Introduction to Stata- A. Chevalier

Lecture 3: The basic of programming- do file and macro

Content of Lecture 3:

-Entering and executing programs
    do file
    program
    ado file
-macros

A] Entering and executing programs

So far we have only worked with stata interactively, that is we were executing commands one at the time, getting results before moving to the next one. There is nothing wrong with this approach, but it is not convenient to keep track of what you have been doing, even if you have opened a log file. Also if modifications are needed, you will have to start from scratch and try to remember how you got so far in the first place.
There is a simple way to work with stata that allows you to keep track of your work, speeds things up and allows you to make changes easily. This involves programming. But this is not hardcore C++ stuff, programming for the moment will only mean put together stata commands that you know.

First, to make sure you are not running out of the room, let's write the simplest program. You can use any text editor to do that; notepad and LATEX are probably the most common. There is also an inbuilt editor in stata, which we are going to use later.

* First method: The do file

Do files are useful as a way to keep track and edit your work

Say you are using notepad
Type:

Display "Hello, world"

Save it as hello.do in your stata working directory

All stata do-files have a do extension and nothing else will do.
Now going back to stata type:

```
. do hello                          <- you typed this

. display "hello world"             <- Stata typed that
hello world

.
end of do-file
.                                   <- stata awaits next
                                    command
```

This seems simple enough but it may not have worked for everybody; what might have gone wrong:
You get a "file hello.do not found" reply
-either hello.do was not saved in your current directory
- you did not save the file with the right do extension

hello.do is found, but the results are not what you expected.

Hello.do was not saved as an ASCII file, you did not specify ASCII in your wordprocessor (this is the reason why it is more convenient to use notepad rather than word)

Nothing happens

You forgot to press the return key at the end of your do file.
To avoid that, you can finish all your do files with exit, which will make sure that the last line typed is executed.

*Second method: Interactive –program define- command

program define allows you to create a stata command, it is therefore useful when you have a set of commands that are going to be used repetitively. Rather than typing the all set again, just type the program name and all the commands are executed. A program is defined by a name that cannot be longer than 8 characters. The program starts with a program define name statement and ends with an "end" statement.

```
. program define hello              <- you type
  1. display "hello, world"         <- stata display 1,
                                    and you type …
  2. end                            <- program end

. hello                             <- you type
hello, world                        <- stata display
```

You have build your first stata command. You will never want to define a program interactively, but it is easier to demonstrate this way.
So what's the difference with the do file:

You invoke your build stata command like any other stata command
Your stata command needs to be defined
The core of the program finishes with end

Problems with program define:

1) redefinition
you may want to change your program and add/alter a line

```
. program define hello
hello already defined
r(110);
```

Stata remembers programs defined throughout the session, to redefine a program you need to drop the previous version.

```
. program drop hello

. program define hello
  1. display "I'll be back"
```

```
  2. end

. hello
I'll be back
```

2) reserved name

you cannot define your command as an official stata command, in fact you can but it executes stata official command rather than yours. To find out which name you can/cannot use.

```
. which des
built-in command:  describe

. which hello
command hello not found as either built-in or ado-file
r(111);
```

3) debugging

When debugging large and complicated program, you may want some help relative to which line causes problem.

Do to so, there is a trace command, that shows all program lines as the command is executed.

So after your program has crashed;

```
. set trace on           /* turn on trace */

. set more off           /* screen keeps on scrolling */

. log using junk,replace      /* open log file */

invoke your program

. log close              /* close log file */

. set more on            /* more on */

. set trace off          /* trace off */
```

So what is that about:

All stata display stops after reaching the bottom of the page and ask for more until you press a key. As setting the trace on generates a large output, you don't want to be pressing a key constantly, so you have the option to stop this more.

You also open a log file, so that you can have a look at the output you generate and which line generated an error message.

After debugging, close your log file, and set the options back to their initial setup.

4) size matters

There is a limit of 3500 lines to a program. This is not a constraint because I have never seen a program even remotely close to this limit, second, programs can (and

should) call each other, so you should have a succession of short program rather than a massive one (it also makes things easier to debug).

5) programs that are defined this way cannot be saved, (I told you this has only explanation power, and programs are never defined interactively).

* Third method: program define in a do file

To overcome all these problems associated with program define, it is best to define your program within a do-file

In your editor, type;

```
capture drop program hello
program define hello

display “hello,again”

end                              /* end your program */

hello
exit                             /* end your do file */
```

Before defining the program, I made sure that program hello did not exit. I could have typed `drop program hello`, but this would have crashed if program hello did not exist in the first place. Capture is a really useful command, when placed at the beginning of a command line, it will execute the line if possible, otherwise it will go to the next one , but you should use it thoughtfully.

