Lesson 5: Macros and Retrieving Information

This lesson covers: using macros and retrieving information from stored results.

5.1 Macros

So far, we've learnt about how to view, summarise and manipulate variables. We'll now learn about **macros**, which are completely distinct from variables and observations, and an incredibly helpful feature of Stata.

A **macro** in Stata is a piece of information, which can be assigned to a name and then referenced whenever you like. That piece of information, the contents of the **macro**, can be a number, a string, an expression, a collection of numbers or strings, a list of variable names: essentially, the contents of a **macro** can be virtually anything. When referenced, the contents of the **macro** are used as if they had been typed out in full, so if the **macro** contains a number, it can be used in an expression; if the **macro** contains a variable name, then that variable will be used in a command where the **macro** is referenced instead of the variable name, and so on.

The term "**macro**" has many different meanings in different contexts. A **macro** in Excel is a set of commands that you can tie together and perform together on command, and a keyboard **macro** is similar and allows you to assign a keyboard button to perform a set of button presses: a Stata **macro** is not at all like these things, it is just some information tied to a name. Equally, Stata **macros** have nothing to do with micro/macroscopic. If you are an R user, then Stata **macros** are the equivalent of values.

We'll cover both **local** and **global macros**, which have very similar operations but are referenced differently. First off, let's set the working directory and load in the dataset we saved at the end of the last lesson, or **Lesson_05.dta** if you want a fresh dataset.

local macros

First, an example to show how **macros** operate (if you are using a do file, run both lines together):

```
local a = 3
display `a'
```

```
3
```

In this bit of code, we use the **`local`** command to assign the value of 3 to a **local macro** called *a*, and then we made Stata **`display`** the value of *a*. The name of a **local macro** can be up to 31 characters long (32 for **global macros**), but they can start with a number (unlike variable names). To reference a **local macro**, we put the name we gave the **macro** between single left and right quotes, e.g. **`a'**. Note that **macros** are not variables, so you won't find them in the variables window. Like value labels, we need to use a special command to view all the **macros** in memory:

```
macro list
S_FNDATE:        6 Mar 2021 11:55
S_FN:            Lesson_04_SH.dta
S_level:         95
F1:              help advice;
F2:              describe;
F7:              save
F8:              use
S_ADO:           BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
S_StataSE:       SE
S_FLAVOR:        Intercooled
S_OS:            Windows
S_OSDTL:         64-bit
S_MACH:          PC (64-bit x86-64)
_a:              3
```

Depending on how your version of Stata is set up, this information may all be different. However, at the bottom there should be an entry for **_a**, which should read **3**: **local** macros have an underscore when listed, as opposed to **global** macros, which don't. When viewed like this, **global** macros are listed first and **local** macros follow: within those categories, the most recently assigned macro appears at the top and the least recently assigned macro appears at the bottom.

If you are using a do file, note that **local macros** only exist while the do file is running; as soon as the commands you've executed stop, all **macros** you've defined are deleted. Therefore, whenever you are using **local macros**, make sure you run all the code you need at once or you'll typically get an error message. If you're using the command line, then a **local macro** will persist until you close Stata: even using **`clear`** doesn't remove **macros** from memory. To remove **macros**, you need to use the same type of command as labels (note, to remove **local macros**, you need to put an underscore at the front of the **macro** name):

```
macro drop _a
macro list
S_FNDATE:        6 Mar 2021 11:55
S_FN:            Lesson_04_SH.dta
S_level:         95
F1:              help advice;
F2:              describe;
F7:              save
F8:              use
S_ADO:           BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
S_StataSE:       SE
S_FLAVOR:        Intercooled
S_OS:            Windows
S_OSDTL:         64-bit
S_MACH:          PC (64-bit x86-64)
```

However, note that **macros** are rarely deleted, because they take up little memory and don't clutter up Stata like excess variables do. **Macros** can also be overwritten without any special options, unlike variables, so there's no need to drop a **macro** before reassigning it.

