Plan A, ver 2.0

- Class 1 - Introduction
  - Graphical user interface
  - Datasets, command syntax and do-files
- Class 2 - Programming basics
  - Macros and post-estimation variables
  - Loops and control flow
- Class 3 - Posting results and two main examples
  - Posting results to a table
  - Creating a summary statistics table
  - Monte Carlo simulation

What is a Macro?

A Macro is a string (= a sequence of characters) that we name and can refer to.

One type of a macro is the local macro (local = can not be referred to outside the program):

```
// Define local
local <macro name> = <expression>

// Refer to local
[...] ‘<macro name>’
```

Note the back-quote and quote signs: ‘ is the character usually on the upper left corner of the main part of your keyboard (where ~ is). ’ is the usual single-quote sign you’re using.

What is a Macro?

```
// Define local
local number = 17

// Refer to the define local
di ‘number’ * 4
list age age_sq in 1/‘number’
```

A small nuance: the local **number** does *not* contain the *number* 17. It contains the *two characters* 17:

```
local number = -4

// Square it, Take 1
di ‘number’ ^ 2
-16
// Square it, Take 2
di (‘number’) ^ 2
16
```

What is happening here?

Stata goes through each command, looking for backquotes and quotes.
When it finds `‘something’` it looks for a local named `something` and replaces it with the expression we put in it.
Only then, after all backquotes and quotes are gone (replaced by the expressions) will Stata run the command, as if there were no locals involved.

```
di ‘number’ ^ 2
```

Remember, ^ has precedence over the minus sign, so first (4 ^ 2) is calculated and then the negation is made

Strings and Macros

In Stata, we put string expressions between double-quotes. For example:

```
gen     girl = 1 if sex == "female"
replace girl = 0 if sex == "male"
```

If we don't put double quotes in a string expression, Stata will look for a variable with that name. We need to tell Stata the word is a value instead of part of the command. This is why we need the double quotes.

Same goes for macros:

```
local greeting = "Hello world!"
di "‘greeting’"      // will be run as: di "Hello world!"

local mycommand = "tab"
‘mycommand’ age female   // will be run as: tab age female
```

Strings and Macros - Something Stupid

String values in Stata usually do not exceed 255 characters.

If you want to put a longer string into a macro, you will need to drop the = sign from the macro assignment:

```
local justCantShutUp "a really really really really [...] long string"
```

Otherwise, if you'll insist on the = sign, your string will be heartlessly cut after 255 characters without special notice.

Nested Macros

What's nice about Macros is that you can nest them: Put one macro's name as another macro's value:

```
local greeting = "Hello world!"
local macro2use = "greeting"
di "`macro2use'"    // will go like this: (1) di "`macro2use'"
                    //                    (2) di "`greeting'"
                    //                    (3) di "Hello World!"
                    // and only then the command will be run


local command_east = "reg"
local command_west = "xtreg"
local side = "east"
`command_`side'' income educ
// will do: (1) `command_`side'' income educ
//          (2) `command_east' income educ
//          (3) reg income educ
// and only then the command will be run
```

For now it looks stupid and unnecessary, but when we'll start doing loops you'll be king

Globals

In most cases you will not need them at all. However, some people are weird enough to insist on using them. You might end up working with one of those weird people, so at least know globals when you see them.

```
// Define the global and assign an expression to it
global <macro name> = <expression>

// Refer to the global
[...] ${<macro name>}
```

Example:

```
global a = 4

// Refer to the global
di "Four is ${a}"   // will print: Four is 4
```

First loop: `forvalues`

Loops are lines of code that can be run more than one time. Each time is called an **iteration** and in `for` loops there is also an index that is changing each iteration (the index is actually a local).

In the case of `forvalues`, the index is incremented each iteration:

```
// Define the loop
forvalues <index name> = <starting value>/<ending value> {
   // Commands to run each iteration
   // ... more commands ...
}
```

The loop will put <starting value> into <index name>, then run the commands until it reaches the closing }. Then it will go back, increase the value of <index name> by one and run the commands again, until it is done with the commands for the <ending value>.

