Local and Global
Macros, More Looping!

Macros

- A Stata Macro is a named object that holds value for the user
    - A Macro has a **name** and **contents**
- Assigning a value to a Stata macro is a useful way to hold on to a result or number that the programmer wants to use again later (to compare to another result for instance)
- Additionally, macros can be used as shortcuts to make code more readable and easier to write
- Macros are called in a script using the backtick + quote (`**macro_name'**) convention (the same as iterators in loops)
- I will accidentally refer to a macro as a “variable” (I’m sorry)

Local vs. Global Macros

- Local Macros are defined and are visible “locally”
    - Locally means they are defined in a particular do-file or interactive session
    - Once a session is closed or a new do-file is run, the contents of local macros will be lost
- This allows you to use the same macro names across different do-files without worrying

Local vs. Global Macros

- Global Macros are defined and are visible “globally”
    - This global scoping means that a macro defined in one do file will be carried over to another do-file
    - Pursuant to Guttmacher’s Coding Style Guide we’ll leave Global Macros to advanced users who feel comfortable using them
- **Only Use Local Macros!**

local string

```
local local_name string
```

- This command will assign the content *string* to the local macro named `local_name`
- The string stored in the local macro will be treated exactly as if we typed out the exact same characters -- it is like we just cut and pasted words in. This means it will be **evaluated only when it is used**
- This means that a macro defined as a string **can change depending on intervening code**

```
local important_vars price headroom trunk

summarize `important_vars'
```

this is exactly the same as:

```
summarize price headroom trunk
```

`local =`

```
local local_name = expression
```

- This command will assign the result of the expression *`expression`* to the local macro named `local_name` **immediately**

- This local macro results in a **constant** no matter when it gets called later on in a script

```
local variable_mean = r(mean)
```

```
local price_cutoff = 5000
```

string vs. =

- We can examine this difference by doing a summarize on the data set

```
summarize price

local mean_price_string r(mean)

local mean_price_equals = r(mean)

display `mean_price_string'

display `mean_price_expression'
```

- Now if we do another summarize..

```
summarize mpg

display `mean_price_string'  <- this is as if we typed display r(mean)

display `mean_price_equals' <- this is as if we typed display -.00002979
```

`string vs. =`

- We see that the local macro contents change!
- The macro contents when defined with an = remain as the same constant entry, but the macro contents for the string are evaluated at the moment we use the **`display`** command (because it's like we just cut and paste the string)
- Taking a look at **`macro dir`** will also reveal the current contents of macros
  - *This command needs to be run along with the local command in order to work*

Macro Advice

- My advice to you on creating macros:
  - If you want your macro to equal a constant **number** *that will not change*, **use =**
    - `local cutoff = 3100`
    - `local mean_result = r(mean)`
  - If you want your macro that will essentially be a shortcut for typing, **do not use an equal sign**
    - `local ind_vars price mpg headroom`
    - `local my_scatter_options mlabel(S) mcolor(forest_green)`

Using `strings` and `expressions` in a loop

```
sysuse auto, clear

local vars_of_interest headroom trunk

foreach var in `vars_of_interest' {

   regress `var' price

   local `var'_price_coefficient = _b[price]

}

display `headroom_price_coefficient'

display `trunk_price_coefficient'
```

The power of local macros + loops becomes apparent!

The flexibility of `strings`

```
local drop_condition price > 5000

drop if `drop_condition'
```

Or another example

```
local keep_conditions (price > 5000 & mpg < 25) | ///

      (trunk < 2 & headroom < 2)

keep if `keep_conditions'
```

- This use of local macros can make code more readable and vastly reduce retyping or cutting/pasting that leads to typos and human errors

The flexibility of `strings`

```
local filenames cancer bpwide bplong

foreach file in `filenames' {

 webuse `file', clear

 notes: Cale's Personal Version 7/27/17

 save `file'.dta, replace

}
```

Here we see that the string local macro can help us easily iterate over a set of files and save local copies of them

Local Macro Guidelines

- Using local macros in your scripts:
  - allows you to hold on to intermediate results (=)
  - easily perform a series of calculations (=)
  - makes your code more readable to others (string)
  - less typo or human error-prone for repeated options (string)
- But local macros that are poorly named (local_1, thing_22) and poorly commented can make your scripts even harder to decipher
- Debugging advice: If you are getting odd behavior in your do files, check any local macros first!

