Programming

LPO 9951 | Fall 2015

PURPOSE Stata programming will save you time, energy, and sanity. Investing the time now into learning how to program will certainly pay off. It may seem easy enough now to just copy code 10 times if you need to complete an operation 10 times, but force yourself to use your programming skills. By Maymester, you will thank yourself.

Tools you already have

Programming is more than just knowing the most convenient commands to shorten the time you spend on menial tasks. It involves thinking about how the commands you do can be combined to make a more efficient, readable do-file for you and anyone else who will look at it in the future.

The following points are good places to start when you are trying to make your program file more efficient.

- Previous code: You may have already encountered this strategy in the work that you have done thus far for the class. Snippets of code that you have already toiled over can be used again and again. The following tips might come in handy.
  - Save your do-files
  - Label them well
  - Re-use old code, copy-paste
  - Make templates if you use a certain piece of code often
  - Create files to include or do (e.g., "programs" you can immediately run for things like dealing with missing data)
- Programming: When you approach your Stata script as a programmer, you have a different perspective, a certain general approach on how to put these pieces together. The following points are questions you might ask yourself in going through the general process for your program.
  - What is the overall task I am trying to accomplish?
  - How are the variables structured? Which variables go together?
  - What tasks need to be repeated?
  - What procedures may stay the same, though the numerical values may change?

Organizing your do file

As your do files increase in length, you will want some type of organizational structure. A table of contents at the top of the script can be very helpful. You certainly don't have to do it the way the way shown below, but you should have something that makes sense to you and will be clear to others who may read your script.

```
. // TABLE OF CONTENTS
. // 0.0 Set preferences/globals
. // 1.0 Describing
. // 2.0 Scalars
. //   2.1 return
. //   2.2 ereturn
. //   2.3 scalar
. // 3.0 Estimates
. //   3.1 estimates store
. //   3.2 estimates restore
. // 4.0 Shortcuts
. //   4.1 numlists
. //   4.2 varlists
. // 5.0 Macros
. //   5.1 globals
. //   5.2 numerical locals
. //   5.3 varlist locals
. //   5.4 nested locals
. // 6.0 Switches
. // 7.0 Loops
. //   7.1 if / else
. //   7.2 foreach
. //   7.3 forvalues
. //   7.4 while
. // 8.0 Nests
```

File header

Like you've seen in the do files from earlier lectures, it's often useful to place your file preferences at the top of the script. These may include, but aren't limited to, graphics settings and global macros storing directory structurs or url links. If you are only using one dataset for your analysis, this is a good place to load it.

```
.
. clear all                                 // clear memory

. set more off                              // turn off annoying "__more__" featu
> re

. global datadir "../data/"

. use ${datadir}loondata, clear
```

Describing

`bysort`: Used by itself or in combination with `gen` or `egen`, this command also allows you to perform a task on numerous categories of a variable or variables.

For example, we might want to know what the average flock size is by status as a loon. We could use the following code:

```
. sum flock1 if loon == 0
```

```
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        157    82822.28    20149.31      37267     125631

. sum flock1 if loon == 1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        278    78270.19    20389.95      12812     136822
```

A slightly easier bit of code would use `tab` with the `summarize` option:

```
. tab loon, summarize(flock1)

            |      Summary of Size of flock
            |             represented
       Loon |        Mean   Std. Dev.       Freq.
------------+------------------------------------
          0 |    82822.28   20149.306         157
          1 |   78270.187   20389.948         278
------------+------------------------------------
      Total |   79913.126   20397.942         435
```

Still another line of code uses the `bysort` command, which takes the form `bysort <sorting variable>: <command> <variable>`:

```
. bysort loon: sum flock1

-------------------------------------------------------------------------------
-> loon = 0

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        157    82822.28    20149.31      37267     125631

-------------------------------------------------------------------------------
-> loon = 1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        278    78270.19    20389.95      12812     136822
```