Then you save your do file and run it

```
. do hello

. capture program drop hello

.
. program define hello
  1.
. display "hello,again"
  2.
. end                                /* end your program */

.
. hello
hello,again

.
. exit

end of do-file
```

Your program scrolls down and is executed.
An alternative to `do hello` is to `run hello`, the only difference, is that your program does not scroll down before being executed.
You can have programs calling each other, and also do files calling each other.

* the ado file
If the set of commands that you are typing are going to be used regularly, not only during this session but over time, it may be a good idea to save them as an ado file. Ado stands for automatically loaded do file. An ado file is a stata command that you create yourself; they work pretty much the same way as do files.
Going back to you editor, delete the line hello (asking for the program hello to be executed) and save the program hello.do as hello.ado.

In stata,
```
. program drop hello          <-just to show you, hello is not loaded
. hello
hello world
```

Like any other program, ado files stay in memory unless overwritten, but this time, stata does not let you know that you are not updating, when you thought you were. So after changing an ado file, type `discard` at your stata prompt before running your ado file.

If an adofile is going to be used regularly for different projects, it is best to save it on your personal ado space; C:\ado\personal (see lecture 1)

* There is another type of file that you may have to use from time to time. (Dictionary)
Datasets don't always come up in stata format, nor in a format that is easy for stata to read with the infile command (see lecture 1). They may come in a really complex format where each observation in on one line but all variables are collated. In such a case you need to write a dictionary, basically describing where the variable splits are.

AMC Concord	4099	2232.5112930
AMC Pacer	4749	1733.0113350
AMC Spirit	3799	223.0122640
Buick Century	4816	2034.5163250
Buick Electra	7827	1544.0204080
Buick LeSabre	5788	1834.0213670
Buick Opel	4453	263.0102230
Buick Regal	5189	2032.0163280
Buick Riviera	10372	1633.5173880
Buick Skylark	4082	1933.5133400
Cad. Deville	11385	1434.0204330
Cad. Eldorado	14500	1423.5163900

Dictionaries are do files with a different extension, where you describe the data. Look for help

In the above example, we will write something like:

```
Dictionary using cars.raw {
_column(1) str19 mandm %19s      "make and model"
_column (20) int price           "Price"
_column(28) int mpg              "mpg"
….
```

Working with do files makes your work easier to replicate. So organise your do file in a way that makes things easier for you to go back to your analysis, this is a matter of personal taste. You can add comments in your dofile to help yourself. Any line starting with a * is not executed nor are all the lines between /* and */.

At first, you may think that do files, just prevent you from doing simple calculations, for example, normalising a variable, making some prediction after a regression, where in the interactive mode, you will just have enter the value yourself.

Normalise a variable:
In interactive mode:

```
. su mpg

    Variable |     Obs        Mean   Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         mpg |      74     21.2973   5.785503         12         41

. gen nmpg=mpg-21.2973
```

In general, it is easier to do:

```
. su mpg

    Variable |     Obs        Mean   Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         mpg |      74     21.2973   5.785503         12         41

. gen nmpg2=mpg-r(mean)
```

So that if your sample changes, you do not have to change your do file.

I can check that the two methods gave the same solution:

```
. su nmpg nmpg2

    Variable |     Obs        Mean   Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        nmpg |      74   -2.73e-06   5.785503    -9.2973    19.7027
       nmpg2 |      74   -4.03e-08   5.785503  -9.297297    19.7027
```

There is nothing magical to this r(mean). Most of stata commands leave behind them a trail of information that can be used easily. This information can take the form of strings, numbers or matrices and is a great tool for programming but also for constructing output tables.

To find out what exists after each command producing an output you can type:
`return list`
After each estimation, there is also stack of information in stata memory; try:
`estimates list`

