Loops: Essential tools for repetitive tasks

Gabriela Ortiz

StataCorp LLC

January 18, 2023

Fitting many models

- Suppose you need to perform the same task for groups of observations
- You could repeat the command for each group:

```
. logistic heartatk i.diabetes i.sex if agegrp==1
. logistic heartatk i.diabetes i.sex if agegrp==2
. logistic heartatk i.diabetes i.sex if agegrp==3
. logistic heartatk i.diabetes i.sex if agegrp==4
. logistic heartatk i.diabetes i.sex if agegrp==5
. logistic heartatk i.diabetes i.sex if agegrp==6
```

Write a loop

- Or, you could save some time with a loop:

```
forvalues g = 1/6 {
    logistic heartatk i.diabetes i.sex if agegrp==`g'
}
```

Performing multiple tests

- Now suppose you need to perform the same task for multiple variables
- You could repeat the command for each variable:

```
. ttest tcresult, by(sex) unequal
. ttest tgresult, by(sex) unequal
. ttest hdresult, by(sex) unequal
. ttest bpresult, by(sex) unequal
```

Write a loop

- This can also be done more quickly with a loop:

```
foreach var of varlist *result {
    ttest `var', by(sex) unequal
}
```

Goals

- Learn how to
  - Use macros
  - Loop over values
  - Loop over variables
  - Use tracing to debug your loops
  - Issue code conditional on an expression
  - Write loops that will run despite any errors

Why we need macros

- To loop over variables, we need
    - A list with the variables we'll be working with
    - A way to move through the items in the list, in order to issue the command separately for each variable

Why we need macros

- To loop over variables, we need
  - A list with the variables we'll be working with
  - A way to move through the items in the list, in order to issue the command separately for each variable
- To loop over values, we need
  - A list with the values we'll be working with
  - An easy way to work with a range of values

Why we need macros

- To loop over variables, we need
    - A list with the variables we'll be working with
    - A way to move through the items in the list, in order to issue the command separately for each variable
- To loop over values, we need
    - A list with the values we'll be working with
    - An easy way to work with a range of values
- We can use macros to store our list of values and variables
- We'll use `forvalues` to work with lists of values
- We'll use `foreach` to work with lists of variables and other items

Defining the contents of a local macro

- In Stata, there are global and local macros
    - We'll only work with local macros, but we'll discuss global macros later
- First, let's see how we can store items in a local macro

Defining the contents of a local macro

- We can store a string in a macro:

  `local` *macroname* `"` *string* `"`

  - The `"` marks are optional but often can help readability

Defining the contents of a local macro

- We can store a string in a macro:

  `local` *macroname* `"` *string* `"`

  - The `"` marks are optional but often can help readability

- We can use macros to evaluate expressions:

  `local` *macroname* `=` *exp*

Defining the contents of a local macro

- We can store a string in a macro:

  `local` *macroname* `"` *string* `"`

  - The `"` marks are optional but often can help readability
- We can use macros to evaluate expressions:

  `local` *macroname* `=` *exp*
- And we can use special functions to define a macro

  `local` *macroname* `:` *macro_function*

Expanding local macros

- A local macro is expanded via

  `` `macroname' ``
  - Note the left and right quotes!
    - The left quote is above the *tab* key on US keyboards

Storing text in a macro

```
. local me "Gabriela"

. display "Hello, my name is `me'"
Hello, my name is Gabriela
```

Storing an expression in a macro

```
. local age "20+1"

. display "‘age’"
20+1
```

Using a macro to evaluate an expression

```
. local age2 = 20 + 1

. display "‘age2’"
21
```

Macro function for working with a variable label

```
. sysuse auto, clear
(1978 automobile data)

. describe make

Variable      Storage   Display    Value
    name         type    format    label      Variable label
---------------------------------------------------------------------
make            str18   %-18s                 Make and model

. local makelbl : variable label make

. display "`makelbl'"
Make and model
```