Exercises (1)

1. Auto Data

   A. Define a local macro named cheap_foreign_car. We want to use this macro to subset **price < 5000 & foreign == 1** (string macro)

   B. Perform a summarize on only "cheap foreign cars" in the dataset using the macro from A.

   C. Using local macros to hold on to stored results (= macro), compare whether the minimum price of foreign cars is less than the minimum price of domestic cars. You can just display these minimum values, or write an if/else to print messages.

Exercises (2)

1. Auto Data

   A. We want a script that compares the correlation between mpg and price and the correlation between headroom and price. We want to create a scatter plot for whichever pair of variables has a correlation with greater absolute value.

Hint 1: you'll want to use **abs**(x) to get absolute value of x.

Hint 2: the stored result for correlation coefficient from command corr is r(rho)

Hint-3: you can use a loop but you don't have to!

Stata Documentation:
SVY Example

Stata Documentation

- Stata has excellent help files accessible via the **`help`** command
- These files are a great asset for recalling commands, syntax, and brief examples
- These curated help files are also excellent for graphing details (which are very difficult to memorize due to the sheer number of options)

Help File Links

- At the top of many help files is also a link to **more documentation** in the *Stata Reference Manual* and *Stata User's Guide*. Let's take a look at **`help graph`** and **`help regress`**

Extended Documentation

- In addition to the in-program help files and the *Stata Reference Manual*, there are topic-specific guides provided by StataCorp: http://www..stata.com/features/documentation/

- Here you can find detailed information on the commands and tools required to perform more complex analyses or write your own programs

SVY Tools Disclaimer

- Survey design and analysis are not areas of expertise for me statistically
- Please treat this section as an example of how to explore Stata's rich documentation — interrupt and correct me often!
- We can go slowly and explore the methods / information that is most relevant to your needs

SVY Prefix

- The **`svy`** prefix allows for the proper analysis of survey data
- Stata's survey data analysis tools help the user take into account:
  - Sampling weights
  - Cluster sampling
  - Stratification

`svyset`

- In order to properly adjust analyses for survey data, Stata needs to know the type of survey data loaded
- The first step when dealing with survey analysis in Stata is to define the survey settings of the dataset using the **`svyset`** command
- Some data sets (.dta files) will already have survey settings (a blank **`svyset`** call will show current settings)
- Lets look at the **`help svyset`** to learn more about how to set up a survey dataset

svyset commands

```
svy : tabulate v1 [v2]

svy: mean

svy : regress
```

- With a correctly designated **svyset** statement at the beginning of analysis, these commands will return accurately adjusted results based on complex survey data

SVY `subpop` **NOT** `if/in`

Survey data concepts

The variance estimation methods that Stata uses are discussed in [SVY] **variance estimation**.

Subpopulation estimation involves computing point and variance estimates for part of the population. This method is not the same as restricting the estimation sample to the collection of observations within the subpopulation because variance estimation for survey data measures sample-to-sample variability, assuming that the same survey design is used to collect the data. Use the `subpop()` option of the `svy` prefix to perform subpopulation estimation, and use `if` and `in` only when you need to make restrictions on the estimation sample; see [SVY] **subpopulation estimation**.

- Use `subpop()` option rather than `if` or `in` qualifiers when performing subpopulation estimation

SVY `subpop`

- Let's follow along with Example 11 (*page 21)* from the SVY Documentation (www.stata.com/manuals14/svy.pdf)

Documentation Wrap-up

- Stata scales well into more statistically complex realms
- Unlike open-source languages, StataCorp provides and maintains excellent documentation for the user to learn Stata's specialized commands or check how statistical methods have been implemented
- With each new release, Stata adds functionality for more advanced types of analysis: the documentation is where to check out these developments

Reshaping Data: Long / Wide

Repeated Measures

- In this workshop, we will be talking about reshaping data from long to wide forms (and back again)

- These forms of data occur most often for repeated measures of a variable (like blood pressure over 5 visits to the doctor, or temperature over twelve months)

