Introduction to Computing for Sociologists
Neustadtl

Stata Macros

`local` (`help macros` or `help local`)

- A Stata macro is a storage place, you put text in. You then use what's in storage in subsequent commands.
- A macro is simply a name associated with some text.
- Macros can be local or global in scope.

looping (over variables)

foreach (`help foreach`)

- foreach repeatedly sets a local macro *lname* to each element of a list and executes the commands enclosed in braces.
- The loop is executed zero or more times; it is executed zero times if the list is null or empty.

Return Results

`help return` (see also `help ereturn` and `help creturn`)

In addition to the output in the shown in the results window, many of Stata's commands store information about the command and its results in memory. This allows you, as well as other Stata commands, to easily make use of this information. Stata calls these "returned results". Returned results can be very useful when you want to use information produced by a Stata command to do something else in Stata.

Stata returned results can be placed into categories. Arguably, the most important categories are r-class, e-class, and c-class commands.

Macros

A macro is simply a name associated with some text. Whenever you want to use that text you can use the shorter macro as a substitute and Stata will translate the macro to your text. Macros can be local or global in scope. This document will mostly discuss local macros, global macros will only be discussed in passing.

Storing Text in Local Macros

Local macros have names of up to 31 characters and are known only in the place where they were created (the command, a do file, or a program).

You define a local macro using `local` *`lclname`* `[=] text` and you evaluate it using `` `lclname' ``. Note the use of a backtick or left quote. This key is typically on the left of most keyboards above the Tab key and is paired with the tilde key (~).

You can use the `=` sign or not but there is an important difference. Local macros created without an equal sign are used to store text up to ~165k characters (up to a million in Stata SE). The text is often enclosed in quotes but it doesn't have to be. Local macros created with the equal sign are limited to 244 characters because the macro is treated as a string variable (see `help limits`).

When you type: `local name "something"` or . `local name `"something"'`

`something` becomes the contents of the macro. The compound double quotes (`` `" `` and `"' `) are needed when something itself contains quotation marks.

When you type: `local name = something`, `something` is evaluated as an expression, and the result becomes the contents of the macro. Note the presence and lack of the equal sign.

So, if you type:

```
local problem "2+2"
local result = 2+2
```

then `problem` contains 2+2, whereas `result` contains 4.

Macros can be used when you need to run a large number of regression equations that include a standard set of control variables, say *sex*, *race*, *age*, *agesqr*, and *educ*. One option is to type these variable names in each equation (or cut and paste the names). These alternatives are tedious and error prone. Instead you can use a macro:

Without macros:

```
regress sexfreq1 sex race age agesq education.
regress reliten1 sex race age agesq education.
regress attend1  sex race age agesq education.
regress polviews sex race age agesq education.
regress childs   sex race age agesq education.
```

With macros:

```
local controls sex race age agesq education
regress sexfreq1 `controls'
regress reliten1 `controls'
regress attend1  `controls'
regress polviews `controls'
regress childs   `controls'
```

With only one regression to run you haven't saved anything; with many models with different dependent or independent variables macros saves work and ensures consistency.

If you decide you should have used log-income rather than income as a control, all you need to do is change the macro definition at the top of your do file, say to read *logincome* instead of income and all subsequent models will be run with income properly logged (assuming these variables exist).

Warning: evaluating a macro that doesn't exist is not an error; it just returns an empty string. So be careful to spell macro names correctly. If you type `regress sexfreq `contrls'`, Stata will read `regress sexfreq1`, because the macro `` `contrls' `` does not exist. The same would happen if you type `` `control' `` because macro names cannot be abbreviated the way variable names can. Either way, the regression will run without any controls. But you always carefully check your output, right?

Suppose you are working with a demographic survey where age has been grouped in five-year groups and ends up being represented by seven dummies, say *age15to19* to *age45to49*, six of which will be used in your regressions.

```
local controls sex race education
local age "age20to24 age25to29 age30to39 age35to39 age40to44 age45to49"
regress sexfreq1 `controls' `age'
```

Not only is this shorter and more readable, but also closer to what you intend, which is to regress reported sexual frequency on a set of control variables and "age", which happens to be a set of indicator variables. This also makes it easier to change the representation of age; if you later decide to use linear and quadratic terms instead of the six dummies all you do is define local age "age agesq" and rerun your models. Note that the first occurrence of age here is the name of the macro and the second is the name of a variable. I used quotes to make the code clearer. Stata never gets confused.