When we create **macros**, the contents can be a single value (as above), a list of values, an expression, a string, variable names, other macros, even individual values of variables. When referenced, Stata will deal with the contents appropriately.

Let's go through some examples:

```
local number = 5
dis `number' * 10 + 2
52

local list_of_numbers 1 2 3 4 5
dis `list_of_numbers'
12345
```

When giving Stata a list of numbers like this, you can use them in **loops**, which we'll cover in the next lesson. You also don't *need* to use an equals sign to assign **macros**, except when you are assigning an expression (though I almost always use an equals sign for all **macros** I create):

```
local expression = 5*10+2
dis `expression'
52
```

When assigning strings to **macros**, the string doesn't have to be in quotation marks (though they don't hurt and I prefer to use them), but when you reference a **macro** containing a string, the **macro** has to be in quotation marks *as well as* the single left and right quotes, otherwise Stata will assume you're referencing a variable name:

```
local string This is a string
dis "`string'"
This is a string

local string "This is also a string"
dis "`string'"
This is also a string
local string "This is also a string, but strings need quotes"
dis `string'
This not found
r(111);
```

The error message is because the variable *This* doesn't exist. However, we have plenty of variables that do exist, so we can specify those in a **macro** then reference them in a command. This can be useful if you have, for example, a group of variables that you want to use in several commands:

```
local variables id age sex bmi calories
```

```
sum `variables'
```

Variable	Obs	Mean	Std. Dev.	Min	Max
id	1,000	500.5	288.8194	1	1000
age	920	50.68043	17.72256	20	80
sex	994	.5181087	.4999235	0	1
bmi	905	26.98933	3.204688	17.42509	37.25286
calories	987	2150.578	334.3657	1464	3253

You can combine as many **macros** as you want inside other **macros**:

```
local macro_1 = 5
local macro_2 = 10
local macro_3 = 2
local macro_4 = `macro_1' * `macro_2' + `macro_3'
dis `macro_4'
52
```

As a final example, you can assign the value of a specified observation number of a variable to a **macro**:

```
local age_1 = age[1]
dis `age_1'
78
list age in 1
```

	age
1.	78

This is a useful trick in Stata: you can reference any observation of any variable by typing in the variable, immediately followed by the observation number in square brackets. As you can't use *if* and *in* with **macros**, it is useful to be able to specify exactly the observation you want.

global macros

If you are using the command window to input commands, the only notable difference between **local** and **global macros** is that you assign and reference them differently:

```
global a = 3
dis $a
3
```

Instead of using the **`local`** command and referencing the **macro** with single left and right quotes, you use the **`global`** command and reference the **macro** with a dollar sign at the start of the **macro** name. Otherwise, you specify the contents of the **macro** in exactly the same way.

However, if you are using a do file, then the difference is that **global macros** persist even when the do file stops running: once a **global macro** is defined, whether in the command window or a do file, it will continue to exist until Stata is closed.

5.2 Stored Results

When you issue a command in Stata, some information is usually stored in memory, which you can then access. For example, when you **`summarize`** a variable, all the information that is displayed on screen can be accessed, meaning you don't need to read through the results in order to use them in future commands.

return list

There are two different commands commonly used to show this stored information, depending on the initial command: **`return list`** shows information stored from general commands (such as **`summarize`**), and **`ereturn list`** shows information from estimation commands (such as from regressions and models). Let's use **`summarize`** as an example:

```
sum bmi
```

Variable	Obs	Mean	Std. Dev.	Min	Max
bmi	905	26.98933	3.204688	17.42509	37.25286

```
return list
scalars:
                  r(N) =  905
              r(sum_w) =  905
               r(mean) =  26.9893308860821
                r(Var) =  10.27002308567298
                 r(sd) =  3.204687673654483
                r(min) =  17.42509078979492
                r(max) =  37.25286483764648
                r(sum) =  24425.3444519043
```