Using macros in our loops

- Now we know how to store contents in a macro and how to refer to the stored contents
- Now we're ready to loop over any items we store in a macro
- We'll start by looping over a series of values

forvalues syntax

- The `forvalues` command allows us to issue the same code for the range of values we specify
- The syntax is as follows:

`forvalues` *lname* = *range* {

Stata commands referring to `` ` ``*lname*`'`

}

forvalues

```
. forvalues g= 1/3 {
  2.      display "g=‘g’"
  3. }
g=1
g=2
g=3
```

- `forvalues` creates a macro called g
- Then it places the first value in the macro g
- Then the second value, and so forth, one at a time

NHANES data

- We have data from the National Health and Nutrition and Examination Survey (NHANES)

```
. webuse nhanes2, clear

. desc agegrp heartatk diabetes highbp

Variable      Storage   Display    Value
    name         type    format    label      Variable label
--------------------------------------------------------------------

agegrp          byte    %8.0g      agegrp     Age group
heartatk        byte    %16.0g     heartlbl   Prior heart attack
diabetes        byte    %12.0g     diabetes   Diabetes status
highbp          byte    %8.0g               * High blood pressure
```

Age groups

- The variable agegrp groups individuals in their 20s, 30s, etc.

```
. codebook agegrp
--------------------------------------------------------------------------------
agegrp
--------------------------------------------------------------------------------

                  Type: Numeric (byte)
                 Label: agegrp

                 Range: [1,6]                        Units: 1
         Unique values: 6                        Missing .: 0/10,351

            Tabulation: Freq.   Numeric  Label
                        2,320         1  20-29
                        1,622         2  30-39
                        1,272         3  40-49
                        1,291         4  50-59
                        2,860         5  60-69
                          986         6  70+
```

Fit regression models for each group

- Suppose we want to fit separate regression models for each age group
- We could fit our models by issuing multiple `logistic` commands:

```
. logistic heartatk i.diabetes i.sex if agegrp==1
. logistic heartatk i.diabetes i.sex if agegrp==2
. logistic heartatk i.diabetes i.sex if agegrp==3
```

- This can quickly get repetitive

Loop over values: forvalues

- We can do this more quickly with `forvalues`
- We'll skip the first two age groups because some variables get omitted in those models

```
. forvalues g= 3/6 {
  2.      logistic heartatk i.diabetes i.highbp i.sex if agegrp==‘g’
  3. }
```

(*output omitted*)

Loop over values: forvalues

`forvalues g = 3/6 {`

- `forvalues` will create a macro called g and store the first value in there:

  `logistic heartatk ... if agegrp==3`

Loop over values: forvalues

`forvalues g = 3/6 {`

- `forvalues` will create a macro called g and store the first value in there:

  `logistic heartatk ... if agegrp==3`

- Then it places next value in the macro g:

  `logistic heartatk ... if agegrp==4`

Loop over values: forvalues

`forvalues g = 3/6 {`

- `forvalues` creates a macro called g and stores the first value in there:

  `logistic heartatk ... if agegrp==3`

- Then it places next value in the macro g:

  `logistic heartatk ... if agegrp==4`

- Then, the next one:

  `logistic heartatk ... if agegrp==5`

Loop over values: forvalues

`forvalues g = 3/6 {`

- `forvalues` creates a macro called g and stores the first value in there:

  `logistic heartatk ... if agegrp==3`
- Then it places next value in the macro g:

  `logistic heartatk ... if agegrp==4`
- Then, the next one:

  `logistic heartatk ... if agegrp==5`
- It cycles through the values until it gets to the last value, 6:

  `logistic heartatk ... if agegrp==6`
- We'll run `forvalues.do` to see the output

Issuing several commands in the loop

- If we want to perform any other computations with this age group, we can simply add code to our loop

```
forvalues g = 3/6 {
    logistic heartatk i.diabetes i.sex if agegrp==`g'
    margins r.(diabetes highbp sex) if e(sample), post
    etable, append
}
```