This information disappears from stata memory as soon as you type another command, so if you want to use them in a more permanent manner, just extract it and rename it.

```
. su mpg

    Variable |     Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         mpg |      74     21.2973    5.785503         12         41

. return list

scalars:
                  r(N) =  74
              r(sum_w) =  74
               r(mean) =  21.2972972972973
                r(Var) =  33.47204738985561
                 r(sd) =  5.785503209735141
                r(min) =  12
                r(max) =  41
                r(sum) =  1576

. reg mpg weight gratio

      Source |       SS       df       MS              Number of obs =      74
-------------+------------------------------           F(  2,    71) =   66.38
       Model |  1592.05392     2  796.026958            Prob > F      =  0.0000
    Residual |  851.405544    71  11.9916274            R-squared     =  0.6516
-------------+------------------------------           Adj R-squared =  0.6417
       Total |  2443.45946    73  33.4720474            Root MSE      =  3.4629

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -.0059643   .0008013    -7.44   0.000    -.0075621   -.0043665
      gratio |   .0994882   1.364894     0.07   0.942    -2.622033     2.82101
       _cons |   39.00643   6.169955     6.32   0.000     26.70389    51.30897
------------------------------------------------------------------------------

. estimate list

scalars:
                  e(N) =  74
               e(df_m) =  2
               e(df_r) =  71
                  e(F) =  66.38189565945015
                 e(r2) =  .6515573275289056
               e(rmse) =  3.462892920968853
                e(mss) =  1592.05391533063
                e(rss) =  851.4055441288298
               e(r2_a) =  .6417420409804241
                 e(ll) =  -195.3859199192182
               e(ll_0) =  -234.3943376482347

macros:
            e(depvar) : "mpg"
               e(cmd) : "regress"
           e(predict) : "regres_p"
```

```
             e(model) : "ols"

matrices:
                 e(b) :  1 x 3
                 e(V) :  3 x 3

functions:
            e(sample)
```

B] macros

There are two types of macro, local and global, which in most cases have similar usage. For the moment I will concentrate on local macros, but all that is said here is valid for global macros.

Macros are shorthand for one thing standing for another.

For example: I usually put all my independent variables in a macro, so that I'm sure that my specification remains the same throughout the analysis, and if I want to change it, I need to change only 1 line in my do file and not all the regressions. Macros not only save time but reduce the chance of mistakes…
In a do file, I create a local xlist containing all my regressors, then type my regression: note the peculiar notation to call back your local macro `name' (leading left quote, trailing right quote). Most of the problems you will encountered when using macros have to do with this notation!!!!

```
. local xlist " weight length gratio foreign"

.
. reg mpg `xlist'

      Source |       SS       df       MS              Number of obs =      74
-------------+------------------------------           F(  4,    69) =   34.08
       Model |  1622.35839     4  405.589597            Prob > F      =  0.0000
    Residual |   821.10107    69  11.9000155            R-squared     =  0.6640
-------------+------------------------------           Adj R-squared =  0.6445
       Total |  2443.45946    73  33.4720474            Root MSE      =  3.4496

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -.0036863    .001777    -2.07   0.042    -.0072313   -.0001413
      length |  -.0839138   .0564141    -1.49   0.141    -.1964569    .0286293
      gratio |    .673715   1.468161     0.46   0.648    -2.255186    3.602616
     foreign |  -.6496837   .9394243    -0.69   0.492    -2.523784    1.224416
       _cons |   46.37753   8.161182     5.68   0.000     30.09641    62.65864
------------------------------------------------------------------------------
```

Comments: in fact a local macro can be defined as:

```
. local xlist " weight length gratio foreign"
or
. local xlist = " weight length gratio foreign"
```

The two are equivalent, but with the = sign, the statement made in the macro cannot be longer than 80 characters, without the = it is up to 18.632. Thus I prefer not putting an equal sign (rather than bang my head on the wall, wondering why half of my macro is not included…)

In fact a macro does not have to be defined to be called. If I do not defined the macro varlist, I can still use it in a stata statement, it is just empty.

```
. reg mpg `varlist'

      Source |       SS       df       MS              Number of obs =      74
-------------+------------------------------           F(  0,    73) =    0.00
       Model |        0.00     0           .            Prob > F      =       .
    Residual |  2443.45946    73  33.4720474            R-squared     =  0.0000
-------------+------------------------------           Adj R-squared =  0.0000
       Total |  2443.45946    73  33.4720474            Root MSE      =  5.7855
```

```
------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |    21.2973   .6725511    31.67   0.000      19.9569    22.63769
------------------------------------------------------------------------------
```

The local `varlist' is empty, hence an expression using `varlist' will not return an error message. This feature is desirable (in programming) but can sometime lead into problems of its own

Macros are not limited to containing a list of variables, in fact they can contain anything.

- list of variables
as seen above

- Statement:

```
local if "if (hours>=0 | hours ~=.) & week>0 and earn>0 & earn/weeks>100"

sum xxx `if'
tab xxx`if'
reg yyy xxx `if'
```

Once again this makes do files clearer and reduces the risk of mistakes

- Numbers

```
local i=1
```

note that here I put the = sign and do not put "".