When we use the **`return list`** command, we see all the information that was stored after the previous command. A **scalar** is like a **macro**, in that it stores some information, but **scalars** are generally created by commands rather than users directly. We can see the number of non-missing observations in **r(N)**, the mean in **r(mean)**, variance in **r(Var)**, standard deviation in **r(sd)**, minimum value in **r(min)**, maximum value in **r(max)**, and the total of all values in **r(sum)**. We can access all the information using the **r-class scalars** listed by referencing them (called **r-class** because all the values start with **r**):

```
dis r(N)
905
```

We can use the **r-class scalars** like **local macros**, though note that the values only exist until we run another command that stores different **r-class** values. Alternatively, we could store the information in **macros**, which we can then use at our leisure. In general, I prefer to assign **r-class scalars** to **macros**, since there's less chance I'm going to accidentally run another

command that will overwrite the **r-class** values. For example:

```
local N = r(N)
local mean = r(mean)
dis "The number of observations is `N', the mean is `mean'"
```

```
The number of observations is 905, the mean is 26.9893308860821
```

If you remember back to last lesson, we can use the **`string()`** function to format the mean a little better:

```
local N = r(N)
local mean = r(mean)
dis "The number of observations is `N', the mean is
"string(`mean',"%9.2f")
```

```
The number of observations is 905, the mean is 26.99
```

Let's try using **`summarize`** to normalise a variable, that is, transform the variable so it has a mean of 0 and a standard deviation of 1. We can do this by summarising a variable, subtracting the mean (the new mean will be 0), then dividing by the standard deviation (the new standard deviation will be 1).

Quick note for those unfamiliar with means and standard deviations: the mean of a variable is when you add up all the values and divide by the number of values (often called the "average", but "average" could mean a few things depending on who says it), and the standard deviation is a measure of the spread of the values – if the values are highly spread out, then the standard deviation will be big, and vice versa.

Let's create a new variable, *bmi_norm*, which will be a normalised version of *bmi*:

```
quietly sum bmi
local mean = r(mean)
local sd = r(sd)
gen bmi_norm = (bmi-`mean')/`sd', a(bmi)
```

```
(95 missing values generated)
```

```
sum bmi_norm
```

Variable	Obs	Mean	Std. Dev.	Min	Max
bmi_norm	905	7.72e-10	1	-2.984453	3.202663

```
label variable bmi_norm "Body mass index (normalised)"
```

I've introduced the **`quietly`** command (short: **`qui`**) here, which suppresses the output of the following command, reducing the amount of clutter that appears in the results window. Notably, when you **`quietly`** run commands, all the information from the command is still

stored and available, and you can use it at your leisure, as we did here. When we generated *bmi_norm*, we subtracted the mean, then divided by the standard deviation (the brackets were necessary here), and we can see from the summary of *bmi_norm* that the mean is very close to 0 (not exactly 0, but 0 to 9 decimal places), and the standard deviation is 1. Therefore, we can say that *bmi_norm* is nice and normalised.

Using options may change the amount of information stored by a command. If we add the **`detail`** option to **`summarize`**, we both see more information, and more information is stored:

```
sum age, detail
```

```
                            Age (years)
-------------------------------------------------------------
      Percentiles      Smallest
 1%           20            20
 5%           23            20
10%           26            20       Obs                 920
25%           35            20       Sum of Wgt.         920

50%           51                     Mean           50.68043
                        Largest       Std. Dev.      17.72256
75%         66.5            80
90%           75            80       Variance       314.0893
95%           78            80       Skewness      -.0410042
99%           80            80       Kurtosis       1.802987
```

```
return list
```

```
scalars:
                  r(N) =  920
              r(sum_w) =  920
               r(mean) =  50.68043478260869
                r(Var) =  314.0892794625538
                 r(sd) =  17.72256413340219
           r(skewness) =  -.0410041938073991
           r(kurtosis) =  1.802986873702794
                r(sum) =  46626
                r(min) =  20
                r(max) =  80
                 r(p1) =  20
                 r(p5) =  23
                r(p10) =  26
                r(p25) =  35
                r(p50) =  51
                r(p75) =  66.5
                r(p90) =  75
                r(p95) =  78
                r(p99) =  80
```

