BASIC STATA PROGRAMMING

Macros, loops, and user-defined programs

Hendrik van Broekhuizen

Research on Socio-Economic Policy (ReSEP), Department of Economics, Stellenbosch University

24-26 October 2014

RESEP
Research on Socio-Economic Policy

Disclaimer

These slides are meant to serve as a basic introduction to Stata programming and offers an overview of *macros*, *loops*, and *user-defined programs*.

All notes, examples, and applications are my own work, but in areas draw heavily from a number of excellent sources, most of which are listed at the end of this presentation. Users are strongly encouraged to consult these resources for a more complete treatment of the concepts introduced and discussed here.

Any omissions and/or errors in this presentation remain mine alone.

Basic Stata vs Advanced Stata I

Towards more power, flexibility, and efficiency

- Stata is very powerful and has a formidable repertoire of canned commands
- But many of its most useful features, functions, and commands for dealing with repetitive or iterative tasks tend to be underutilized because users are either
    - not aware that they exist,
    - not aware of what they can do, and/or
    - intimidated by their steeper than average learning curve
- This means that users often struggle needlessly to do what they want in Stata and tend to do (much) more work than is necessary

Basic Stata vs Advanced Stata II

Exploiting Stata's true power, flexibility, and efficiency: Macros, loops, and programs

- Few users ever create routines/programs intended for sharing with other Stata users
- However, the vast majority of Stata users will benefit massively from employing macros, loops, and programs in their own do-files
  - More often than not, these commands, functions, and features bring significant gains in terms of power, flexibility, and efficiency
  - Learning how to use them well requires some exposure to their syntax and uses, practice, and a healthy dose of patience
  - But, the advantages of using them far outweigh the initial investment required to learn how to use them

Objective

An introduction to basic Stata programming

- **Objective:** provide some exposure to macros, macro functions, loops, and programs, and show
  - why should they be used,
  - how are they used, and
  - what can they be used to do
- The aim throughout is to
  - understand the rationale behind using macros, loops, and user-defined programs, and
  - understand the links between macros, loops, and user-defined programs and
- To this end, many illustrative examples and applications will be used
  - Not all (or any) of the examples may be pertinent to you or even of particularly great practical use
  - but they are intended to be illustrate the rationale behind and the mechanics of the underlying function or command

Data

NIDS2008_sample.dta

- The example data used throughout is a non-representative subset of data from Wave I of the National Income Dynamics Study (*NIDS2008_sample.dta*) [1]
- The data is intended for illustrative purposes only and is not suitable for real-world analysis

[1] NIDS website

Outline

Structure of the presentation

- **Section 1: Introduction**
- **Section 2: Basic programming**
  - Introduces macros, macro functions, loops, branching, and user-defined programs
- **Section 3: Developing a program**
  - Provides a real world example of how a program may be developed, starting from the reason why the program is needed, to its conceptual and practical evolution from basic code to a self-contained *ado-file*
- **Section 4: Examples & Applications**
  - Provides a set of further examples and applications for macros, loops, and programs

Section 2: Basic Programming

Stata Macros I

What are they?

- Macros in Stata function like variables in most other programming languages
  - i.e. as an *alias* with both a *name* and a *value* that, when *dereferenced*, returns its value (Baum, 2005, p. 4)
- But they are defined and called slightly differently than in other languages
- e.g. Python, Matlab, Mata, etc.

```
x = 2
2 + x
>>> 4
```

- e.g. Stata

```
local x = 2
display 2 + `x'
4
```

- Their names and contents are largely arbitrary (STS)[2]

[2] See StataCorp (2013c, pp. 208 - 219)

Stata Macros II

Two types: `local` and `globals`

- `locals` and `globals` differ in terms of their scope and how they are called once defined
- A `local` has a “local” scope
  - it is only valid within the do-file, loop, or program within which it was defined
- A `global` has a “global” scope
  - within a particular instance of Stata, it is always valid, regardless of where it was defined
- Defining and calling a `local`:

```
local x = 2
display 2 + `x'
4
```

- Defining and calling a `global`:

```
global x = 2
display 2 + $x
4
```

- ROT: Use `locals` whenever `globals` can be avoided

An aside: Defining macros, and the equals sign (=)

Aliasing vs evaluating expressions when defining macros

- Macros can be defined with or without the equal sign (=)
  - With `(=)` : remainder of expression is *evaluated*
  - Without `(=)` : remainder of expression is *aliased*

- Immediate evaluation:

```
local word = hello
>>> hello not found
>>> r(111);
```

- Deferred evaluation (aliasing):

```
local word hello
di `word'
>>> hello not found
>>> r(111);
```

- It is generally preferable to define macros without equals sign unless explicitly required

Example A: Aliasing vs evaluating expressions

Stata Snippet: A case where immediate evaluation is preferable to deferred evaluation

Task: calculate 1 + 1 by creating a local with an initial value of 1, incrementing the local by 1, and then displaying the answer in/as a string.

```
* 1. Deferred evaluation (incorrect)
local i = 1          // Store value of 1 in local i
local i `i' + 1      // Increment `i' by 1, but defer evaluation of expression
di "Answer: `i'"     // Display string and evaluate local within
>>> Answer: 1 + 1

* 2. Immediate evaluation (correct)
local i = 1          // Store value of 1 in local i
local i = `i' + 1    // Increment `i' by 1 and evaluate expression
di "Answer: `i'"     // Display string containing previously evaluated local
>>> Answer: 2
```

Stata Macros III

Three special types of `local`: `tempvar`, `tempname`, and `tempfile`

- Three special `local` cases that are particularly useful when programming
    - `tempvar` defines local(s) that may be used to create and refer to temporary variables in the data
    - `tempname` defines local(s) that may be used to create and refer to temporary scalars or matrices in the data
    - `tempfile` defines locals that may be used to create and refer to temporary files
- `tempvar` and `tempname` are used to ensure efficient and clean programming
- `tempfile` is particularly useful when calculations require *'destructive'* data manipulations
    - e.g. when collapsing data or creating sub-samples
- More on this later...

Stata Macros IV

What are they for?

- Macros are most useful/powerful in the context of loops and programs
  - i.e. for executing iterative or repetitive procedures
- But they also have other uses
  - making complex/busy code more readable
  - shorthand for referencing/obtaining data attributes or stored results (macro extended functions)
  - *assigning keyboard shortcuts (Windows only)*

Example 1: Making complex/busy code more readable I

Macros for control variables and macros for conditional statements

- Depending on the complexity of what you are trying to accomplish, command lines in Stata can get quite long
- This can make *do-files* too "busy" and difficult to read
- Suppose, for example, you wanted to
  1. run a series of regressions, each of which has a slightly different specification, but the same set of control variables *or*
  2. run a series of commands, each of which must be restricted to using the same underlying sample via a fairly complex `if` statement
- In both instances, you can use macros to make the code "cleaner" and easier to alter than would otherwise be the case

Example 1: Making complex/busy code more readable II

Stata Snippet: Using a `local` as shorthand for a list of variables

Task: estimate (OLS) association between education and ln(earnings), controlling for age, age^2, race, the interaction between being female and being married, household size, and province

```
* 1. Writing out the full set of variables each time
reg learnings educ ///
    c.age##c.age i.race i.female##i.married hhsize i.province
reg learnings c.educ##c.educ ///
    c.age##c.age i.race i.female##i.married hhsize i.province
reg lhhincome c.educ##c.educ father_educ mother_educ ///
    c.age##c.age i.race i.female##i.married hhsize i.province

* 2. Using a local as shorthand for the control variables
local controlvars c.age##c.age i.race i.female##i.married hhsize i.province
reg lwages educ `controlvars'
reg learnings c.educ##c.educ `controlvars'
reg lhhincome c.educ##c.educ father_educ mother_educ `controlvars'
```

