If (else) executes the command(s) if the initial condition is true. The structure of branching is always: if (condition) { do something }.
For example you are using pooled CPS data, from 1990 to 1999. Unfortunately, after 92 the number of years of education was no longer reported in the CPS and it was replaced with the qualification attained. So here is an extract of your do file, were regression pre and post 92 will defer.

```
if `i’<=92     {
          local xlist “educ age age2”
          }
else      {
          local xlist “qualify age age2”
          }
local i=90
while `i’ <=99 {
     reg lny `xlist’ if year==`i'
     local i=`i'+1
}
```

DANGER:
Be careful when closing }. Stata ignores the rest of the line after the } are closed. So in the previous example, if we had typed

```
if `i’<=92     {
          local xlist “educ age age2”
          } else         {
          local xlist “qualify age age2”
          }
```

The second condition would not have been assessed, before executing the second part of the branching.

C] Example the bootstrap

Bootstapping is becoming increasingly popular, so it is probably worth having a look at this section.
Basically, what you do when you bootstrap some results, is that you take a random sample of your data, do whatever needs to be done to generate your results, store these results somewhere, and do this sequence a large number of times (say 1000). By the end, you have results from each individual replication, which allows you to calculate a standard error.

So basically what you want to do is:

-take a random sample of your data
- compute the results
- save the data in memory
-open the data set of results
-add new results
-save data set of results
-open original dataset

That will work but will be slooowww

There is a series of command, that will help you a lot in this task.

-postfile: opens a new stata.dta dataset without disturbing the original dataset
-post: stores another observation in the left open file, the values are specified as arguments
-postclose closes the file, saves it and returns the original dataset.

The bootstrap command takes care of postfile and postclose, as well as the random sampling, so all that is left to do is to compute and post the results.

The syntax of post is: post postname (results) (more than one result can be posted)
Where postname is the name of the file where the results are to be posted.

So a bootstapping program has the following form:

```
program define <progname>
if "`1'" =="?" {
      global S_1 "<result names>"
      exit
      }
      <perform calculations>
      post `1' (results)
end
```

```
bstrap progname, reps (xxx)
```

Bootstap is based on an old stata (pre release6) where internal macro (e(),r(), see lecture 3) were defined as S_
Line 3 uses this old convention, all you need to know is that you have to provide names for all the results you want to bootstrap (all names will have to be separated by a space).

Here is an example: Using the auto dataset, we want to get the standard error of the median of mpg.

Fist, you have to remember that the median is available after the summarize, detail command.

```
su mpg,detail
```

We know that the results of the median is stored in r(p50) after a summarize, detail command (see lecture 3). Hence we can write our bootstrap program:

```
capture program drop median
program define median
if "`1'" =="?" {
      global S_1 "medmpg"
      exit
      }
      summarize mpg,detail
      post `1' (r(p50))
end
```

To speed things up, we are going to restrict ourselves to 100 replications:

```
bstrap median, reps (100)
```

Now if we are to run that bootstrap again, we get slightly different results, this is because, there is no reason to select the same random samples. If you want your results to be replicable, you can set a seed, so that the same random samples will always be extracted.

```
set seed 123456
bstrap median, reps (100)
```

Also you may want to keep the file of results,

```
bstrap median, reps (100) saving (“bsmedian”)
```

the file bsmedian has now been saved on my current directory

Warning: Be aware of missing variables, as they are going to be selected in your random samples, and will provide you with bias results. So before bootstrap, we should have included a keep if mpg~=.
To speed things up, it is also useful to restrict the sample to the variables that are needed for the bootstap.

Comment: Stata has to open and close two files for each replication, the smaller the files the quicker this operation is.

D] miscellaneous

* Using quietly.

Do files can generate a large volume of output. It is often the case that most of it, is junk and you are only interested in a few results. You can use the quietly command to reduce the junk.
If you do not want a command to display any output, just add quietly at the beginning.

```
local xvar "weight gratio foreign"

quietly reg mpg `xvar'
local cst= _b[_cons]
di "the constant is: `cst'"
```

you can also group statement together between {} and put a quietly in front of it.

* version control

Stata changes regularly, so programs that you wrote in the past may no longer be working. Rather than having to rewrite them to comply with the new syntax, it is possible to let stata do the conversion work. In fact, previous stata syntax are always available and understood by stata, but the default is set to work with the current syntax. You can change this default by using the version control.

Say that you wrote a do file a while ago on stata5. Now that we have updated to version7, part of your program does not work. The only thing you have to do is to add the following line in your do file:

```
program define progname
        version 5.0
        …..
```

For ado files, it is a good practice to always use version control, even if you are using the latest release of stata, to ensure that your adofile will still be up to the job in a few years time. It is also good practice to date and comment on the first line of your program using * . Alternatively you can marked them with *!. This has the advantage that if you use the which command, your comments will appear.

```
which progname
D:\ado\personal\progname.ado
*! AC – 29 Sept 2001
*! AC – 1 Oct 2001, bug on display fixed
```

E] saving results

If you want your programs to returns some results, as for example the summarize program does, you need to save these results. Saving results in not difficult:

1- add the rclass option to the program define statement
2- add a return statement in the body of your program

Scalar, macros and matrices can be returned in results statement.

Example:

```
capture program drop example
program define example, rclass
     Return scalar x =1
end
```

Now if you try it:

```
example
return list
scalars:
     r(x)           = 1
```

Let also see what happens when we omit either the rclass statement or the return line.

Now compare these two programs:

```
program define example1
      quietly summarize `1'           program define example2, rclass
end                                         quietly summarize `1'
                                      end


. example1 mpg                        . example2 mpg

. return list                         . return list

scalars:
                 r(N) =  74
             r(sum_w) =  74
              r(mean) =  21.297
               r(Var) =  33.472
                r(sd) =  5.785
               r(min) =  12
               r(max) =  41
               r(sum) =  1576
```

By specifying rclass, stata distinguishes between the returns that are provided by stata command and those that come from your program. If your program has been defined as r-class, stata only returns your saved results when the command return list is displayed. If you want stata to return results from another stata command as well as yours, you need to include return add to the body of your program.

```
program define example3, rclass
      quietly summarize `1'
      return add
end
```

By specifying add, you can return your own and stata results.

A returned scalar is thereafter referred as: r(<name>)
A returned macro is `r(<name>)’
A return matrix is r(<name>)

Stata also saves results in s and e, so you can make your program S-class or E-class. You cannot combined classes, so a program is either R, S or E class or nothing is returned.
S class programs return only macros, they are used for advanced and non-standard parsing, so we should not review them here.
E-class is the estimation class, so you will be using it, if you write a new estimate
By convention, some results are always saved in the same E commands.
The name of the estimation command is stored in e(cmd), estimates and the matrix of variance-covariance are in e(b) and e(V), the sample used is marked in e(sample) were observations used are marked with 1 and others with 0. If you write your own estimates, following these conventions will allow you to use stata commands to execute post estimation commands.
The main difference between E and R results, is that the results are posted using estimate rather than result.