In addition to the **r-class scalars** for **`summarize`**, we have skewness in **r(skewness)**, kurtosis in **r(kurtosis)**, and various percentiles, for example the 50th percentile (the median) in **r(p50)**. You could also find the interquartile range with **r(p25)** and **r(p75).**
The information that is stored for each command can be found at the bottom of the relevant help file. For instance, for **`tabulate`**, we can see at the bottom of the help file (selecting **`tabulate oneway`**):

```
help tab
```

Stored results

tabulate and **tab1** store the following in **r()**:

Scalars
- **r(N)** number of observations
- **r(r)** number of rows

When we use **`tabulate`**, Stata will store the number of observations that were tabulated in **r(N)** and the number of rows in the tabulation in **r(r)**. Let's try it:

```
tab hair_colour
```

Hair colour	Freq.	Percent	Cum.
Black	199	20.29	20.29
Blonde	288	29.36	49.64
Brown	294	29.97	79.61
Grey	97	9.89	89.50
Red	103	10.50	100.00
Total	981	100.00	

```
return list
scalars:
                  r(N) =  981
                  r(r) =  5
```

We now have two **r-class** values, as the help file said we would.

ereturn list

I won't go into detail about **`ereturn list`**, because we're not going to use estimation commands in this course (that's a different course altogether). But for completeness, let's do a quick example. If this section makes no sense to you, do not worry at all and feel free to skip straight to the exercises. If you are interested in estimation in Stata, other courses will go through different commands in detail (hopefully including a future course of mine):

```
regress bmi age
```

Source	SS	df	MS	Number of obs	=	831
				F(1, 829)	=	0.01
Model	.109777589	1	.109777589	Prob > F	=	0.9180
Residual	8575.77609	829	10.3447239	R-squared	=	0.0000
				Adj R-squared	=	-0.0012
Total	8575.88587	830	10.3323926	Root MSE	=	3.2163

bmi	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
age	-.0006471	.0062816	-0.10	0.918	-.0129769	.0116827
_cons	27.04441	.3374456	80.14	0.000	26.38206	27.70676

ereturn list

```
scalars:
                  e(N) =  831
               e(df_m) =  1
               e(df_r) =  829
                  e(F) =  .010611940018593
                 e(r2) =  .0000128007288034
               e(rmse) =  3.21632148228331
                e(mss) =  .1097775892958453
                e(rss) =  8575.7760943622
               e(r2_a) =  -.0011934564476395
                 e(ll) =  -2148.942751164406
               e(ll_0) =  -2148.948069901265
               e(rank) =  2

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

matrices:
                  e(b) :  1 x 2
                  e(V) :  2 x 2

functions:
             e(sample)
```

The regression looks at how *bmi* changes as *age* increases (it doesn't), and much of the information displayed in the regression output can be seen either as an **e-class scalar** (i.e. **scalars** that start with an **e**, including number of observations in **e(N)** and the R^2 value in **e(r2)**, referenced the same as **r-class scalars**), or in the **e(b)** and **e(V) matrices**.

We also won't cover **matrices** here, but again for completeness, you can think of **matrices** as **macros** that contain a table of information, rather than a single piece of information. You could also think of them as small, temporary datasets without any labelling. Here, the **e(b)** matrix contains the coefficients for *age* and the constant in a table with a single row and 2 columns, and the **e(V)** matrix contains a 2x2 table with the variances and covariances of the *age* and constant estimates. We could display the **matrices** with:

```
matrix list e(b)
```

```
e(b)[1,2]
          age       _cons
y1  -.0006471   27.044411
```

```
matrix list e(V)
```

```
symmetric e(V)[2,2]
             age       _cons
  age   .00003946
_cons   -.0020005   .11386957
```

Note that the variance of an estimate is the square of the standard error, so 0.0062816 (standard error of the *age* coefficient) squared is 0.00003946 (the variance of *age* in the **e(V) matrix**):

```
dis 0.0062816^2
```

```
.00003946
```

Let's save the dataset with our initials and the lesson number, and we'll move on to the exercise:

```
save "Lesson_05_SH.dta", replace
```