Keyboard Mapping with Global Macros

Global macros have names of up to 32 characters and as the name indicates have global scope—they continue to exist outside of the context in which they were created. You define global macros using `global name [=] text` and evaluate it using `$name`. You can use `${name}` to clarify where the name ends.

I suggest you avoid global macros because of the potential for name conflicts. A useful application, however, is to map the function keys on your keyboard. If you work on a shared network folder with a long name try something like this:

```
global F5 \\server\shared\research\project\subproject\
```

Then when you use the F5 key Stata will substitute the full name. Your do files can use commands like `do ${F5}dofile` (the braces to indicate that the macro is called F5, not F5dofile).

You probably don't want to type this macro each time you use Stata. So, enter it in your profile.do file, a set of commands that is executed each time you run Stata. Type `help profile` to learn more. If you are working on public computers you can create a small do file that has Stata commands to set up your computing environment and run it at the beginning of your work session.

More on Macros

Macros can also be used to obtain and store information about the system or the variables in your dataset using extended macro functions. For example you can retrieve variable and value labels, a feature that can come handy in programming.

There are also commands to manage your collection of macros, including `macro list` and `macro drop`. Type `help macro` to learn more.

Looping

Some programming tasks are repetitive and nothing is better at repetitions than a computer. The main looping command in Stata is `foreach` (see `help foreach`). From the help file:

Syntax

```
        foreach lname {in|of listtype} list {
                commands referring to `lname'
        }
```

Allowed are

```
    foreach lname in any_list {
    foreach lname of local    lmacname   {
    foreach lname of global   gmacname   {
    foreach lname of varlist  varlist    {
    foreach lname of newlist  newvarlist {
    foreach lname of numlist  numlist    {
```

Braces must be specified with **foreach**, and

1. the open brace must appear on the same line as the **foreach**;
2. nothing may follow the open brace except, of course, comments; the first command to be executed must appear on a new line;
3. the close brace must appear on a line by itself.

Say you wanted to summarize a large number of variables. You could use the `foreach` command in the following way:

```
. foreach var of varlist age educ sexfreq1 {
  2.   summarize `var'
  3. }
```

Variable	Obs	Mean	Std. Dev.	Min	Max
age	**39427**	**45.96807**	**17.49501**	**18**	**89**

Variable	Obs	Mean	Std. Dev.	Min	Max
educ	**39449**	**12.85328**	**3.081281**	**0**	**20**

Variable	Obs	Mean	Std. Dev.	Min	Max
sexfreq1	**19711**	**55.69299**	**65.99505**	**0**	**237**

The command begins with `foreach`. This is followed by *lname* which is a local macro name, in this case, *var*. Since I am summarizing variables the next part of the command is `of varlist` followed by the list of variables to be stored in the local macro *var* (*age*, *educ*, and *sexfreq1*). Finally, the line ends with an open bracket ({). Stata commands that are to be repeated across the variable list are inserted on the next line (or several lines). In this case `summarize `var'`. Note that *var* is enclosed in the macro-specific quotes (` and ‘) because it is a local macro. The following Stata code produces the same output and has the advantage of only changing the values of `local varsum` in case we have lots of analyses of those variables.

```
local varsum “age educ sexfreq1”
foreach var of varlist `varsum’ {
  summarize `var’
}
```

Loops can be used to create new variables. Consider the following example:

```
foreach var of varlist socrel socommun socfrend socbar {
  recode `var' (1=1) (2/7=0) (else=.), gen(`var'1)
}
```

This `foreach` loop creates four new dummy variables (*socrel1*, *socommun1*, *socfrend1*, and *socbar1*) that are coded 1 if the respondents socialize "almost daily" and 0 otherwise. Note how the `generate` option substitutes the original variable name with a "1" appended to it.

This example is a little more complicated not because of the looping but because it uses other commands (`egen`) and a logical expression that evaluates to either 0 or 1 to create the dummy variable. In this example cases coded 1 have values greater than the median for the list of variables. Note that the original variables names are used as a prefix for the generated variable name:

```
foreach x of varlist educ prestg80 wordsum {
  egen median_`x'  = median(`x')
  gen byte `x'_above = (`x'>median_`x') & !missing(`x')
  label var `x'_above "`x' above median dummy"
  drop median_`x'
 }
```

The following example creates three dummy variables based on the race variable in the GSS by looping over the values of the race variable. Note how the numbers are appended to the variable names and labels:

```
foreach num of numlist 1(1)3 {
  gen racedum`num'=(race==`num') if race !=.
  label var racedum`num' "Dummy for race=`num'
}
```

Problems

1. Using loops (i.e. `foreach`) create two new variables called *leduc* and *lage* that are logged versions of *educ* and *age*.