- Scalars

Regression excluding the 5th and 95th percentile.

```
quietly su lny, detail
local lo=r(p5)
local hi=r(p95)
regress lny `xlist' if `lo'<lny & lny<`hi'
```

Macros can have the same name than a variable and stata will not be confused (although you may). Local and global macro may also have the same name (but this time you are really asking for trouble). All that is possible because of the specific notation of macros.
The rules for naming macros are slightly different than for variables:
-locals can only be 7 characters long.
-locals can be named numbers (this is mostly used in programming (see lecture 5) so unless you are absolutely sure what you are doing, don't name a local by a number without a reason).

In programs and do files, the macros are local or, if you prefer private. No other program or do files can change them or in fact know that they exist.

This is quite tricky, and can get your mind close to insanity for a while, but look at the following example extracted from a do file;

```
local gender “sex”

program define truc
     su `gender’ if crash==1
end

truc
```

This do file will work but not do what you expect because the local `gender’ is defined outside the program truc. Applying our rule, the program truc has no idea that the local gender has been defined, and within truc, `gender’ is empty.

We will see in lecture 5 how to parse information to programs, but in this simple case, we only need to define the local `gender’ within the program truc rather than outside.

```
capture program drop truc
program define truc
     local gender “sex”
     su `gender’ if crash==1
end

truc
```

Replicate this program using the auto dataset, replace crash by foreign and sex by mpg.

Then, what do you think of this do file:

```
capture program drop truc
program define truc
     local gender “sex”
     su `gender’ if crash==1
end

truc
su `gender’
```

This time stata has no idea what `gender’ is outside the truc program, so `gender’ would have to be defined twice inside and outside the program truc.
To avoid this kind of problem, stata also provide another type of macro, that are not private and remain accessible and changeable everywhere. This type of macro is called global

Globals are defined in a similar way as locals, but can be 8 characters long and cannot be a number. When calling back a global, you use $macroname rather than `localname'.

```
global gender "sex"

capture program drop truc
program define truc
          su $gender if crash==1
end

truc
su $gender
```

However, you should rarely have to use globals, as their contents may be changed by other programs without you noticing (this is because they are not private to a specific program).

```
global gender "sex"

capture program drop truc
program define truc
          su $gender if crash==1
end

truc
do truc2                  <- note new line
su $gender
```

It is possible that in truc2 I have a line saying global gender "length", which means that without studying truc2, I have no way of knowing whether su $gender is going to be executed as;
`su sex` or `su length`

* Digression on parsing

Whenever, you type some command, this information has to be translated into things that stata does understand, this is what parsing is about, and that tends to be quite technical.
As stated previously, do not define locals by a number unless you are confident in what you are doing. This is because Stata automatically defines the local `1', `2', … when it enters a program or a do file. All you need to know for the moment is that stata associates locals to variables typed by the user. So if you type:

list age sex educ lny /* I could have use any command, it is always true */
stata fills the local macros 1, 2, 3 and 4 with respectively age sex educ lny
/* remember, local macros can be called by numbers */

If you type:

do myfile alpha

Macro `1' will contain alpha and `2' and all the remaining macros will remain empty

If now you type
do myfile beta alpha

Macro `1' will contain beta and `2' contains alpha

In this way you can pass argument to your do file and programs. We can check that with the following program:

```
capture program drop tester
program define tester
     display "local 1 contains :`1' "
     display "local 2 contains :`2' "
     display "local 3 contains :`3' "
     display "local 4 contains :`4' "
end
```

tester alpha
tester beta alpha

stata recognises that arguments are separated by space or within " "
Try:

```
tester abc?@*&£ 2+2
tester "this is a stupidly long argument"
```

We will see more comprehensive ways of parsing information to a stata program in Lecture 5.

Occasionally, you may experience the following problem. Your macro name follows a \. This confuses stata and the ` is not read, leading a to an error message. The solution is to enter \\ or `/', see below:

```
use hdno persno using "F:\LICENSE DATA\L.F.S\child
vars\\`r'`y'q`q'dpchvars.dta"
or
use hdno persno using "F:\LICENSE DATA\L.F.S\child
vars`\'`r'`y'q`q'dpchvars.dta"
```

What the difference between macro and scalar?

Scalars are used to store numbers. So you can type `scalar x= 3 display x` or `local x =3 display `x'` and get the same answer. The difference comes from the precision, macro store numbers with an accuracy of 12 digits, while scalar uses 16 digits.
In most cases, locals are fine, but when you have statement like `x'=`y' approximation errors may be playing trick with you and you may adopt scalars.