# 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!