First loop: `forvalues`

For example:

```
forvalues i = 1/3 {
   di "Iteration #‘i’"
}
```

Will print:

```
Iteration #1
Iteration #2
Iteration #3
```

```
forvalues i = 7(7)21 {
   replace age = 0 in ‘i’
}
```

Will set the value of the variable age to 0 for observations 7, 14 and 21.

What is it good for? Part 1

Imagine you have three different specifications:

$$y_t = \beta_0 + \beta_1 x_t + \varepsilon_t$$

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 age_t + \beta_3 agesq_t + \beta_4 educ_t + \eta_t$$

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 age_t + \beta_3 agesq_t + \beta_4 educ_t + \beta_5 mo\_educ_t + \beta_6 fa\_educ_t + u_t$$

```
local spec1 ""
local spec2 "age agesq educ"
local spec3 "`spec2' mo_educ fa_educ"

forvalues i = 1/3 {
   reg y x `spec`i''
}
```

Still not convinced? You're right. This example, as it is now, is longer than just writing three lines of regressions. But hold on...

foreach

When you want to iterate on other lists - not just an arithmetic sequence of numbers - you will want to use `foreach`.

The simplest form to use `foreach` is:

```
foreach <index name> in <list separated by space> {
   // Commands to run each iteration
}
```

For example:

```
foreach i in 3 15 17 39 {
   di "I am number `i'"
}

foreach dep_var in income consumption health_score {
   reg `dep_var' educ age agesq
}
```

Even though we didn't put double-quotes on the values, since they are inside a foreach loop with the `in` word, Stata knows to treat them as values

foreach

If you want to loop over values that have space in them, use the double-quotes:

```
foreach fullname in "Roy Mill" "John Doe" Elvis Presley Madonna {
   di "Hello ‘fullname’"
}
```

Will print:

```
Hello Roy Mill
Hello John Doe
Hello Elvis
Hello Presley
Hello Madonna
```

`foreach` and variables lists

When you iterate over variables' names it's better to put `of varlist` instead of `in`:

```
foreach <index name> of varlist <varlist> {
   // Commands to run each iteration
}
```

This way:

- Stata will check that each element of the variables list is actually a variable (avoid typos)
- You will be able to use wildcards

```
foreach mother_vars of varlist mother_* {
   // This loop will go over all variables that begin with mother_
}
foreach setvar of varlist set?_score {
  // This loop will go over all variables that have one character
  // where the ? is. For example set1_score, set2_score, ...
  // (but not set14_score)
}
```

What is it good for? Part 2

Remember our three specifications? Now imagine we want to run them on three different samples: males, females and both. Here is one way to do that:

```
local spec1 ""
local spec2 "age agesq educ"
local spec3 "‘spec2’ mo_educ fa_educ"

foreach sampleCond in "if male == 1" "if male == 0" "" {
  forvalues i = 1/3 {
   reg y x ‘spec‘i’’ ‘sampleCond’
  }
}
```

What is it good for? Part 2

This loop is equivalent to running:

```
reg y x if male == 1
reg y x age agesq educ if male == 1
reg y x age agesq educ mo_educ fa_educ if male == 1
reg y x if male == 0
reg y x age agesq educ if male == 0
reg y x age agesq educ mo_educ fa_educ if male == 0
reg y x
reg y x age agesq educ
reg y x age agesq educ mo_educ fa_educ
```

Now imagine you want to change the standard errors to robust, or add another control variable to the second specification. How much work will you need for the loops version and how much for the this version? And wait until you will need to post the results to a table.

Scalars

Another place you can keep values in is a scalar. Unlike the macro, a scalar is actually holds the value and its name can be referred to like a variable's name:

```
scalar <scalar name> = <expression>

// Referring to the scalar:
di <scalar name>
```

The expression can be either a number or a string.

Referring to the scalar is done just by writing the name. No ‘’ or `${}` needed. Just like referring to a variable in the dataset.