Example 2: Making complex/busy code more readable I

Stata Snippet: Using a `local` as shorthand for an `if` condition

Task: `summarize` monthly earnings, `tabulate` education, and graph the distribution of ln(hhincome) for females between the ages of 15 and 64 who are either married or living with a partner

```
* 1. Writing out the same if condition each time
sum earnings if inrange(age,15,64) & female & inlist(marital,2,3)
tab educ if inrange(age,15,64) & female & inlist(marital,2,3)
kdensity lhhincome if inrange(age,15,64) & female & inlist(marital,2,3)

* 2. Using a local as shorthand for the if condition
local condition inrange(age,15,64) & female & inlist(marital,2,3)
sum earnings if `condition'
tab educ if `condition'
kdensity lhhincome if `condition'
```

Macro extended functions[4]

Accessing OS parameters, data attributes, stored results, and much more

- Stata's macro extended functions are incredibly useful and powerful tools for, among other things, accessing OS parameters, data attributes, and estimation results
- They can be used to...
  - Access data/variable attributes
    - storage type, display format, variable label, value label name, label(s) associated with numeric value(s), etc.[3]
  - Access OS parameters
    - current/specified directory and/or sub-directories, specific types of files in current/specified directory and/or subdirectories, etc
  - Access stored estimation results
    - anything stored in `e()`, `r()`, or `s()`
  - Etc., etc., etc.

[3] Stata's `describe` command, for example, is based on these functions

[4] See StataCorp (2013*b*, pp. 261 - 276)

Example 1: Macro extended functions & data attributes

Stata Snippet: Accessing variable attributes

Task: access the storage type, variable label, value label name, and the value label corresponding to a numeric value of 1 for the variable 'pcode' and then display these attributes

```
* 1. Access attributes of variable pcode
local vartype: type pcode                  // get variable storage type
local varlab : variable label pcode        // get variable label
local valuelabname : value label pcode     // get value label name
local valuelab1 : label (pcode) 1          // get label for value of 1

* 2. Display 'captured' attribute of pcode
di "`vartype'"                             // display local "vartype"
di "`varlabel'"                            // display local "varlabel"
di "`valuelabname"                         // display local "valuelabname"
di "`valuelab1'"                           // display local "valuelab1"

* 3. Access and display attributes of pcode in one step
di "`: type pcode'"                        // display variable storage type
di "`: variable label pcode'"              // display variable label
di "`: value label pcode'"                 // display value label name
di "`: label (pcode) 1'"                   // display label for value of 1
```

Example 2: Macro extended functions & macro lists

Stata Snippet: Accessing elements from a list

Task: Get the third element in the list '*2 guys 1 girl 1 pizza place*'

```
* 1. Mixed list with no quotes
local phrase 2 guys 1 girl 1 pizza place
di "`:word 2 of `phrase''"
>>> Answer: guys

* 2. Mixed list with inner quotes
local phrase "2 guys" "1 girl" "1 pizza place"
di "`:word 2 of `phrase''"
>>> Answer: guys

* 3. Mixed list with inner and outer quotes
local phrase `" "2 guys" "1 girl" "1 pizza place" "'
di "`:word 2 of `phrase''"
>>> Answer: 1 girl
```

An aside: Macros vs scalars

How do the differ, and when should you use which?

- Both Stata macros and scalars can contain numeric and non-numeric information, but there are differences
- Macros
  - Limited to 1mill+ characters
  - Numeric information is converted (potential loss of accuracy)
  - Must be dereferenced (referred to using syntax)
  - e.g.:

```
local root = sqrt(2.15)
di `root'/4
>>> .36657196
```

- Scalars
  - Limited to 244 characters
  - No conversion of numeric information (full numeric precision)
  - Can be referred to by name
  - e.g.:

```
scalar root = sqrt(2.15)
di root/4
>>> .36657196
```

- Use scalars for intermediate calculations if precision is critical
- Use macros everywhere else

Assigning shortcuts with Macros

Mapping commands to function keys (*F-keys*)[5]

- In Windows OSs, it is possible to map the F-keys to commands using global macros
- On startup, Stata's default mapping (which may be redefined as desired) is as follows:

F-key	Definition	
F1	`help`	`help advice;`
F2	`#review;`	`desribe;`
F3	`desribe;`	`-`
F7	`save`	`save`
F8	`use`	`use`

- To view all of the macros currently defined in Stata's memory (including the current *F-key* mapping), simply issue the command `macro dir`

[5]This section from Driver (2005)

Example 1: Mapping commands to F-keys

Mapping `preserve` to F5 and `restore` to F6

- The `preserve` command preserves a copy of the current state of the data in Stata's memory, ensuring that data can be restored after do-file/program termination or after the `restore` command is issued
- This is particularly useful when executing experimental and potentially destructive procedures which one may wish to undo at a later stage.

Task: assign the `preserve` and `restore` commands to the *F5* and *F6* keys, respectively

```
* 1. Assign 'preserve' to F5
global F5 "preserve;"

* 2. Assign 'restore' to F6
global F6 "restore;"
```

- Ending the definition of a shortcut macro with a semicolon ensures that the command will be executed once the corresponding shortcut key is hit
  - Without the semicolon, hitting the shortcut key will only make the command apper in the Stata's *Command* window.

Making user-defined shortcut keys permanent

Using profile.do[6]

- Every time Stata is invoked, it searches for a file called *profile.do* and, if found, it executes all of the commands contained therein
- This means that it is possible to make your preferred *F-key* shortcut mappings permanent by placing their macro definitions in the *profile.do* file
- To enable Stata to find this file, it is recommended that it be located in *C:\ado\personal,* if "C:\" is the root directory where Stata is installed
- The commands that may appear in profile.do are by no means limited only to macro definitions
    - E.g. you could specify your favourite working directory in profile.do or
    - include a welcome message or
    - start a log file
    - etc.

[6] See StataCorp (2013*a*, p. 127)

Loops

What macros were born to do

- Though useful in and of themselves, macros are most useful in the context of loops
- Loops allow you to “loop through” or repeat blocks of code based on specified criteria
- Three types of loops in Stata
  - `forvalues`
    - loop over consecutive/fixed interval values
  - `foreach`
    - loop over elements of a list (values, variables, macros, or names)
  - `while`
    - loop while specified expression is evaluated as *true*
- Each type has slightly different syntax
- Which one you should use depends on what you want to accomplish
  - But it is often possible to accomplish something using either of the three

The why and how of loops, lists, and indices (Part I)

Looping over values

- Say you want to print/display the numbers [0, 10] in intervals of 2 below one another in the results window
- You could run the following code:

```
di 0    // display value 0
di 2    // display value 2
di 4    // display value 4
di 6    // display value 6
di 8    // display value 8
di 10   // display value 10
```

- Adequate in present example, but what if you wanted to
  - change the range to $[-10, 0]$ or to [0, 100]?
  - change the interval from 2 to 0.5?
  - or reverse the ordering (i.e. go from $[10, 0]$ in steps of -2)?
- When performing iterative or repetitive tasks, loops generally offer significant advantages i.t.o. power, flexibility, and/or efficiency

Example 1.1: Basic looping over values

Stata Snippet: Using `forvalues`, `foreach`, and `while` to loop over values/lists of values

```
* 1. forvalues loop
forvalues i = 0(2)10 {                       // for the range [0,10] in steps of 2
    di `i'                                   // display value
}                                            // repeat/end loop

* 2.1 foreach loop with specified numlist
foreach i of numlist 0 2 4 6 8 10 {          // for each of the values specified
    di `i'                                   // display value
}                                            // repeat/end loop

* 2.2 foreach loop with anything specified
foreach i in 0 2 4 6 8 10 {                  // for each of the things specified
    di `i'                                   // display value
}                                            // repeat/end loop

* 3. while loop with initial value and increment
local i = 0                                  // define intial value for local i
while `i' <=10 {                             // while value of local i<=10
    di `i'                                   // display value
    local i = `i'+2                          // increment local i by 2
}                                            // repeat/end loop
```