5.3 Exercise

For this exercise, see how you do with the following:

1. Change your directory (if necessary) and load in **Exercise_04_{initials}.dta**
    a. If you're not confident with your dataset, load in **Exercise_05.dta**

2. Assign the number 12 to a local macro called *number*
    a. Then display this macro multiplied by 8
    b. Then reassign the macro *number* as *number* squared
    c. Then display this macro divided by 3
    d. Finally, divide the macro by 3, add 1, and display the square root of the result
        i. Note: you can use the sqrt() function to find the square root

3. Take your full name, and create local macros for each name, then display your full name using these macros
    a. Now use the same macros to display your full name all in upper case letters

4. Create several global macros, containing:
    a. Today's date, stored as a Stata date code (i.e. the number)
    b. Today's day as day of the week (Monday, Tuesday etc.)
    c. Today's day as the numeric day of the month (1st, 2nd, 3rd etc.)
    d. Today's month (January, February etc.)
    e. Today's year (2021, 2022 etc.)
    f. Then use those 4 global macros to display the following message, appropriate for your current date:
    g. "Today is Sunday the 7th of March, 2021, which in Stata code is: 22346"

5. Create local macros for height, weight and BMI in the 2nd observation: set the macro for BMI to 1 decimal place
    a. Then create a second macro for BMI using the height and weight macros, also to 1 decimal place
    b. Then compare the two BMI macros in the same display command

6. Quietly summarise weight, then use global macros to record the median and interquartile range (50th, 25th and 75th percentiles respectively), the minimum and maximum values, and display these numbers to 1 decimal place in the same command

7. Create a normalised version of height, label it, and order it after height
    a. Summarise the new variable to be sure it's normalised

8. Save the modified dataset, calling it **Exercise_05_{initials}.dta**

5.4 Exercise - Answers

1. As before:
    a. `use "Exercise_04_SH.dta", clear`
    b. Or, if you're not confident in your save from exercise 4:
    c. `use "Exercise_05.dta", clear`

2. **local macros**:
    a. `local number = 12`
    b. `dis `number'*8`
    c. `local number = `number'^2`
    d. `dis `number'/3`
    e. `dis sqrt(`number'/3+1)`

3. **local macros**:
    a. `local first_name "Sean"`
    b. `local last_name "Harrison"`
    c. `display "`first_name' `last_name'"`
    d. `display upper("`first_name' `last_name'")`

4. **global macros**:
    a. `global date = date("07/03/2021","DMY")`
    b. `global day_week = "Sunday"`
    c. `global day_month = "7th"`
    d. `global month = "March"`
    e. `global year = 2021`
    f. `dis "Today is $day_week the $day_month of $month, $year, which in Stata code is: $date"`

5. **local macros**:
    a. `local height = height[2]`
    b. `local weight = weight[2]`
    c. `local bmi = string(bmi[2],"%9.1f")`
    d. `local bmi2 = string(`weight'/`height'^2,"%9.1f")`
    e. `dis "BMI in 2 = `bmi', and BMI from height/weight = `bmi2'"`

6. **`summarize`**, and **global macros**:
    a. `qui sum weight, d`
    b. `global median = string(r(p50),"%9.1f")`
    c. `global lower = string(r(p25),"%9.1f")`
    d. `global upper = string(r(p75),"%9.1f")`
    e. `global min = string(r(min),"%9.1f")`
    f. `global max = string(r(max),"%9.1f")`
    g. `dis "The median (IQR) of weight is: $median ($lower to $upper), and the range is $min to $max"`

7. **`summarize`**, and **local macros** (you could also use the **r-class scalars**):
    a. `qui sum height`
    b. `local mean = r(mean)`
    c. `local sd = r(sd)`
    d. `gen height_norm = (height-`mean')/`sd', a(height)`
    e. `sum height_norm`
    f. `label variable height_norm "Height(normalised)"`
    g. `su height_norm`

8. And a **`save`** command:
    a. `save "Exercise_05_SH.dta", replace`