But be careful with naming scalars and variables with the same name. It won't create an error, but it can cause Stata to go crazy.

Matrices

There are matrices in the regular Stata environement and matrices in the Mata environment. We will discuss the former. Here is the simplest way to define and refer to a matrix:

```
// Defining a matrix
matrix <matrix name> = <matrix expression>

// Printing a matrix
matrix list <matrix name>

// Referring to a matrix's cell:
di <matrix name>[<row>,<column>]
```

"matrix expression" means either another matrix, or a function that returns a matrix or a typed-in matrix like this:

```
// Defining a 3X3 matrix
matrix A = (3 , 2 , 1 \ 2 , 3 , 5 \ 6 , 3 , 1)
```

Backslashes (\) represent end of row.
Inside every row elements are separated by commas.

Matrices

You can perform matrix algebra operations:

```
// Define matrices for OLS
matrix X = (2 , 3 \ 2 , 1 \ 1, 1)
matrix Y = (1 \ 0 \ -1)

// Calculate OLS coefficients:
matrix betas = inv(X'*X)*X'*Y
```

You can also turn parts of the dataset to a matrix and put matrices back on the dataset in memory (but you are limited by `matsize`):

```
// From the dataset to a matrix:
mkmat age educ if city == "Palo Alto", matrix(age_educ)

drop _all     // Drops all variables

// From a matrix to the dataset:
svmat betas
```

Matrices

A few notes:

- Matrix can get only numeric values. No strings.
- As always, this is just the tip of the iceberg. More on matrices can be learned by typing `help matrix` in the command line. Special sections worth of note:
  - matrix operators
  - matrix functions
  - matrix subscripting
  - matrix rownames
  - mkmat

Getting values returned by commands

Most commands you run will not only print output to the results window, but also return scalars, macros and matrices to the memory.

There are three types of programs:

1. Programs that return values to r(): r-class programs.
2. Programs that return values to e(): e-class programs.
3. Other programs that we don't care about right now.

Please, do NOT ask why. It's not important. All you need to know is to check whether `return list` or `ereturn list` gives you what you want.

Getting values returned by commands

```
su union
```

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       union |     19238    .2344319    .4236542          0          1
```

```
return list
```

```
scalars:
                  r(N) =  19238
              r(sum_w) =  19238
               r(mean) =  .2344318536230377
                r(Var) =  .179482889232214
                 r(sd) =  .4236542095060711
                r(min) =  0
                r(max) =  1
                r(sum) =  4510
```

Getting values returned by commands

```
 reg union age south c_city

 ereturn list

scalars:
                  e(N) =  19226
               e(df_m) =  3
               e(df_r) =  19222
                  e(F) =  159.8965961359796
[...]
               e(ll_0) =  -10764.76183026799

macros:
            e(cmdline) : "regress union age south c_city"
              e(title) : "Linear regression"
                e(vce) : "ols"
[...]
          e(estat_cmd) : "regress_estat"

matrices:
                  e(b) :  1 x 4
                  e(V) :  4 x 4
```

Getting values returned by commands

```
// Show coefficients after reg
matrix list e(b)
```

```
e(b)[1,4]
            age        south      c_city        _cons
y1   .00242889  -.11547906   .06972916   .18224179
```

```
// Show coefficients' variance-covariance matrix
matrix covmat = e(V)
matrix list covmat
```

```
symmetric covmat[4,4]
                age        south      c_city        _cons
   age   2.399e-07
 south  -9.321e-08   .00003756
c_city   2.739e-07  -6.147e-07   .00004085
 _cons  -7.578e-06  -.00001244  -.00002226   .00025953
```

Getting values returned by commands

Let's calculate t-stat for the hypothesis that $H_0 : \beta_{south} = 0.5$.

In other words, we need to calculate:

$$t = \frac{\hat{\beta}_{south} - 0.5}{se(\hat{\beta}_{south})}$$

First, get $\hat{\beta}_{south}$:

```
// Long way (in this case there is a shorter way)
matrix betas = e(b)
local b_south = betas[1,colnumb(betas, "south")]