Example 1.2: Advanced looping over values I

Stata Snippet: Using indices to loop over elements of another list

- Basic looping as in the example above is already very powerful
- But there are scenarios where it is useful to dereference macros or elements of a macro list based on elements of a secondary *numlist* or macro list
  - One of the ways in which this can be done is to use the macro extended function `` `:word # of local' `` illustrated above
  - Another way is via the `tokenize` command which divides strings into tokens, storing the results sequentially in positional local macros `` `1' ``, `` `2' ``,... `` `n' ``
- Loops and programs become even more powerful/flexible with the aid of this function and/or command

Example 1.2: Advanced looping over values II

Stata Snippet: Using indices to loop over elements of another list via a macro extended function

```
* 1. Dereferencing elements of a numlist with a secondary numlist
local list 0 2 4 6 8 10             // store numlist in local `list'
forvalues i = 1(1)6 {               // for each value of `i' = [1,6]
    di `: word `i' of `list''       // display the `i'th number in `list'
}                                   // repeat/end loop

* 2. Dereferencing elements of a string list with a secondary numlist
local list "0" "2" "4" "6" "8" "10" // store string list in local `list'
forvalues i = 1(1)6 {               // for each value of `i' = [1,6]
    di "`: word `i' of `list''"     // display the `i'th word in `list'
}                                   // repeat/end loop
```

Example 1.3: Advanced looping over values III

Stata Snippet: Using indices to loop over elements of another list via the `tokenize` command

```
* 1. Indexing a numlist and looping over elements with a secondary numlist
tokenize 0 2 4 6 8 10       // store elements of list in sequential locals
forvalues i = 1(1)6 {       // for each value of `i' = [1,6]
    di ``i''                // display the `i'th positional local
}                           // repeat/end loop

* 2. Indexing a string list and looping over elements with a secondary numlist
tokenize `" "0" "2" "4" "6" "8" "10" "' // store elements of list in sequentia
forvalues i = 1(1)6 {               // for each value of `i' = [1,6]
    di "``i''"                      // display the `i'th positional local
}                                   // repeat/end loop
```

The why and how of loops, lists, and indices (Part 2)

Looping over variables

- In addition to iterating over values and *numlists*, it is also possible to loop over variables
  - When working with data there are many instances where we wish to repeat more or less the same procedure several times for different variables
  - the `foreach` loop is the main workhorse when it comes to looping over variables rather than values
- Say, for example, one wanted to
  1. cross tabulate educ against several other categorical/discrete variables or
  2. run a series of regressions, incrementally adding regressors to see how the results change or
  3. create a series of graphs showing the distributions of the log of household income from different sources?
- None of these task are particularly difficult, but they can be tedious if done manually
  - The solution is to loop over variables

Example 2.1: Basic looping over variables I

Stata Snippet: Using `foreach` to perform a series of cross-tabulations

```
* 1. Cross-tabs with educ and other vars: Manually
tab educ hh_head, col nofreq
tab educ female, col nofreq
tab educ race, col nofreq
tab educ marital, col nofreq
tab educ area, col nofreq
tab educ status, col nofreq

* 2. Cross-tabs with educ and other vars using foreach loop
foreach var of varlist hh_head female race marital area status {
    tab educ `var', col nofreq
}
```

Deciphering the code: Some notes

Stata Snippet: Using `foreach` to perform a series of cross-tabulations

- (10) This is the standard syntax that must be used when `foreach` is invoked to loop over variables
  - The choice of the term "var" here is completely arbitrary and simply sets up the alias (i.e. `` `var' ``) that may be used within the loop to refer to the respective variables in the list
  - "of varlist" is compulsory syntax needed to ensure that `foreach` loops over the *varlist* specified thereafter
- (11) If one notes that the local `` `var' `` is simply an alias for each of the respective variables specified in the *varlist,* then it should be obvious that this line simply evaluates to the command line on line 2 for the first iteration, the command line on line 3 for the second iteration, and so on until it evaluates to the command line on line 7 for the final iteration of the loop

Example 2.2: Basic looping over variables II

Stata Snippet: Using `foreach` to run a series of regressions, adding regressors incrementally

```
* 1. Run series of regressions, adding variables incrementally: Manually
reg lwages educ
reg lwages educ age
reg lwages educ age female
reg lwages educ age female i.race
reg lwages educ age female i.race i.marital
reg lwages educ age female i.race i.marital i.area

* 2. Run series of regressions, adding variables incrementally: foreach
foreach var in educ age female i.race i.marital i.area {
    local newlist `newlist' `var'
    reg lwages `newlist'
}
```

Deciphering the code: Some notes

Stata Snippet: Using `foreach` to run a series of regressions, adding regressors incrementally

- (10) The syntax here is different from that used in the previous example
  - We have included factor variable operators when specifying *i.race*, *i.marital*, and *i.area* . These variables do not exists as specified in the data, so Stata will not be able to find them if told to search only through the existing variables
  - what we have actually specified is a list of strings, rather than a list of variables (it does not matter that they refer to variables once specified in the context of the `regress` command)
  - "in" is compulsory syntax needed to ensure that `foreach` loops over strings, or anything other than values, variables, matrices, or scalars
- (11) The self-referential use of the local `‘newlist’` in the definition of `newlist` is a neat trick that allows one to incrementally build a list over loop iterations. To explain, consider the following:
  - Iteration 1: `‘newlist’` is empty such that the line evaluates to `local newlist ‘var’` which in turn evaluates to `local newlist educ`
  - Iteration 2: `‘newlist’` is already an alias for `educ`. The line thus evaluates to `local newlist educ ‘var’` which in turn evaluates to `local newlist educ age`
  - Iteration 3: `‘newlist’` is already an alias for `educ` age. The line thus evaluates to `local newlist educ age ‘var’` which in turn evaluates to `local newlist educ age female`
  - This process repeats itself and through every iteration, `‘newlist’` gains another variable

Example 2.3: Basic looping over variables III

Stata Snippet: Using `foreach` to generate a series of temporary logged variables and create graphs

```
* 1. Create series logged variables and kdensity graphs: manually
gen lhhinc_labour = ln(hhinc_labour)
gen lhhinc_grant = ln(hhinc_grant)
gen lhhinc_inv = ln(hhinc_inv)
gen lhhinc_remit = ln(hhinc_remit)
gen lhhinc_rent = ln(hhinc_rent)

kdensity lhhinc_labour, name(lhhinc_labour, replace)
kdensity lhhinc_grant, name(lhhinc_grant, replace)
kdensity lhhinc_inv, name(lhhinc_inv, replace)
kdensity lhhinc_remit, name(lhhinc_remit, replace)
kdensity lhhinc_rent, name(lhhinc_rent, replace)

* 2. Create temporary series of logged variables and kdensity graphs: foreach
foreach stub in labour grant inv remit rent {
    tempvar logvar
    gen `logvar' = ln(hhinc_`stub')
    kdensity `logvar', name(hhinc_`stub', replace)
}
```

Deciphering the code: Some notes

Stata Snippet: Using `foreach` to generate a series of temporary logged variables and create graphs

- (15) As in the previous example, we are looping over strings rather than variables. The syntax used is thus the standard `foreach` syntax for looping over *anything*
  - This was done purely for the sake of convenience. All of the variable names have a common prefix. It is therefore not necessary to write this prefix out every time
- (16) This line designates logvar as an alias that may be used to refer to a temporary variable which may or may not be created during the loop
- (17) This line generates the temporary variable *logvar* based on the alias `` `logvar' `` and sets it equal to the natural logarithm of the respective household income variables
  - Temporary variables exist only within the loops/programs/do-files within which they are created. The temporary variable *logvar* therefore ceases to exit as soon as the loop terminates. This is useful if we don't want the logged versions of the household income variables for any purposes other than generating the kdensity graphs (which we assume is the case here)
- (18) For the first iteration of the loop, this line evaluates to `kdensity logvar, name(hhinc_labour,replace)`