Issuing several commands in the loop

- If we want to perform any other computations with this age group, we can simply add code to our loop

```
forvalues g = 3/6 {
    logistic heartatk i.diabetes i.sex if agegrp==`g'
    margins r.(diabetes highbp sex) if e(sample), post
    etable, append
}
```

- After we fit the model, we estimate contrasts using the estimation sample
- Then, we create a table with the results from `margins`
  - In each run of the loop, we are appending results to the existing table

Adding labels to our output

- We can add labels to distinguish the output from each model

```
forvalues g = 3/6 {
    local agelbl :  label agegrp `g'
    display as result "Age group=`agelbl'"
    logistic heartatk i.diabetes i.sex if agegrp==`g'
    margins r.(diabetes highbp sex) if e(sample), post
    etable, append
}
```

- The local macro `agelbl` will contain the label corresponding to the value in g
- We then display this label `as result`, meaning it will be black and bold

Inline use of macro functions

- We could be more efficient when using a *macro_function*
- Instead of typing this:

```
local agelbl :  label agegrp `g'
display as result "Age group=`agelbl'"
```

Inline use of macro functions

- We could be more efficient when using a *macro_function*
- Instead of typing this:

```
local agelbl :  label agegrp `g'
display as result "Age group=`agelbl'"
```

- We can simply type the following:

```
display as result "Age group=`:label agegrp `g''"
```

Inline use of macro functions

- We could be more efficient when using a *macro_function*
- Instead of typing this:

```
local agelbl :  label agegrp `g'
display as result "Age group=`agelbl'"
```

- We can simply type the following:

```
display as result "Age group=`:label agegrp `g''"
```

- More generally, we type:

`:*macro_function*'

Other ways to work with forvalues

- There are two other ways to work with `forvalues`

```
. * from 10 to 100, in increments of 10
. forvalues vals = 10(10)100 {
  2.      display "Value: ‘vals’"
  3. }
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
Value: 60
Value: 70
Value: 80
Value: 90
Value: 100
```

Other ways to work with forvalues

```
. * from 3 to 12, in increments of 6-3
. forvalues vals = 3 6 to 12 {
  2.      display "Value: ‘vals’"
  3. }
Value: 3
Value: 6
Value: 9
Value: 12
```

Looping over variables

- If you need to perform the same task for multiple variables, you can use `foreach`
- The syntax is as follows:

  `foreach` *lname* `of varlist` *varlist* {

  Stata commands referring to `` ` ``*lname*`'`

  }
- The `of varlist` tells Stata that what follows is a list of variables; this means you can abbreviate variables, specify a range of variables, and use wildcards

Goal

- We would now like to compare the proportion of men and women who have high blood pressure, diabetes, and who have had a heart attack
- First let's see how we can do this for a single variable

Test of proportions

```
. prtest heartatk, by(sex)

Two-sample test of proportions                  Male: Number of obs =     4915
                                              Female: Number of obs =     5434
------------------------------------------------------------------------------
       Group |       Mean   Std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
        Male |   .0646999   .0035089                      .0578227    .0715771
      Female |   .0290762   .0022793                      .0246088    .0335435
-------------+----------------------------------------------------------------
        diff |   .0356237   .0041842                      .0274229    .0438245
             |  under H0:   .0041234     8.64   0.000
------------------------------------------------------------------------------
        diff = prop(Male) - prop(Female)                          z =   8.6394
    H0: diff = 0

    Ha: diff < 0                 Ha: diff != 0                 Ha: diff > 0
 Pr(Z < z) = 1.0000         Pr(|Z| > |z|) = 0.0000          Pr(Z > z) = 0.0000
```