// Short way
local b_south = _b[south]
```

Getting values returned by commands

Second, get $se(\hat{\beta}_{south})$:

```
// Long way (in this case there is a shorter way)
matrix covmat = e(V)
local se_south = sqrt(covmat[rownumb(covmat, "south"), ///
                             colnumb(covmat, "south")])

// Short way
local se_south = _se[south]
```

Third, calculate:

```
local my_t = (‘b_south’-0.5) / ‘se_south’

di "t-stat for our strange hypothesis is : ‘my_t’"
```

Note that our use of the returned values freed the code from almost any specific numbers. If tomorrow we are using a different dataset, our code is very easy to adjust.

Extensions 1 - _variables

Besides the _b[] and _se[] we have other special variables that start with _:

- `_n` refers to the observation number:

```
sort school
gen     id_in_school = 1
replace id_in_school = id_in_school[_n-1] + 1 ///
                    if school == school[_n-1]
```

- `_cons` refers to the constant term in a regression - in the `_b[]` or `_se[]` context.
- `_N` contains the total number of observations in the dataset

The relevant help file is `help _variables`.

Extensions 2 - Extended functions

Extended functions allow you to put in more information in your macros.

Extended functions that deal with strings can also deal with strings longer than 255 characters, unlike the usual string functions.

To use them, you will need to define the macro with the colon sign:

```
local <macro name> : <extended-function>
```

Some which are worthy of notice:

- **word count <string>** will count the number of words in **<string>**
- **word <number> of <string>** will give you the **<number>**th word of the **<string>**

```
local fullname "Maria do Carmo Mao de Ferro e Cunha de Almeida"
local name_count : word count ‘fullname’
local last_last_name : word ‘name_count’ of ‘fullname’
```

Extensions 2 - Extended functions

- `variable label <varname>` - will put in the label of the variable in question

```
foreach var of varlist gdp_* {
    local cur_label : variable label `var'
    di "`var' - `label'"
}
```

- `type <varname>` - will give you the type of the variable.

```
foreach var of varlist * {
    local cur_type : type `var'
    // substr takes a substring from a string. In this case it will
    // start from the first three characters. So if "`cur_type'" is
    // "str34" or "str4", substr("`cur_type'", 1, 3) will be "str".
    if (substr("`cur_type'", 1, 3) == "str") {
        count if `var' == ""
    } else {
        count if `var' == .
    }
}
```

- More on extended functions on `help extended_fcn`
- For string functions (like substr): `help string functions`

Extensions 3 - `while` and `if`

`if` - up until now we used `if` as an argument of commands, to let them know which observations to work on. `if` can also be used to control the flow of the program - especially inside loops.

```
if <condition> {
    // Commands
}
[else if <condition> {
    // Commands
}]
[else {
    // Commands
}]
```

`while` - in addition to `foreach` and `forvalues` sometimes we don't know in advance how many iterations we will need. We just need to loop as long as some condition holds (for example, as long as we haven't reached convergence).

Extensions 3 - `while` and `if`

But be careful with **`while`**s, because if the condition will not be satisfied, you will enter a never-ending loop.

Usually, it's preferable to use some maximum number of iterations, in case there is some probability the usual condition will not work:

```
local converged = 0
local iter = 0
local max_iter = 800
while (!‘converged’ & ‘iter’ < ‘max_iter’) {
    // Commands that do something and check whether convergence
    // was achieved. If convergence was achieved it does
    // local converged = 1

    local iter = ‘iter’ + 1
}
```

Extensions 3 - `while` and `if`

But then, if we're already counting iterations, we might as well do it all with forvalues:

```
local converged = 0
local max_iter = 800
forvalues iter = 1/‘max_iter’ {
    // Commands that do something and check whether convergence
    // was achieved. If convergence was achieved it does
    // local converged = 1
    if ‘converged’ {
        continue, break
    }
}
```

`continue`, without the `break` option, stops the execution of the *current* iteration and goes on to the next iteration. With the `break` option it exits the loop altogether.