Example 3: Intermediate looping over variables I

Using `foreach` to loop over variables and call a macro extended function

- The example data contains a series of household possession variables
  - *radio*, *stereo*, *tv*, *computer*, *camera*, *cellphone*, *fridge*, and *car*
- At present, the variable labels associated with these variables are not sufficiently descriptive
  - It is not immediately obvious that the variables measure household ownership of the items
- You want to assign to each variable a more descriptive label
  - However, you do not wish to 'throw away' the existing variable labels, but simply want to prefix it with the phrase *"HH Owns a"*
- There are at least three ways of doing this
  1. Manually
  2. Semi-automated using a macro extended function
  3. Fully automated using a loop and a macro extended function

Example 3: Intermediate looping over variables I

Stata Snippet: Using `foreach` to loop over variables and call a macro extended function

```
* 1. Add a prefix to existing variable labels manually
label var radio     "HH owns a Radio (hf1)"
label var stereo    "HH owns a Hi-Fi stereo, CD/Mps player (hf2)"
label var tv        "HH owns a TV (hf3)"
label var computer  "HH owns computer(hf6)"
label var camera    "HH owns a camera (hf7)"
label var cellphone "HH owns a cellphone (hf8)"
label var fridge    "HH owns a fridge (hf13)"
label var car       "HH owns a private vehicle in running condition (hf17)"

* 2. Add a prefix to existing variable labels using macro extended function
label var radio     "HH owns a`:variable label radio'"
label var stereo    "HH owns a`:variable label stereo'"
label var tv        "HH owns a`:variable label tv'"
label var computer  "HH owns a`:variable label computer'"
label var camera    "HH owns a`:variable label camera'"
label var cellphone "HH owns a`:variable label cellphone'"
label var fridge    "HH owns a`:variable label fridge'"
label var car       "HH owns a`:variable label car'"

* 3. Add a prefix to existing variable labels using foreach loop
foreach var of varlist radio stereo tv computer camera cellphone fridge car {
    label var `var' "HH owns a`:variable label `var''"
}
```

Deciphering the code: Some notes

Stata Snippet: Using `foreach` to loop over variables and call a macro extended function

- (12) – (19) All of these lines employ the macro extended function `` `:variable label *varname*' `` to dereference the existing variable label for the variable in question. This is included within quotation marks as a suffix after the phrase *HH Owns a* when assigning the new variable labels to the variables.
- (22) This is the standard syntax that must be used when `foreach` is invoked to loop over variables
  - The choice of the term "var" here is completely arbitrary and simply sets up the alias (i.e. `` `var' ``) that may be used within the loop to refer to the respective variables in the list
  - "of varlist" is compulsory syntax needed to ensure that `foreach` loops over the *varlist* specified thereafter
- (23) If one notes that the local `` `var' `` is simply an alias for each of the respective variables specified in the *varlist,* then it should be obvious that this line simply evaluates to the command line on line 12 for the first iteration, the command line on line 13 for the second iteration, and so on until it evaluates to the command line on line 19 for the final iteration of the loop

Example 4: Advanced looping over variables I

Using nested `foreach` to loops to establish joint non-missingness for pairs of variables

- Stata's loop commands allow loops to be combined and nested within one another
  - One can combine/nest any number of `forvalues`, `foreach`, and/or `while` loops that loop over values or variables
- This exponentially increases the usefulness of loops, but also tends to make things more complicated and difficult to stay on top of
- For our final example, we will use two nested foreach loops to determine the proportion of observations in the data are jointly non-missing for different pairs of variables

Example 4: Advanced looping over variables II

Stata Snippet: Using nested `foreach` to loops to establish joint non-missingness for pairs of variables

```
* 1. Determine joint-non-missingness using nested foreach loops
local varlist language age female race marital pregnant
foreach var1 of varlist `varlist' {
    foreach var2 of varlist `varlist' {
        qui count if !missing(`var1',`var2')
        local nonmis = round(`r(N)'/_N*100)
        di "{res}`nonmis'% {txt} of observations are " ///
        "non-missing for both `var1' & `var2'"
    }
}
```

Deciphering the code: Some notes

Stata Snippet: Using nested `foreach` to loops to establish joint non-missingness for pairs of variables

- (3) Initialises the “outer” `foreach` loop
- (4) Initialises the “inner” `foreach` loop
  - It is important to understand the order of iteration when nesting loops. The inner loop will iterate over all of the variables in `` `varlist' `` before the outer loop proceeds to the second iteration. During the outer loop's second iteration, the inner loop will again iterate over all of the variable in `` `varlist' `` before the outer loop proceeds to the third iteration
- (5) use the `count` command to count the number of observations for which both ‘var1’ and ‘var2’ have non-missing values. The `qui` (short for `quietly`) prefix to the command will suppress any output that might otherwise be printed in the *Results* window due to the `count` command's execution
- (6)the `count` command stores its results in `r()` as the scalar `r(N).` This scalar value may either be called directly or dereferenced via `` `r(N)' `` as is done in line 6. This line line divides the number of jointly-nonmissing observations by the total number of observations in the data (as measured by the system variable `_N`), multiplied by 100, and rounded off to the nearest integer via the `round()` function. The answer is then stored in the local `` `nonmis' ``
  - repeat/terminate outer loop All of these lines employ the macro extended function `` `:variable label `` *`varname`*`'` to dereference the existing variable label for the variable in question. This is included within quotation marks as a suffix after the phrase *HH Owns a* when assigning the new variable labels to the variables.
- (9) end/terminate inner loop
- (10) end/terminate outer loop

Basic branching

Executing parts of code conditionally

- A further way in which loops and programs can be made more powerful, general, and robust is through the use of branching
- Branching allows one to conditionally execute blocks of code, depending on whether or not a certain condition is true
  - that is, it allow allows us to specify what should happen if a certain condition is true, and what should happen if that condition is false and/or another condition applies
- Code can be "branched" by using the `if`, `else`, and, in some instances, the `else if` commands
- Suppose, for the purposes of illustration, we wanted to `summarize` all int variables, `ds` all of str variables, and do nothing with the other variables in the example data
- Branching makes this easy to achieve

Example 1: Basic branching I

Stata Snippet: Using `foreach` with branching to `summarize` all *int*, `ds` all *str*, and ignore all other variables

```
* 1. Summarize all int, ds all str, ignore the rest
foreach var of varlist * {
    if "`:type `var''" == "int" {
        sum `var'
    }
    else if substr("`:type `var''",1,3) == "str" {
        ds `var'
    }
    else {
    }
}
```

Deciphering the code: Some notes

Stata Snippet: Using nested `foreach` to loops to establish joint non-missingness for pairs of variables

- (2) When used to specify a *varlist* in Stata, an asterisk serves as a shorthand for "all variables in the dataset"
- (3) – (5) This block of code will only be executed if the condition in line 3 is evaluated as true. In other words, only if the variable type of the variable in question is *int* (short for integer)
- (6) – (8) The `else if` command used to initialise this block of code means that it's contents will only be executed if
    - the statement in line 3 evaluates to *false* AND the statement in line 6 evaluates to *true*
- (9) – (10) The `else` command used to initialise this block of code means that it's contents (which is nothing) will only be executed if
    - the statement in line 3 evaluates to *false* AND the statement in line 6 evaluates to *false* (i.e. the variable in question is neither a string nor an integer variable)

Example 2: Advanced branching I

Using nested `foreach` loops plus branching to establish joint non-missingness for unique pairs of variables