We could actually ask for numerous variables summarized in this way.

```
. bysort loon: sum flock1 flock2 flock3

-------------------------------------------------------------------------------
-> loon = 0

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        157    82822.28    20149.31      37267     125631
```

```
      flock2 |        157    84158.22    28303.17      23892     175592
      flock3 |        157    83077.31    34877.37      17005     207132

-------------------------------------------------------------------------------
-> loon = 1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        278    78270.19    20389.95      12812     136822
      flock2 |        278    78014.35    25704.09      13561     160571
      flock3 |        278    76961.31    27805.51      14268     168056
```

Quick Exercise

> Find the average number of feathers in each period by the double condition of being a loon and location of nest.

Scalars

Scalars temporarily save information that you can use later. There are two types of information that are stored in STATA after you run commands. The first is saved as `r` and can be found by using `return list`. Here are some examples:

```
. sum shells1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
     shells1 |        435    114167.8    26180.23      45770     164786

. return list

scalars:
                  r(N) =  435
              r(sum_w) =  435
               r(mean) =  114167.8252873563
                r(Var) =  685404214.9233223
                 r(sd) =  26180.22564691379
                r(min) =  45770
                r(max) =  164786
                r(sum) =  49663004

. di r(mean)
114167.83

. di r(sd)
26180.226
```

The second type of information that is stored is under `e`. These can be found by using `ereturn list`:

```
. mean shells1

Mean estimation                   Number of obs    =        435
```

```
-------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+-----------------------------------------------
     shells1 |   114167.8   1255.246      111700.7    116634.9
-------------------------------------------------------------

. ereturn list

scalars:
               e(df_r) =  434
             e(N_over) =  1
                  e(N) =  435
               e(k_eq) =  1
               e(rank) =  1

macros:
            e(cmdline) : "mean shells1"
                e(cmd) : "mean"
                e(vce) : "analytic"
              e(title) : "Mean estimation"
          e(estat_cmd) : "estat_vce_only"
            e(varlist) : "shells1"
       e(marginsnotok) : "_ALL"
         e(properties) : "b V"
             e(depvar) : "Mean"

matrices:
                  e(b) :  1 x 1
                  e(V) :  1 x 1
                 e(_N) :  1 x 1
              e(error) :  1 x 1

functions:
             e(sample)

. di e(N)
435
```

Keep in mind, however, that each time you run an expression, your previously stored information in both `return` and `ereturn` are overwritten. For example, if you run a `sum` command on one variable, you might have an `r(mean) = 100`. However, the next time you run `sum` on a new variable, the `r(mean)` will be overwritten, so you need to be aware of which variable you are using.

So how do we store this information into memory for future use without fear of it being overwritten? There are multiple ways to do so. One easy way includes naming and storing your own scalar using the `scalar` command, which takes the form of `scalar <name> = <value>`.

You can name a scalar whatever you want and assign it a value. Let's do this for the mean of total number of shells in the first period and show how this preserves the value despite the fact that we run another mean on feathers.

```
. sum shells1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
```

```
-------------+---------------------------------------------------------
     shells1 |        435    114167.8    26180.23      45770     164786

. scalar mean_shells1 = r(mean)

. sum feathers1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
   feathers1 |        435    121527.2    19807.43      65728     181036

. di mean_shells1
114167.83
```

Quick Exercise

> Use a scalar to calculate the average number of shells across all three periods.