2. This problem uses a lot of different Stata concepts. Write a loop to produce the following table:

```
1 -1.9203678 .18697281
2 -2.3836519 .23321738
3 -2.1965718 .12503279
4 -2.9450347 .17872824
5 -1.1439784 .027275
```

   Columns 1, 2, and 3 respectively represent 1) the value of *marital*, 2) the regression coefficient from regressing *sexfreq3* on *age*, and 3) the r^2 from each of these models based on 2012 data.

   To create this table you will need to do several things.

   a) Create a sexual frequency measure as follows:

```
/* Create the yearly sexual frequency variable */
capture drop sexfreq3
generate sexfreq3=.
replace  sexfreq3=  0 if sexfreq==0
replace  sexfreq3=  2 if sexfreq==1
replace  sexfreq3= 12 if sexfreq==2
replace  sexfreq3= 36 if sexfreq==3
replace  sexfreq3= 52 if sexfreq==4
replace  sexfreq3=156 if sexfreq==5
replace  sexfreq3=208 if sexfreq==6
label variable sexfreq3 "Yearly sexual frequency"
```

   b) You need to estimate five regression models, one for each value of marital status, of *sexfreq3* regressed on *age*. Anytime you see something repetitive it might make sense to loop.

   c) You need to suppress some of the output (i.e. the regression results) but not others (i.e. the coefficients, etc.). You can suppress output using the `quietly` command (`help quietly`).

   d) You need to use the `display` command (`help display`) to send the values marital status, the regression coefficients, and the r^2's to the screen. So, you need to know how to access the values stored in the `e()` saved results (`help ereturn`) and how to access the regression coefficients stored in the matrix of coefficients `e(b)` (`help regress` and `help _variables`).

   e) Put all this together to create the listing of results for each level of marital status.

3. This is not a problem but another example to demonstrate how great looping can be to reduce your work and errors. The following code used the `ds` command (`help ds`) to identify all numeric variables in a dataset and store their names in the `r()` local macro called r`(varlist)`. This local macro is then used in a `foreach` loop to create *z*-transformed variables:

```
/* Create z-scores from all variables in the dataset */
ds, has(type numeric)
foreach var of varlist `r(varlist)' {
  egen `var'_z=std(`var')
}
```

The `ds` command is very powerful and worth exploring. A new user-written command called `findname` is very similar and has more features (`findit findname`).

r-class

In addition to producing output in the results window, commands like summarize return results in temporary variables in `r()`. These results generally must be used before executing more commands that replace the current values of `r()`. To see the `r()` variables you type `return list` after issuing a command like `summarize`. For example:

```
sum age
return list
```

```
. sum age

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         age |     56859    45.69795    17.47211         18         89

. return list

scalars:
                  r(N) =  56859
              r(sum_w) =  56859
               r(mean) =  45.69795458942296
                r(Var) =  305.2746961222475
                 r(sd) =  17.47211195368916
               r(min) =  18
               r(max) =  89
               r(sum) =  2598340
```

All statistics from the summarize command are saved temporarily in `r()` until the next command is called. Here we can see that 56,859 valid cases were used in this command [`r(N)`] and that the mean of age is 45.969795…[`r(mean)`] with a standard deviation of 17.47211…[`r(sd)`]

To create a *z*-transformed variable for *age* we can use the return values from the summarize command and the `generate` command to create a new variable with mean 0 and standard deviation 1. This must be done after using the summarize command for *age* and before another command that will write-over/replace (i.e. destroy) the values of `r()`:

```
generate age_z=(age-r(mean))/r(sd)
```