- Branching can be used to improve our code for determining joint non-missingness used above
    - The *varlist* specified in the code contains 6 variables
    - A maximum of 18 unique pairs can be formed from these 6 variables
    - However, the code produced 36 lines of output
    - This is a direct result of the fact that the outer and inner loops contain the same varlists
        - As the outer loop iterates over the inner loop, the order of the variables from the varlist that are passed to the commands is reversed. i.e. the command produces results twice for each unique pair of variables
- With the aid of branching, we can prevent the code from generating results for pairs of variables for which results have already been generated

Example 2: Advanced branching II

Stata Snippet: Using nested `foreach` loops plus branching to establish joint non-missingness for unique pairs of variables

```
* 1. Determine joint-non-missingness using nested foreach loops and branching
local varlist language age female race marital pregnant
foreach var1 of varlist `varlist' {
    foreach var2 of varlist `varlist' {
        local pair `var2' `var1'
        if strpos("`pairs'","`pair'") == 0 {
            qui count if !missing(`var1',`var2')
            local nonmis = round(`r(N)'/_N*100)
            di "{res}`nonmis'% {txt} of observations are " ///
            "non-missing for both `var1' & `var2'"
            local pairs `pairs' `var1' `var2'
        }
        else {
        }
    }
}
```

Deciphering the code: Some notes

Stata Snippet: Using nested `foreach` loops plus branching to establish joint non-missingness for unique pairs of variables

- There are only three substantive new lines to the code: (5), (6), (11)
  - (13) – (14) These lines are technically also new, but the loop would function precisely the same if they were deleted from the code
- (5) This stores the variable pairs specified by the outer and inner `foreach` loops in reverse order in the local `pair`
- (6) – 12 This block of code will only be executed if the condition in line 6 is evaluated as true
  - The string function used here searches for the position in the string `` ``pairs'" `` at which `` ``pair'' `` is found. If it is not found, the expression evaluates to zero
  - (5) This uses the same self-referential "trick" to incrementally concatenate a string containing all of the variable pairs for which lines (7) – 10 have been executed
  - the rationale here is that, if the string `` ``pair'" `` is found within `` ``pairs'' `` such that `` strpos("`pairs'","`pair'") != 0 ``, then it must be the case that the joint-non-missingness for that pair of variables has already been determined and the current iteration of the inner loop should therefore be skipped
  - the evaluates values of `` `pair' `` and `` strpos("`pairs'","`pair'") `` for each of the iterations of the outer and inner loops are presented on the next frame

Deciphering the code: Some notes

Stata Snippet: Using nested `foreach` loops plus branching to establish joint non-missingness for unique pairs of variables

Outer Itt	Inner Itt	`` `pair' ``	``strpos("`pairs'","`pair'")``
1	1	`language language`	0
1	2	`age language`	6
1	3	`female language`	0
1	4	`race language`	0
1	5	`marital language`	0
1	6	`pregnant language`	0
2	1	`language age`	0
2	2	`age age`	0
2	3	`female age`	0
2	4	`race age`	0
2	5	`marital age`	0
2	6	`pregnant age`	75
3	1	`language female`	19
3	2	`age female`	24
3	3	`female female`	0
3	4	`race female`	0
3	5	`marital female`	129
⋮	⋮	⋮	⋮
6	5	`marital pregnant`	217
6	6	`pregnant pregnant`	225

User-defined programs

From loops to self-contained commands

- In the much the same way that loops extend the functionality of basic Stata code, programs (can) extend the functionality of loops and macros
- User-defined programs offer additional layers of power, flexibility, and efficiency, at the potential cost of greater complexity
- User-defined commands (hereafter programs) function in precisely the same way as Stata's commands
    - Most of Stata's commands are in fact user-defined programs or based on user-defined programs
- Programs
    - are called via a unique command name,
    - may or may not accept/require compulsory and/or optional arguments and
    - once issued execute one or more 'procedures' in Stata
- What these procedures are and precisely how they are executed depends on how the program, and its syntax, is defined

Developing Stata programs

Why would you want to?

- There are a number of reasons why Stata users develop programs, but they mostly relate to actual or percevide deficiencies in the available repertoire of Stata commands
- Some of the major reasons why you may want to develop a program are
  - There is no existing Stata command that can do what you need it to do
  - You are not aware of any existing Stata command(s) that can do what you need it to do (more likely)
  - Stata's command(s) for doing what you want are needlessly slow and you believe you can program something more efficient
  - There are Stata commands that can do 99% of what you want/need, but you want that last 1%
  - You want to have cleaner do-files, type fewer lines of code, and speed up your coding

Defining a Stata program

Syntax in a number of flavours

- There are a number of ways in which one can define Stata programs
- These differ in terms of
  - The scope, flexibility, and robustness of the programs they can be used to define
  - How easy their definition syntax is (how easy it will be for you to define the program)
  - How general their specification syntax is (how easy it will be for another Stata user to call the program)
- Learning to create your own Stata programs can be daunting at first and it is tempting to avoid complicated syntax insofar as is possible
- However, there are good reasons why you should try to learn how to adhere to standard programming practice and standard Stata syntax

Example 1: A Basic program I

`revalve`: combining `rename`, `label var`, and `replace`

- The initial cleaning of a new dataset in Stata tends to involve quite a lot of variable renaming, variable labelling, and variable recoding/replacement
- Stata has perfectly adequate commands for doing any one of these tasks, but in some instances it would be nice if one could rename, assing a variable label, and set to missing invalid values on a variable in one step
- This is precisely what the `revalve` command in the example below does
- Note that `revalve` does nothing novel
  - it simply serves as a wrapper for existing Stata commands in an attempt to reduce the amount of typing required to clean data
-

Example 1: A Basic program II

revalve: combining rename, label var, and replace

```
* 1. Revalve based on positional arguments
capture program drop revalve
program revalve
    rename `1' `2'
    label variable `2' "`3'"
    cap replace `2'=. if `2'<0
end

* 2. Revalve based on aliased positional arguments
capture program drop revalve
program revalve
    args oldvarname newvarname varlabel
    rename `oldvarname' `newvarname'
    label variable `newvarname' "`varlabel'"
    cap replace `newvarname'=. if `newvarname'<0
end

* 3. Revalve based on standard Stata syntax
capture program drop revalve
program revalve
    syntax varlist(min=1), Rename(string) [Label(string asis)]
    rename `varlist' `rename'
    label variable `rename' "`label'"
    cap replace `rename'=. if `rename'<0
end
```

Section 3: Developing a program

Stata's `encode` command I

An incredibly useful, but often insufficient command[7]

- It is generally preferable to convert string variables to (labeled) numeric variables for the purposes of analysis
- `encode` efficiently maps distinct values of a string variable to integer-valued numeric variable such that string values become value labels
    - By default, it uses alphanumeric order of distinct string values to determine numeric values
    - This can be problematic if a string variable's alphanumeric ordering differs from its rank/logical ordering
    - E.g. the string variable *read_hl* in the example data which reflects respondents' self-reported home language reading levels in one of four categories
        - *"Not at all"*, *"Not well"*, *"Fair"*, and *"Very Well"*
        - The variable has a clear logical ordering, but this ordering does not correspond to the alphabetical ordering of the categories

[7] See Schechter (2011)

Stata's `encode` command II

An incredibly useful, but often insufficient command

- The table below illustrates what would happen if one used `encode` to convert the string variable, *read_hl*, into a numeric variable:

Original string variable		*New encoded variable*	
Values (alphabetic)	Rank ordering	Values	Value Labels
`Fair`	3	1	`Fair`
`Not at all`	1	2	`Not at all`
`Not well`	2	3	`Not well`
`Very Well`	4	4	`Very Well`

- How might one overcome this potential pitfall?
- One obvious answer would be to avoid the use of `encode` and instead manually recode the variable...