Accessing returned results

```
. return list

scalars:
                 r(N1) =  4915
                 r(N2) =  5434
                 r(P1) =  .0646998982706002
                 r(P2) =  .0290761869709238
             r(P_diff) =  .0356237112996764
                r(se1) =  .0035088558630739
                r(se2) =  .0022792999752738
           r(se_diff0) =  .00412341715350 9
            r(se_diff) =  .0041841699111187
                r(lb1) =  .0578226671520332
                r(ub1) =  .0715771293891672
                r(lb2) =  .0246088411094242
                r(ub2) =  .0335435328324234
            r(lb_diff) =  .0274228889686876
            r(ub_diff) =  .0438245336306652
                  r(z) =  8.639366324932814
                r(p_l) =  1
                  r(p) =  5.65256590335e-18
                r(p_u) =  2.82628295167e-18
```

Loop: test of proportions

```
foreach var of varlist heartatk diabetes highbp {
    prtest `var', by(sex)
    matrix `var'= r(N1), r(P1), r(N2), r(P2), ///
     r(P_diff), r(p)
}
```

- `foreach` creates a macro called `var`, and will cycle through the variable list (`varlist`)
- With this loop, we can perform the test of proportions for each variable and create a matrix with the results

Issuing several commands in the loop

- We just created matrices named `heartatk`, `diabetes`, and `highbp`
- If we combine them, and provide descriptive row and column names, we'll get a nice table:

```
matrix prtest = heartatk \ diabetes \ highbp
matrix colnames prtest = "Males" "Males" ///
 "Females" "Females" "Difference" "p-value"
matrix rownames prtest = "Heart attack" ///
 "Diabetes" "High BP"
```

- Let's see this work in `foreach.do`

foreach: Working with other items

`foreach` will work with any other set of items

- You could simply list the items in the `foreach` command:

  `foreach` *lname* `in` *anylist* {

- Or, you could store items in a local macro first, then loop over the items in *lmacroname*

  `foreach` *lname* `of local` *lmacroname* {

foreach: Working with other items

`foreach` will work with any other set of items

- You could simply list the items in the `foreach` command:

  `foreach` *lname* `in` *anylist* {
- Or, you could store items in a local macro first, then loop over the items in *lmacroname*

  `foreach` *lname* `of local` *lmacroname* {
- You could also use `foreach` with a list of numbers:

  `foreach` *lname* `of numlist` *numlist* {

foreach: Working with other items

`foreach` will work with any other set of items

- You could simply list the items in the `foreach` command:

  `foreach` *lname* `in` *anylist* {

- Or, you could store items in a local macro first, then loop over the items in *lmacroname*

  `foreach` *lname* `of local` *lmacroname* {

- You could also use `foreach` with a list of numbers:

  `foreach` *lname* `of numlist` *numlist* {

  - Compared with `forvalues`, the advantage here is that you don't have to work with evenly spaced values
  - Your *numlist* could be, for example, `1 -2 4`

Survey of young women

- We have information on wages, union membership, and college completion for young women:

```
. use nlswork2, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)

. d wage collgrad union ttl_exp ind_code

Variable      Storage   Display    Value
    name         type    format    label      Variable label
--------------------------------------------------------------------

wage            float   %9.0g
collgrad        byte    %12.0g     grad       College graduate
union           byte    %12.0g     union      Union member
ttl_exp         float   %9.0g                 Total work experience
ind_code        byte    %8.0g                 Industry of employment
```

Fitting multiple models

- We would like to fit a regression model for wages, separately for each industry
- You could repeat the command for each level of industry:

```
. regress wage i.collgrad i.union if industry==1
. regress wage i.collgrad i.union if industry==2
. regress wage i.collgrad i.union if industry==3
. ...
```

- But we don't know how many industries are present in this dataset

Accessing levels with levelsof

- We can use the `levelsof` command to list the levels of a variable, and store them in a named macro

```
. levelsof ind_code, local(industries)
1 2 3 4 5 6 7 8 9 10 11 12
. display "‘industries’"
1 2 3 4 5 6 7 8 9 10 11 12
```