Estimates

Similar to scalars and returns, estimates store multiple values. This will be especially useful when we get into regressions next semester. For now, let's just use estimates to store information we've learned from the `mean` command.

```
. mean ideas1

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
      ideas1 |   11.51724    .224046      11.07689    11.95759
--------------------------------------------------------------

. estimates store m_ideas1

.
```

Now we'll use estimates restore and estimates replay to bring up previous information that we've stored.

```
. mean eggs1

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
       eggs1 |   5.878161   .0935593      5.694275    6.062047
--------------------------------------------------------------

. estimates store m_eggs1
```

```
. estimates restore m_ideas1
(results m_ideas1 are active now)

. estimates replay

-------------------------------------------------------------------------------
Model m_ideas1
-------------------------------------------------------------------------------

Mean estimation                    Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
      ideas1 |   11.51724    .224046      11.07689    11.95759
--------------------------------------------------------------

. estimates clear
```

Shortcuts: Numlists and Varlists

Numlists and varlists can make your life much easier by streamlining your code. Here are some examples of numlists. Notice how we sort the data using both `sort` and `gsort`. Also notice the `-` sign used in the second `list` command (`-10/l`) and with `gsort -ideas1`. In the first case, the sign tells Stata to `list` the last 10 observations ('starting at the end, go back 10). In the second case, Stata understands that we want to sort our data based on the values in `ideas1`, but instead of sorting from smallest to largest, as is the default, we instead want descending values.

```
. sort shells1

. list id shells1 loon upper in 1/10

     +-------------------------------+
     |  id   shells1   loon   upper |
     |-------------------------------|
  1. | 371     45770      0       0 |
  2. | 356     48682      0       0 |
  3. | 357     52093      0       0 |
  4. | 350     55419      0       0 |
  5. | 309     58637      0       1 |
     |-------------------------------|
  6. | 396     58916      0       0 |
  7. | 321     59106      0       1 |
  8. | 401     59578      0       0 |
  9. | 335     60526      0       0 |
 10. | 326     60763      0       1 |
     +-------------------------------+

. list id shells1 loon upper in -10/l

     +-------------------------------+
     |  id   shells1   loon   upper |
     |-------------------------------|
```

```
426. | 260    153530      1       0 |
427. | 278    153992      1       0 |
428. |  21    154898      1       1 |
429. | 251    155580      1       0 |
430. |  87    155591      1       0 |
     |-------------------------------|
431. |  45    156288      1       1 |
432. |  61    157157      1       0 |
433. | 115    160948      1       0 |
434. | 164    162947      1       0 |
435. |   5    164786      1       1 |
     +-------------------------------+

. gsort -ideas1

. list id ideas1 loon upper in 1/10

     +------------------------------+
     |  id   ideas1   loon   upper |
     |------------------------------|
  1. | 421       25      0       0 |
  2. | 347       24      0       0 |
  3. | 405       23      0       0 |
  4. | 291       23      0       1 |
  5. | 368       22      0       0 |
     |------------------------------|
  6. | 384       21      0       0 |
  7. | 412       21      0       0 |
  8. | 320       21      0       1 |
  9. | 318       21      0       1 |
 10. | 363       21      0       0 |
     +------------------------------+

. list id ideas1 loon upper in -10/l

     +------------------------------+
     |  id   ideas1   loon   upper |
     |------------------------------|
426. | 274        3      1       0 |
427. | 219        3      1       0 |
428. |  60        3      1       0 |
429. | 100        2      1       0 |
430. | 138        2      1       0 |
     |------------------------------|
431. | 158        2      1       0 |
432. | 263        1      1       0 |
433. | 103        1      1       0 |
434. | 241        0      1       0 |
435. | 216        0      1       0 |
     +------------------------------+

.
```

And here's how we might use varlists. Notice how instead of listing every variable, we can list the starting and final column with a - between. Using this format requires that we know the order of the variables in our dataset. We can also use wildcards such as *. As you can see, Stata returns every variable that starts with `flock`. Keep this feature in mind as you name your variables.

```
. sum shells1-flock3, sep(3)

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
     shells1 |        435    114167.8    26180.23      45770     164786
     shells2 |        435    114560.5    36139.61      28835     236489
     shells3 |        435    113311.4    42467.27      27373     265026
-------------+---------------------------------------------------------
   feathers1 |        435    121527.2    19807.43      65728     181036
   feathers2 |        435      122083    33439.98      47380     265891
   feathers3 |        435    120652.8    40768.82      34113     278089
-------------+---------------------------------------------------------
      flock1 |        435    79913.13    20397.94      12812     136822
      flock2 |        435    80231.79    26802.23      13561     175592
      flock3 |        435    79168.69    30648.81      14268     207132

. sum flock*

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      flock1 |        435    79913.13    20397.94      12812     136822
      flock2 |        435    80231.79    26802.23      13561     175592
      flock3 |        435    79168.69    30648.81      14268     207132
```