Example 1: Converting a string to a numeric variable I

Stata Snippet: The "brute force" manual way

```
* 1. Create new variable 'newvar' and fill with missing values
gen newvar = .

* 2. Fill in values of 'newvar' based on string values of 'read_hl'
replace newvar = 1 if read_hl == "Not at all"
replace newvar = 2 if read_hl == "Not well"
replace newvar = 3 if read_hl == "Fair"
replace newvar = 4 if read_hl == "Very Well"

* 3. Define/Modify appropriate value label
label define newvar 1 "Not at all", modify
label define newvar 2 "Not well", modify
label define newvar 3 "Fair", modify
label define newvar 4 "Very Well", modify

* 4. Assign new value label and appropriate variable label to 'newvar'
label values newvar newvar
label var newvar "Respondent's self-reported home language reading level"
```

Example 1: Converting a string to a numeric variable II

The "brute force" manual way reviewed

- The code obviously does what it is supposed to, but...
  - It is not very general
    - What if there had been more than 4 categories?
  - It is not very flexible
    - If we messed up the ordering and had to rectify our mistake(s), it could require quite a lot of editing
  - It is prone to syntax and spelling errors
    - Stata is case-sensitive and does not tolerate spelling errors
  - We also had to manually assign the variable label
- To overcome some of these issues, we might consider using the `encode` and `recode` commands in conjunction

Example 2: Converting a string to a numeric variable I

Stata Snippet: The “moderate force” manual way with encode and recode

```
* 1. Convert string var 'read_hl' to labeled numeric var 'newvar'
encode read_hl, gen(newvar)

* 2. Adjust 'newvar's values using recode
recode newvar (1=3) (2=1) (3=2)

* 3. Modify 'newver' value label
label define newvar 1 "Not at all", modify
label define newvar 2 "Not well", modify
label define newvar 3 "Fair", modify
```

Example 2: Converting a string to a numeric variable II

The "moderate force" manual way with `encode` and `recode` reviewed

- This code seems like a significant improvement over the previous example
  - It is much more compact
  - Only some of the values and value labels needed to be redefined
  - The variable label was automatically assigned to the *newvar* variable
- But it is still not very general and is overly reliant on manual input
- Next up: introduce some automation using a loop and macro extended functions

Example 3: Converting a string to a numeric variable I

Stata Snippet: The semi-automatic way

```
* 1. Create new variable 'newvar' and fill with missing values
gen newvar = .

* 2. Get list of unique values for 'read_hl' and store in local `levels'
levelsof read_hl, local(levels)

* 3. Specify initial value for index local `i'
local i = 1

* 4. Automatically recode and label 'newvar' using loop with ordered numlist
foreach num of numlist 3 1 2 4 {
    replace newvar = `num' if read_hl == "`: word `i' of `levels''"
    label define newvar `num' "`: word `i' of `levels''", modify
    local ++i
}

* 5. Assign new value label and appropriate variable label to 'newvar'
label values newvar newvar
label var newvar "`:variable label read_hl'"
```

Deciphering the code: Some notes

Stata Snippet: The semi-automatic way

- (11) Begin iteration over numbers 3, 1, 2, 4
- (12) Recall that
  - `` `levels' = `"Fair"' `"Not at all"' `"Not well"' `"Very Well"' ``
  - For the first iteration of the loop, line 12 thus evaluates to

  -
    ```
    >>  replace newvar = 3 if read_h1 == "`: word 1 of `levels'"
    >>> replace newvar = 3 if read_h1 == "Fair"
    ```
- (13) For the second iteration of the loop, line 13 evaluates to

  -
    ```
    >>  label define newvar 1 "`: word 2 of `levels'", modify
    >>> label define newvar 1 "Not at all", modify
    ```
- (14) Increment the indexing local `` `i' `` by one
- (19) Use a macro extended function to assign the variable label associated with *read_hl* to the *newvar*

Example 3: Converting a string to a numeric variable II

The semi-automatic way reviewed

- This code is more general than before
  - We only need to specify the encoding ordering, and it largely automates the recoding and labelling process
- BUT...
  - The code is noticeably longer and more complex than in the previous example
  - It still requires us to manually assign the variable and value labels and set up the placeholder variable
- In the present context, the disadvantages of this code may actually outweigh its advantages
- This raises an important issue concerning the use of loops and programs in Stata...

An Aside: To loop, or not to loop...

When you should and when you should not use loops

- Loops do not necessarily make code more efficient (they can do the opposite)
  - Unless we intend to repeat the process, there is no real need for a loop in the context of the present example
- The initial setup cost of a loop can be high
  - User-defined programs tend to have significantly higher setup costs
- Decision rule: ese loops/programs *iff*

$$E\left[\text{Effort/Time saved by using loop/program}\right] > E\left[\text{Effort/Time required to set up loop/program}\right]$$

Towards an ordered `encode`

Where do we go from here?

- It is useful to note that the example data contains three more string variables (*write_hl*, *read_en*, and *write_en*) that have exactly the same coding as *read_hl*
    - we may want to use our code to achieved ordered encodes of these variables also
- More generally, being able to specify the order in which a string variable is encoded might prove useful in a range of contexts
- Our goal is thus
    1. To make it as easy as possible to achieve an ordered encode and
    2. To make the code with which we do so as general/flexible as possible
- We can achieve both of these goals by converting our loop code into a self-contained program

An Aside: Creating user-defined commands/programs I

The masochist's delight

- Writing good user-defined commands/programs in Stata...
  - ...tends to require significant initial investments and setup costs
  - ...is likely to be subject to a lot of trial and error
  - ...requires practice, patience, and perseverance
  - ...is more than worth it once you get the hang of it

An Aside: Creating user-defined commands/programs II

How you should write your programs

- Plan ahead
    - What arguments and options will your program need/take, etc.
    - Will it accept weights and if statements, etc.
- Use good syntax
    - Adhere to syntax and usage guidelines given in StataCorp (2013*b*, pp. 505 - 519)
    - This makes debugging easier, makes it easier for others to understand what your program does, and makes it easier for you to remember what you were trying to achieve
- Start simple
    - Try to create a minimal working example (MWE) that achieves basic objectives first and then add complexity and refinement in incremental steps

Example 4: A basic oencode I

Converting the code into a minimal working example (MWE) program

- We want to create our first version of the program, trying to keep the number of changes we make to the bare minimum
- At the very least, our MWE will require the following elements:
    - A program/command name
        - We'll call the program `oencode` (short for "ordered encode") since it is moderately descriptive and also a non-reserved name
    - Allowance for two compulsory arguments:
        1. The name of the string variable to be encoded and
        2. The order in which the string variables values should be encoded

Example 4: A basic oencode II

Stata Snippet: The first MWE

```
* oencode: program to encode string variable using user-specified ordering
* Version 1.0
program define oencode
    syntax varlist, order(numlist)
    gen newvar = .
    levelsof `varlist', local(levels)
    local i = 1
    foreach num of numlist `order' {
        replace newvar = `num' if `varlist' == "`: word `i' of `levels''"
        label define newvar `num' "`: word `i' of `levels''", modify
        local ++i
    }
    label value newvar newvar
    label var newvar "`:variable label `varlist''"
end

oencode read_hl, order(3 1 2 4)
```

Deciphering the code: Some notes

Stata Snippet: The first MWE

- (3) Define the program as `oencode`
- (4) The syntax specified requires the program to be called with two compulsory arguments:
  - an input *varlist* (one or more existing variables in the dataset). When the program is evaluated, this input is aliased as the local `` `varlist' ``
  - a input *numlist* (the user-specified order in which the input variable should be encoded). When the program is called, the user must specify this *numlist* within the wrapper `order()`. When the program is evaluated, this input is aliased as the local `` `order' ``
- (6) Get the list of unique values for the variable(s) specified in the input *varlist* (i.e. the variable(s) aliased by `` `varlist' ``) and store in the local `` `levels' ``
- (8) Begin iteration over the value(s) specified in the input *numlist* (i.e. the value(s) aliased by `` `order' ``)
- (15) Indicate the end of the program definition
- (17) Call the program `oencode` with *read_hl* as the input *varlist* and '3 1 2 4' as the input `order()` *numlist*