- Now we can just loop over the items in this macro called `industries`

Fitting a regression model for each industry

```
. levelsof ind_code, local(industries)
1 2 3 4 5 6 7 8 9 10 11 12
. foreach i of local industries {
  2.      regress wage i.collgrad i.union ttl_exp if ind_code==‘i’
  3.      estimates store ind‘i’
  4. }
  (output omitted)
```

- In addition to fitting the model for each industry, we're storing the results with a name corresponding to the industry number
- Let's see this work with `foreach.do`

Goal: loop over items and values

- Our next goal is to fit separate regressions for each industry and year
- This can be done by looping over the year for each industry

```
. estimates clear

. levelsof ind_code, local(industries)
1 2 3 4 5 6 7 8 9 10 11 12

. foreach i of local industries {
  2.     forvalues y = 68(1)88 {
  3.         regress wage i.collgrad i.union ttl_exp if ind_code==‘i’ & year==‘y’
  4.         estimates store ind‘i’_‘y’
  5.     }
  6. }
no observations
r(2000);

end of do-file

r(2000);
```

trace: Looking inside the loop

- It's hard to know which industry and year is causing this error
- If only we could see the commands that are being issued

trace: Looking inside the loop

- It's hard to know which industry and year is causing this error
- If only we could see the commands that are being issued
- If we `set trace on`, Stata will show us what is going on behind the scenes
- By default, this will provide too much information and output

trace: Looking inside the loop

- It's hard to know which industry and year is causing this error
- If only we could see the commands that are being issued
- If we `set trace on`, Stata will show us what is going on behind the scenes
- By default, this will provide too much information and output
- So, we use `set tracedepth 1`
    - This means we'll trace the execution of programs we call, like `regress`, but not the programs that `regress` may call
- Let's see this work with `error.do`

Deciphering the output from trace

- Each line of code starts with a marker:
    - Nothing if the line was not executed
    - - if a line was executed
    - = if macros were expanded to show precisely what was done
- The output will easily fill up the `Results` window, so it's best to store it in a log file

Fitting models with a large sample

- There may be other industries and years with this issue
- One solution is to fit the models conditional on a large enough sample
- First, we'll count the number of observations for that year and industry, for which none of our variables are missing

```
. count if ind_code==`i' & year==`y' & ///
  !missing(wage, collgrad, union, ttl_exp)
```

  - The number of observations will be stored in `r(N)`
  - We can then fit the model conditional on `r(N)` exceeding some value
- Let's see this work with `error.do`

Guarantee a smooth run

- Another option is to fit as many models as possible
- For any cases where we don't have enough observations, we'll just ignore the error
- `capture` will suppress (or capture) any errors and output
- Let's see this work with `error2.do`

noisily

- We can combine `capture` with `noisily` to display the output and errors, but continue to run the commands in the loop despite these errors
- Let's see this work with `error2.do`
  - Errors will be displayed, but they won't stop the loop from running to completion

Quietly: suppressing output

- `capture` is great if you expect errors, but you want your code to keep running
    - For example, we knew there wouldn't be enough observations for some of our models
- If you want to suppress output, but you want the loop to stop running when there is an error, you can use the `quietly` prefix
- Let's see this work with `error2.do`

Nesting loops

- We have 12 different industries and about 15 different years
- While we could fit the model for every year available for each industry, let's keep things simple

Nesting loops

- Suppose we’re only interested in industries 4, 6, and 7, and the years 1980 and 1985

```
. estimates clear

. foreach i of numlist 4 6 7 {
  2.      forvalues y = 80(5)85 {
  3.          regress wage i.collgrad i.union ttl_exp if ind_code==‘i’ & year==‘y’
  4.          estimates store ind‘i’_19‘y’
  5.      }
  6. }
```

(*output omitted*)

- We can’t use `forvalues` here, because our values are not evenly spaced. We have an increment of 2 and an increment of 1, but `forvalues` can only work with one increment.