- The need to do this usually occurs because it is far easier to record and enter data in a **wide format** (with the visits or months across the top of the page), but it is often easier to analyze and visualize data in *long format*

Wide Data

- An example of a wide data set could be this one, comparing my temperature with that of a lizard

Name	Temp1	Temp2	Temp3	Temp4	Temp5	Temp6
Cale	98.4	98.7	98.6	98.3	98.5	98.6
Lizard	69	75	84	92	86	79

- This is a totally reasonable way to record/enter these temperatures in a spreadsheet, but as soon as we consider how to graph or analyze them…

can we **summarize**? can we **graph twoway line**?

Long Data

- A much friendlier data format for commands is the long format
- It may seem like there is a lot of repetition, but now each observation represents the unit of analysis we care about — an animal/time/ temperature combination

Name	Time	Temp
Cale	1	98.4
Cale	2	98.7
Cale	3	98.6
Cale	4	98.3
Cale	5	98.5
Cale	6	98.6
Lizard	1	69
Lizard	2	75
Lizard	3	84
Lizard	4	92
Lizard	5	86
Lizard	6	79

Reshaping W/L

- In Stata, the command **`reshape`** transforms wide data to long data and back again:

long

i	*j*	*stub*
1	**1**	4.1
1	**2**	4.5
2	**1**	3.3
2	**2**	3.0

reshape <⟷>

wide

i	*stub***1**	*stub***2**
1	4.1	4.5
2	3.3	3.0

To go from wide to long:

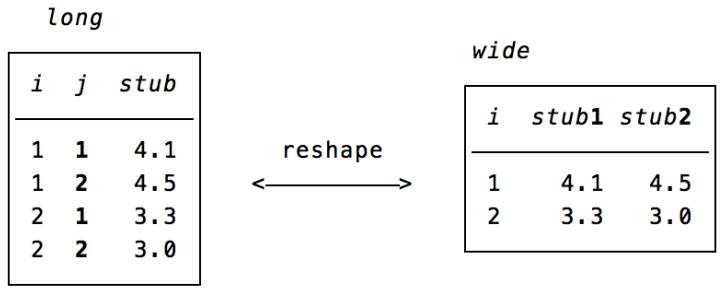
```
reshape long stub, i(i) j(j)
                        \
                         j new variable
```

Reshaping W/L

```
import excel cale_lizard, clear firstrow
```

To go from wide to long:

```
reshape long stub, i(i) j(j)
                           \
                            j new variable
```




```
reshape long Temp, i(Name) j(Time)
```

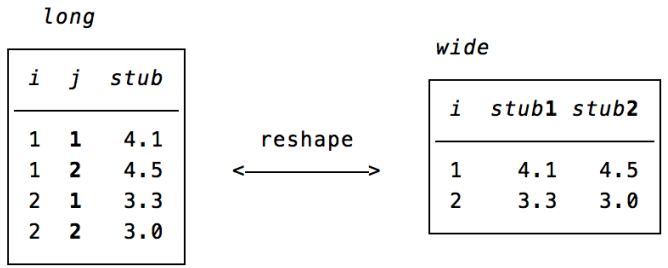
- `stub` is the repeated measure, `i` is the identifier, and `j` is the new variable we wish to create to identify the time points we peeled off of `stub`

Reshaping L/W

To go from long to wide:

j existing variable

reshape wide *stub*, **i**(*i*) **j**(*j*)




```
reshape wide Temp, i(Name) j(Time)
```

- `stub` is the repeated measure, `i` is the identifier, and j the variable in our long dataset that indicates the repeated time or date

Switching W/L L/W

- Once we have defined the reshape `stub`, `i`, and `j` parameters for a dataset with **`reshape`**, we can go back and forth easily:

```
reshape wide
```

```
reshape long
```

Exercises (1)

1. Wide to Long

   A. Load the reshape1 dataset: `webuse reshape1`

   B. Take a look at this dataset in the Data Browser

   C. Reshape this data from wide format to long format

Exercises (2)

1. Long to Wide

   A. Load the bplong dataset: **webuse** bplong

   B. Take a look at the data in the browser

   C. Transform this data from long to wide