Example 4: A basic `oencode` III

The first MWE reviewed

- The MWE works, but it is very far from perfect
  - The user has no control over the name of the new encoded variable
    - it is always generated as *newvar*, which is probelmatic if there is already a variable called *newvar* or if you want to run `oencode` more than once
  - The current specification of *varlist* allows more than one variable to be specified as input
  - The input variable is not explicitly required to be a string variable
  - The program generates display output
    - This is not only unnecessary, but also requires additional computation time
- There are many more issues, but let's address these ones first before moving on

Example 5: A working oencode I

Stata Snippet: The second MWE

```
* oencode: program to encode string variable using user-specified ordering
* Version 1.1
program define oencode
    syntax varlist(max=1 str), GENerate(string) Order(numlist)
    qui gen `generate' = .
    qui levelsof `varlist', local(levels)
    local i = 1
    foreach num of numlist `order' {
        qui replace `generate' = `num' if `varlist' == "`: word `i' of `levels
        label define `generate' `num' "`: word `i' of `levels''", modify
        local ++i
    }
    label value `generate' `generate'
    label var `generate' "`:variable label `varlist''"
end

oencode read_hl, gen(arbitraryvarname) o(3 1 2 4)
```

Deciphering the code: Some notes

Stata Snippet: The second MWE

- (4) The syntax line introduces a host of changes from the previous example
  - the *varlist_specifier* `max = 1` restricts the number of input variables that may be specified to one
  - the *varlist_specifier* `str` restricts the type of input variable that may be specified to string variables only
  - the addition of `GENerate(string)` to the syntax line adds a further compulsory argument to the command. The user must specify a name for the new encoded variable to be created within the wrapper `generate()`.
  - The respective uppercase parts of the `GENerate()` and `Order()` wrappers indicate the shortest abbreviations that may be used to specify these compulsory arguments
- (5),(6),(9) Prefixing these command lines with `qui` (short for `quitely`) suppresses any output that they would otherwise generate
- (5),(9),(10),(13),(14) In each of these lines, *newvar*, has been replaced by `` `generate' `` which is the alias for the new variable created by the command
- (17) Call the program `oencode` with *read_hl* as the input *varlist, arbitraryvar* as the name of the new encoded variable, and '3 1 2 4' as the input `order()` *numlist*

Example 6: A stable `oencode` I

Making the program more robust

- To make our code more robust we will introduce some changes that
  - Check if the number of elements specified in the `order()` *numlist* matches the number of elements in the `` `levels' `` alias,
  - prevents execution of the program if it does not and
  - informs the user of the program termination and the nature of the specification error when neccesary

Example 6: A stable oencode II

Stata Snippet: Making oencode more robust by adding escape clauses

```
* oencode: program to encode string variable using user-specified ordering
* Version 1.2
program define oencode
    syntax varlist(max=1 str), GENerate(string) Order(numlist)

    qui levelsof `varlist', local(levels)

    if `:word count `order'' < `:word count `levels'' {
        error 122
        exit
    }
    else if `:word count `order'' > `:word count `levels'' {
        error 123
        exit
    }
```

...

...

Example 6: A stable oencode II (cont.)

Stata Snippet: Making oencode more robust by adding escape clauses

```
...
...
     qui gen `generate' = .
     local i = 1
     foreach num of numlist `order' {
         qui replace `generate' = `num' if `varlist' == "`: word `i' of `levels
         label define `generate' `num' "`: word `i' of `levels''", modify
         local ++i
     }
     label value `generate' `generate'
     label var `generate' "`:variable label `varlist''"
 end

 oencode read_hl, gen(randomvar) o(3 1 2)        // too few values
 oencode read_hl, gen(randomvar) o(3 1 2 4 5)    // too many values
 oencode read_hl, gen(randomvar) o(3 1 2 4)      // correct
```

Deciphering the code: Some notes

Stata Snippet: Making `oencode` more robust by adding escape clauses

- (6),(17) The order in which these two lines of code are executed has been switched around in the code to prevent execution of the command in the event of specification errors
- (8) This line can be interpreted as: *count the number of elements/words in the numlist specified within the* `order()` *wrapper and count the number of elements/words in the* `` `levels' `` *local. If the former is smaller than the latter, do the following...*
  - (9) Issue Stata error code number 122: *"invalid numlist has too few elements"*[8]
  - (10) Terminate the program immediately without executing any further part of it
- (12) This line can be interpreted as: *if the previous if statement was evaluated as 'false', count the number of elements/words in the numlist specified within the* `order()` *wrapper and count the number of elements/words in the* `` `levels' `` *local. If the former is greater than the latter, do the following...*
  - (13) Issue Stata error code number 123: *"invalid numlist has too many elements"*
  - (14) Terminate the program immediately without executing any further part of it
- (28) – (30) Call the program `oencode` with too few, too many, and the right number of elements in the input `order()` *numlist*

[8] See StataCorp (2013*b*, pp. 182 - 195) for a list of Stata's error codes

From a stable to a final program

Making oencode more flexible by adding options

- Our program is now fairly stable, but there remains room for improvement (almost always the case) and scope for enhancements
  - E.g. when encoding a string variable, the user may want to replace the original string variable with the encoded one rather than generating an additional variable in the data
  - Obviously, this can be done by using `oencode` to generate a new variable, dropping the old string variable, assigning the old string variable name to the new encoded variable, and ordering the new variable where the original string variable used to be positioned in the data
  - However, it would be neat if we could include the option to do precisely this directly in our `oencode` command

Example 7: The final `oencode` I

Making `oencode` more flexible by adding options

- For our final iteration of the `oencode` program, we will change the code such that it allows the user
    - Either to generate a new encoded variable by specifying a name in the `generate()` wrapper or to replace the original sting variable with an encoded version by specifying a `replace` option.
- This requires
    - making the `generate()` argument optional and introducing an additional `replace` optional argument
    - including exit clauses in the event that the user specifies both or neither of the `generate()` and `replace` optional arguments
    - specifying what happens if the user choose the `replace` option instead of the `generate()` option

Example 7: The final oencode II

Stata Snippet: Making oencode more flexible by adding options

```
* oencode: program to encode string variable using user-specified ordering
* Version 1.3
program define oencode
    syntax varlist(max=1 str), Order(numlist) [GENerate(string) REPlace]

    qui levelsof `varlist', local(levels)

    if "`generate'" != "" & "`replace'" != "" {
        di as err "options generate and replace are mutually exclusive"
        exit 198
    }
    else if "`generate'" == "" & "`replace'" == "" {
        di as err "must specify either generate or replace option"
        exit 198
    }
    if `:word count `order'' < `:word count `levels'' {
        exit 122
    }
    else if `:word count `order'' > `:word count `levels'' {
        exit 123
    }
```

Stata Snippet: The final `oencode` II (cont.)

```
    if "`generate'" != "" {
        qui gen `generate' = .
        local i = 1
        foreach num of numlist `order' {
            qui replace `generate' = `num' if `varlist' == "`: word `i' of `lev
            label define `generate' `num' "`: word `i' of `levels''", modify
            local ++i
        }
        label values `generate' `generate'
        label var `generate' "`:variable label `varlist''"
    }
    else if "`replace'" != "" {
        rename `varlist' _0lD_`varlist'
        qui gen `varlist' = .
        local i = 1
        foreach num of numlist `order' {
            qui replace `varlist' = `num' if _0lD_`varlist' == "`: word `i' of
            label define `varlist' `num' "`: word `i' of `levels''", modify
            local ++i
        }
        order `varlist', before(_0lD_`varlist')
        label values `varlist' `varlist'
        label var `varlist' "`:variable label _0lD_`varlist''"
        drop _0lD_`varlist'
    }
end
```