Defining and expanding global macros

- Global macros are filled just like a local macro except that the keyword `global` is used

  `global` *macroname* `"` *string* `"`

  - The `"` marks are optional but often can help readability

Defining and expanding global macros

- Global macros are filled just like a local macro except that the keyword `global` is used

  `global` *macroname* `"` *string* `"`

  - The `"` marks are optional but often can help readability
- A global macro is expanded via

  $*macroname*

Examples

- Just like with local macros, we can store text and evaluate expressions

```
. global names Jane Julie Jenna

. display "$names"
Jane Julie Jenna

. global age = 20 + 1

. display "$age"
21
```

Examples

- We can also use the special macro functions with global macros

```
. global data: dir . files "file*.dta"

. clear

. append using $data, generate(whichfile)

. list make whichfile, sepby(whichfile)

     +------------------------------------+
     | make                     whichfile |
     |------------------------------------|
  1. | AMC Concord     Appended dataset 1 |
  2. | AMC Pacer       Appended dataset 1 |
  3. | AMC Spirit      Appended dataset 1 |
     |------------------------------------|
  4. | Buick Century   Appended dataset 2 |
  5. | Buick Electra   Appended dataset 2 |
  6. | Buick LeSabre   Appended dataset 2 |
     |------------------------------------|
  7. | Buick Opel      Appended dataset 3 |
  8. | Buick Regal     Appended dataset 3 |
  9. | Buick Riviera   Appended dataset 3 |
     +------------------------------------+
```

Function keys

- In fact, you can store text in a macro named after one of the function keys (F5, F6, etc.)

  ```
  global F6 regress
  ```

- Now hit the F6 key on your keyboard. Depending on your keyboard, you may need to simultaneously hit the FN key

Local vs. global macros

- Local macros are especially useful in loops because they only exist where they are defined
  - If you define a local macro in a loop, it only exists while the loop is running
  - If you define a local macro in a do-file, it only exists while the do-file is running
- Global macros are known in every context—they can be defined in one (a)do-file and used in another

Macros in other do-files

- We created a local and a global macro in the file `names.do`

```
. do names

. global first George Jenna Sergio

. local last Johnson Medina Clooney

.
end of do-file

. foreach i of global first {
  2.     display "First name: ‘i’"
  3. }
First name: George
First name: Jenna
First name: Sergio

. foreach i of local last {
  2.     display "Last name: ‘i’"
  3. }
```

Including other do-files

- If you want to work with local macros defined in another file, use the `include` command instead of `do`

```
. include names

. global first George Jenna Sergio

. local last Johnson Medina Clooney

.
. foreach i of local last {
  2.     display "Last name: `i'"
  3. }
Last name: Johnson
Last name: Medina
Last name: Clooney
```

- With `include`, it's as if we copied the code from `names.do` into our current file, so local macros are still defined
- Now we can see the names from the local macro `last`

Summary

- Today we learned
  - the different ways to define a macro
  - how to use `forvalues` to fit separate regression models for groups of our data
  - how to use `foreach` to perform tests for multiple variables
  - how to trace the execution of our code in order to see which observations caused an error
  - how to run a series of commands conditional on an expression
  - how to run our loops quietly and despite any errors
  - how to easily append several datasets

Where to learn more

- You can type `help foreach` or `help forvalues` for a quick reference
    - In the help file, you'll see a link to the PDF documentation, which contains worked examples

Where to learn more

- You can type `help foreach` or `help forvalues` for a quick reference
    - In the help file, you'll see a link to the PDF documentation, which contains worked examples
- Take a look at frequently asked questions (FAQs) about looping:

```
. search looping, faq
```

Where to learn more

- You can type `help foreach` or `help forvalues` for a quick reference
  - In the help file, you'll see a link to the PDF documentation, which contains worked examples
- Take a look at frequently asked questions (FAQs) about looping:

```
. search looping, faq
```

- Sign up for a NetCourse on Stata Programming:

  `http://www.stata.com/netcourse/programming-intro-nc151/`

- Take a look at this book about Stata Programming:

  `http://www.stata.com/bookstore/stata-programming-introduction/`

The end

- Thank you!