Macros

Globals

We've already been using global macros throughout this course, but it never hurts to reiterate. Global macros allow you to store many types of information that will persist throughout a Stata session. We've been using them to store relative directory links, but they can also store numerical values and even commands.

Be careful when using global macros. It is easy over the course of a long Stata session to forget what's hanging around in the memory. To see which globals (or any macros you have stored for that matter), you can use the `macro list` command. To drop macros you no longer need, a generally good policy, use the `macro drop <macro names>` command.

```
. global repstr "Long string I will use a lot and don't want to retype"

. macro list
repstr:         Long string I will use a lot and don't want to retype
S_2:            1
S_1:            ideas1
S_FNDATE:       17 Sep 2014 09:25
S_FN:           ../data/loondata.dta
datadir:        ../data/
F1:             help advice;
F2:             describe;
```

```
F7:             save
F8:             use
S_ADO:          BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
S_StataSE:      SE
S_CONSOLE:      console
S_FLAVOR:       Intercooled
S_OS:           Unix
S_MACH:         Macintosh (Intel 64-bit)
S_level:        95
S_MODE:         batch

. di "$repstr"
Long string I will use a lot and don't want to retype

. macro drop repstr

. macro list
S_2:            1
S_1:            ideas1
S_FNDATE:       17 Sep 2014 09:25
S_FN:           ../data/loondata.dta
datadir:        ../data/
F1:             help advice;
F2:             describe;
F7:             save
F8:             use
S_ADO:          BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
S_StataSE:      SE
S_CONSOLE:      console
S_FLAVOR:       Intercooled
S_OS:           Unix
S_MACH:         Macintosh (Intel 64-bit)
S_level:        95
S_MODE:         batch
```

Locals

Locals are a way of storing information that you would not really want to store in a new variable or even scalar. Some of the other automatic results that are given after running some descriptive or estimation command are locals. Locals can store a single value or a list of values, but only for length of time that the script is actively running. This is unlike global macros, which persist throughout a Stata session (until you quit the program or purposefully drop them). Once script has exited, all information stored in locals is lost. There is a very particular way data in locals are stored and recalled.

Here are some of the different ways locals are used with numbers:

```
. local i 1

. di `i'
1

. local j = 2

. di `j'
```

```
2

. local k = `i'+`j'

. di `k'
3

. sum ideas1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas1 |        435    11.51724     4.67285          0         25

. local mean_ideas1 = r(mean)

. di `mean_ideas1'
11.517241

.
```

Locals can also store strings (such as variable names):

```
. local contributions ideas1 ideas2 ideas3 eggs1 eggs2 eggs3

. sum `contributions', sep(3)

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas1 |        435    11.51724     4.67285          0         25
      ideas2 |        435    11.56322     4.79271          0         25
      ideas3 |        435    11.42529     5.11263         -2         26
-------------+---------------------------------------------------------
       eggs1 |        435    5.878161    1.951333          1         14
       eggs2 |        435    5.924138    2.155529          1         16
       eggs3 |        435    5.786207     2.95867         -4         16
```