An easier way to do this is to use `egen` (but that doesn't demonstrate using local variables):

```
egen age_z=std(age)
```

e-class

Commands that return results in e-class variables are *estimation* commands such as regress, logistic, etc., that fit statistical models. The estimation results are available until the next model is fitted. To see the results of an e-class command type `ereturn list` after your estimation command. For example:

```
regress sexfreq1 age agesqr
ereturn list
```

```
scalars:
                  e(N) =  25520
               e(df_m) =  2
               e(df_r) =  25517
                  e(F) =  2231.312876856802
                 e(r2) =  .1488552855987808
               e(rmse) =  62.71995490116255
                e(mss) =  17555044.8038082
                e(rss) =  100378589.4181262
               e(r2_a) =  .1487885736252415
                 e(ll) =  -141828.9161256346
               e(ll_0) =  -143885.4850435673
               e(rank) =  3

macros:
           e(cmdline) : "regress sexfreq3 age agesqr"
             e(title) : "Linear regression"
          e(marginsok) : "XB default"
                e(vce) : "ols"
            e(depvar) : "sexfreq3"
               e(cmd) : "regress"
        e(properties) : "b V"
           e(predict) : "regres_p"
             e(model) : "ols"
          e(estat_cmd) : "regress_estat"

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

functions:
             e(sample)
```

Even though the model *R*-squared and *F*-statistic are provided in the results window as well as in `e(r2)` and `e(F)`, they can be calculated using e-class variables:

```
di e(mss)/(e(rss) + e(mss))           /* R-squared   */
di (e(mss)/e(df_m))/(e(rss)/e(df_r))  /* F-statistic */
```

The regress command also stores values in matrices—Stata has a robust matrix programming language. The matrix `e(b)` contains the regression coefficients and can be listed using the command `matrix list e(b)`. Individual matrix elements may also be accessed using the `_b[varname]` convention. The following commands 1) show the contents of the matrix `e(b)`, display the predicted value of *sexfreq1* for the first case in the dataset, and 3) creates a new variable, *yhat*, containing the predicted values (yes, it is easier to use the `predict` postestimation command):

```
matrix list e(b)
di _b[_cons] + (_b[age]*age) + (_b[agesqr]*agesqr)
generate yhat=_b[_cons] + (_b[age]*age) + (_b[agesqr]*agesqr)
```

```
. matrix list e(b)

e(b)[1,3]
           age      agesqr       _cons
y1  -.60401117  -.00949179   109.82719

. di _b[_cons] + (_b[age]*age) + (_b[agesqr]*agesqr)
80.779246

. generate yhat=_b[_cons] + (_b[age]*age) + (_b[agesqr]*agesqr)
(202 missing values generated)
```

Storing Results in Local Macros

The second type of macro definition `local name = text`, with an equal sign is used to store results. It instructs Stata to treat the text on the right hand side as an expression, evaluate it, and store a text representation of the result under the given name.

You run a regression and want to store the resulting *R*-squared, for comparison with a later regression. The `regress` command stores *R*-squared in `e(r2)`, so you think `local rsq e(r2)` would do the trick.

But it doesn't. Your macro stored the formula `e(r2)`, as you can see by typing display "`rsq'". What you needed to store was the value. The solution is to type `local rsq = e(r2)`, with an equal sign. This causes Stata to evaluate the expression and store the result.

To see the difference try this

```
regress sexfreq1 age
local rsqf e(r2)
local rsqv = e(r2)
di `rsqf'          /* This has the current R-squared    */
di `rsqv'          /* as does this one                  */

regress sexfreq1 age sex race
di `rsqf'          /* This has the new R-squared        */
di `rsqv'          /* This has the old one              */
```

Another way to force evaluation is to enclose `e(r2)` in single quotes when you define the macro. This is called a macro expression, and is also useful when you want to display results. It allows us to type `display "R-squared=`rsqv'"` instead of `display "R-squared=" `rsq'.`

An alternative way to store results for later use is to use scalars (type `help scalars` to learn more). This has the advantage that Stata stores the result in binary form without loss of precision. A

macro stores a text representation that is good only for about 8 digits. The downside is that scalars are global so there is a potential for name conflicts, particular in programs (unless you use temporary names, see `help tempvar`).

c-class

Stata's c-class, `c()`, contains the values of system parameters and settings, along with certain constants such as the value of pi. `c()` values may be referred to but may not be assigned. To see all of the c-class variables type `creturn list`. To use one c-class variable as an example we can display the contents of the variable `c(current_date)`:

```
di c(current_date)
```

```
. di c(current_date)
21 Jan 2009
```

You can learn more about these stored values by looking at the online help files (`help return`, `help ereturn`, and `help creturn`). These returned values are macro variables or simply macros.