Even better, locals can be nested, that is, a local can hold other locals:

```
. local whoareyou loon upper seasons

. local wholeshebang `contributions' `whoareyou'

. sum `wholeshebang', sep(3)

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas1 |        435    11.51724     4.67285          0         25
      ideas2 |        435    11.56322     4.79271          0         25
      ideas3 |        435    11.42529     5.11263         -2         26
-------------+---------------------------------------------------------
       eggs1 |        435    5.878161    1.951333          1         14
       eggs2 |        435    5.924138    2.155529          1         16
       eggs3 |        435    5.786207     2.95867         -4         16
```

```
-------------+---------------------------------------------------------
        loon |        435    .6390805    .4808202          0          1
       upper |        435    .2298851    .4212432          0          1
     seasons |        435    8.036782     1.71696          5         12
```

NB: The quotation marks for locals can be tricky. If you are having trouble getting your locals to do exactly what you want, check to make sure you are using the correct quotes. The left quote `, or backtick, is distinct from the normal single quotation mark, '.

Switches

Now that weve learned how to create locals, lets use them to create switches. Switches are impor- tant because you can use them to turn on and off portions of your code. For example, you can use them to delegate whether you want to input the full dataset or your most recently save data. You could also use them to determine whether you want to turn graphs on or off. They can be very useful.

As an example, below is a switch for turning on or off graphs:

```
. local graphs = 0

. if `graphs' == 1 {
.     scatter shells1 feathers1 if loon == 1
. }
```

As you can see, there is no special switch command. Instead, we create a local called `graphs` that is set to either 0 or 1. Next comes an `if` statement that says if the local is equal to 1, then run the graph. If not, then don't. It's good practice to place your switches in the top of your do file so that you don't have to hunt for them.

One trick to keep track of `if` statements (and loops as you will see below) is to indent code that is within the loop and align the start of the loop with the end of the loop. It's also good practice with long loops to use a comment at the closing brace to label the loop (very helpful when you have many loops in your file).

Loops

Loops are used when you are performing one task on a variable or group of variables and `bysort` and `egen` cannot meet your needs. It is useful to think about how the procedures you are running can be grouped together and how the same structure can be applied to multiple cases. Stata loops have a few different structures:

- `if` (`if` / `else`)
- `foreach`
- `forvalues`
- `while`

`if` / `else`

To start, let's just use our switch structure to specify an alternative action if the switch condition is not met.

```
. local switch = 0

.
. if `switch' == 0 {
.     sum loon if upper == 0

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
        loon |        335    .6925373     .462133          0          1
. }

. else {
.     sum loon if upper == 1
. }

.
```

If `switch == 1` then the first command will run; in all other cases, the second command will run.

`foreach`

Another type of loop uses the `foreach` command. Take a look at the help file for `foreach` statements. As you can see, there are a variety of different ways to use the `foreach` command. Here are some examples:

```
. foreach var of varlist shells1-feathers3 {
  2.     mean `var'
  3. }

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
     shells1 |   114167.8   1255.246      111700.7    116634.9
--------------------------------------------------------------

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
     shells2 |   114560.5   1732.762      111154.8    117966.1
--------------------------------------------------------------

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
     shells3 |   113311.4    2036.15      109309.4    117313.3
--------------------------------------------------------------

Mean estimation                   Number of obs   =        435
```

```
--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
   feathers1 |   121527.2   949.6935      119660.7    123393.8
--------------------------------------------------------------

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
   feathers2 |     122083   1603.324      118931.7    125234.2
--------------------------------------------------------------

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
   feathers3 |   120652.8   1954.715      116810.9    124494.6
--------------------------------------------------------------

. local memberships loon upper

. foreach var of local memberships {
  2.     mean `var'
  3. }

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
        loon |   .6390805   .0230536      .5937699     .684391
--------------------------------------------------------------

Mean estimation                   Number of obs   =        435

--------------------------------------------------------------
             |       Mean   Std. Err.     [95% Conf. Interval]
-------------+------------------------------------------------
       upper |   .2298851   .0201971      .1901888    .2695813
--------------------------------------------------------------

. foreach val in id {
  2.     list `val' if eggs1 < 3
  3. }

     +-----+
     |  id |
     |-----|
 67. | 394 |
 92. | 386 |
```

```
118. | 203 |
120. |  18 |
132. | 345 |
     |-----|
175. | 308 |
267. | 121 |
330. | 135 |
347. | 326 |
398. | 259 |
     +-----+
```

Quick Exercise

Rescale each `shells*` variable so it is in 1000s of shells.

`forvalues`

Another loop command that is quite useful is called `forvalues`. The `forvalues` loop uses a counter within a loop and repeats the loop until you hit the maximum specified value. Here are some examples; notice the different ways to count:

```
. forvalues x = 1/10 {
  2.     di `x'
  3. }
1
2
3
4
5
6
7
8
9
10

. forvalues y = 2(2)10 {
  2.     di `y'
  3. }
2
4
6
8
10

. forvalues z = 2 4 to 10 {
  2.     di `z'
  3. }
2
4
6
8
10
```

Quick Exercise

Use `forvalues` to create means for days in nest.

`while`

Finally, `while` loops are another way to loop using numbers. They are similar to `forvalue`s loops in Stata, but require a counter. Though the two are generally interchangeable, while loops are technically about waiting to fulfill a condition. Therefore, they can be used in more ways than `forvalues` loops. Keep in mind, however, that if you set a condition that will never be fulfilled, your `while` loop will run forever (or until your computer crashes or the network administrator, if you are running code through a network, kills the process and sends you a mean email).

```
. local i = 1

. while `i' < 11 {
  2.     di `i'
  3.     local i = `i' + 1
  4. }
1
2
3
4
5
6
7
8
9
10
```

Nests

It is also possible to nest loops within loops. When you do this, the outer loop runs until it hits an inner loop. Then it evaluates the inner loop until the inner loop is finished. Then it will continue with the outer loop. If the inner loops statement uses an `if` statement, Stata will only evaluate it if the condition is met (evaluates to true). This can get very complicated very quickly, so you need to know where you are in the code. This is why it is smart to indent all commands within a loop to the level of the loop.

```
. local thoughts ideas1 ideas2 ideas3

.
. forvalues i = 1/2 {
  2.     if `i' == 1 {
  3.         local type "Not a loon"
  4.     }
  5.     if `i' == 2 {
  6.         local type "Loon"
  7.     }
  8.     foreach var of local thoughts {
  9.         di `i'
 10.         di "`type'"
 11.         sum `var' if loon == `i' - 1
```

```
 12.     }
 13. }
1
Not a loon

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas1 |        157    15.95541    3.386141          7         25
1
Not a loon

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas2 |        157    16.04459    3.429405          7         25
1
Not a loon

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas3 |        157    15.80255    3.984649          7         26
2
Loon

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas1 |        278    9.010791    3.207036          0         18
2
Loon

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas2 |        278    9.032374    3.399251          0         19
2
Loon

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      ideas3 |        278    8.953237    3.875496         -2         22
```

Quick Exercise

> Using the following local set up, create a nested loop that uses either `forvalues` or a `while` loop to summarize all the listed variables by period, conditional on the status as a loon.

```
local all1 shells1 feathers1 flock1 ideas1 eggs1 nest1
local all2 shells2 feathers2 flock2 ideas2 eggs2 nest2
local all3 shells3 feathers3 flock3 ideas3 eggs3 nest3
```

Sectioning your do-file (templates)

You will go through the same general procedures every time you work with quantitative data. The structure of this class is a good guide for you to create your own template do-file that you can pull up every time you start a new research project. Sections might include the following:

- Setting up Stata (most of what the do files we have been using for class already have)
- Setting up globals/locals/file preferences
- Pulling in the data you will use
- Data cleaning/validation
- Taking account of the survey design
- Descriptive statistics
- Regression model(s)
- Recording output

Init: 16 August 2015; Updated: 18 August 2015