Deciphering the code: Some notes

Stata Snippet: Making `oencode` more flexible by adding options

- (9) – (12) Specifies what will happen if both the `generate()` and `replace` options are specified
- (13) – (16) Specifies what will happen if neither the `generate()` nor the `replace` options are specified
- (17) – (27) Specifies what will happen if the `generate()` option is specified. The code inside this block is the same as in the previous example
- (28) – (41) Specifies what will happen if the `replace` option is specified. The code in this block differs somewhat from what was used before:
  - (29) Rename the original input string variable to avoid renaming conflicts
  - (30) Generate a placeholder variable with the same original name as the now renamed input variable
  - (37) Order the new encoded variable before the renamed original input variable
  - (40) Drop the renamed original input variable from the data

Making `oencode` permanent

Adding the program as an ado file

- This last version of the `oencode` command is general, compact, and fairly robust to specification error
  - There are definitely more refinements and enhancements we could make, but the present version should suffice for most uses
- To permanently add it to Stata's repertoire of commands, we need to save the snippet of program code within a file entitled *oencode.ado* and store it under the *C:\ado\personal* folder
- This will ensure that the `oencode` command is available the next time Stata is opened
- As a final example, we'll test how well the `oencode` command functions

Example 8: Using oencode

Stata Snippet: Using oencode within loops

Task: convert all of the string variables on self-reported reading and writing ability and on self-reported emotional well-being into labelled numeric variables

```
* 1. encode variables on reading and writing
foreach var of varlist read_hl write_hl read_en write_en {
    oencode `var', order(3 1 2 4) replace
}

* 2. encode variables on emotional well-being
local list bothered focus depressed effort hopeful fearful restless lonely
foreach var of varlist `list' {
    oencode `var', order(4 3 1 2) replace
}
```

Section 4: Examples & Applications

Accessing command results and storing in matrices I

Stata Snippet: Getting results stored in `r()` and using them to create a matrix

Task: Summarize hourly wages and store its count, mean, standard deviation, and minimum and maximum value in a nicely formatted matrix

```
* 1. Summarize wages, store results in matrix, adorn matrix, display matrix
sum wages
mat wages = `r(N)',`r(mean)',`r(Var)',`r(sd)',`r(min)',`r(max)'
mat rownames wages = wages
mat colnames wages = Obs Mean Variance StDev Min Max
matlist wages
```

Accessing command results and storing in matrices II

Stata Snippet: Getting results stored in `r()` and using them to create matrices using a loop[9]

Task: Summarize hourly wages, monthly earnings, per capita household income, and household income and store the counts, means, standard deviations, minimum and maximum values in a nicely formatted matrix

```
* 2. Summarize vars, store results in matrices, adorn matrices
foreach var of varlist wages earnings pchhinc hhincome {
    sum `var'
    mat `var' = `r(N)',`r(mean)',`r(sd)',`r(min)',`r(max)'
    mat rownames `var' = `var'
    mat colnames `var' = Obs Mean StDev Min Max
}

* 3. Combine individual matrices into one matrix and display
mat TOTAL = wages \ earnings \ pchhinc \ hhincome
matlist TOTAL

* 4. Use estout to display or export matrix
estout mat(TOTAL)
```

[9] Requires the Stata package `estout`

Accessing and manipulating estimation results

Stata Snippet: Getting standard errors from results stored in e() after regress

Task: Run a basic mincerian earnings function and access/display the standard errors from the estimation

```
* 1. Displaying results stored in e() after regress
reg lwages c.educ##c.educ c.age##c.age
ereturn list            // Display results store in e()

* 2. Displaying the variance-covariance matrix stored in e() after regress
reg lwages c.educ##c.educ c.age##c.age
matlist e(V)            // Display matrix e(V)

* 3. Extracting standard errors from the variance-covariance matrix
reg lwages c.educ##c.educ c.age##c.age
matlist vecdiag(cholesky(e(V)))

* 4. Storing standard errors extract from variance-covariance matrix in vector
reg lwages c.educ##c.educ c.age##c.age
mat se = vecdiag(cholesky(e(V)))
```

Accessing, manipulating, and storing estimation results

Stata Snippet: Getting standard errors from results stored in e() after regress via simple command[10]

Task: Run a basic mincerian earnings function and access the standard errors from the estimation using a simple command

```
* 5. Adding vector of standard errors to results store in e()
reg lwages c.educ##c.educ c.age##c.age
estadd matrix se = vecdiag(cholesky(e(V)))
ereturn list              // Display results store in e()

* 6. Define program that automatically adds vector of standard errors to
* results store in e() following the regress command
cap program drop addse
program define addse
    version 13.1
    cap qui estadd matrix se = vecdiag(cholesky(e(V)))
end
```

[10]Requires the Stata package `estout`

Automating graph adornment I

Stata Snippet: The rigid way: doing it all manually

Task: Create a local polynomial graph between two variables and provide basic adornments to the figure

```
* 1. Basic graph without adorment
twoway (lpoly lwages lhhincome)

* 2. Graph with basic manual adornment
# delimit ;
twoway (lpoly lwages lhhincome),
        ytitle("Log of hourly wages",  size(small))
        xtitle("Log of monthly household income", size(small))
        ylabel(, valuelabel glcolor(gs10) labsize(small) glwidth(vvvthin) gmax
        xlabel(0/17, valuelabel labsize(small))
        title("lwages vs lhhincome", size(medium));
# delimit cr
```

Automating graph adornment II

Stata Snippet: The flexible way: doing it all via macros

Task: Create a local polynomial graph between two variables and provide basic adornments to the figure

```
* 3. Graph with basic automated adornment
local yvar lhhincome
local xvar educ
sum `xvar'
local minx = floor(`r(min)')
local maxx = ceil(`r(max)')

# delimit ;
twoway (lpoly `yvar' `xvar'),
        ytitle("`:variable label `yvar''",  size(small))
        xtitle("`:variable label `xvar''", size(small))
        ylabel(, valuelabel glcolor(gs10) labsize(small) glwidth(vvvthin) gmax
        xlabel(`minx'/`maxx', valuelabel labsize(medsmall))
        title("`yvar' vs `xvar'", size(medium));
# delimit cr
```

Automating graph creation and combination

Stata Snippet: Creating subgraphs via `foreach` and then combining

Task: Graphically illustrate the distribution of the log of household income from labour, grants, government sources, investments, remittances, and rent on one graph

```
* 1. Create automatically adorned kdensities for log of hhincome from sources
foreach source in labour grant govt inv cap rent {
    tempvar logvar
    gen `logvar' = ln(hhinc_`source')
    twoway (kdensity `logvar' if inrange(`logvar',0,11)), ///
        name(`source', replace) ytitle("Density")       ///
        xtitle("Log of household income from `source'") ///
        xscale(range(0 11)) xlabel(0/11) title("`source'", size(small))
    local graphlist `graphlist' `source'
}
* 2. Combine kdensities into single graph
graph combine `graphlist', cols(2) ycommon
```

Fixed-interval categorical from continuous data I

Stata Snippet: The long way: doing it all manually

Task: Generate a labelled age cohort variable with intervals of 4 years based on the *age* variable

```
* 1. Create and fill 'agecohort' variable based on values of 'age'
gen agecohort = .
replace agecohort = 1 if inrange(age,0,4)
replace agecohort = 2 if inrange(age,5,9)
........................................
replace agecohort = 21 if inrange(age,100,104)
replace agecohort = 22 if inrange(age,105,109)

* 2. Define/modify value labels
label define agecohort 1 "0 - 4", modify
label define agecohort 2 "5 - 9", modify
........................................
label define agecohort 21 "100 - 104", modify
label define agecohort 22 "105 - 109", modify

* 3. Assign variable and value labels and order variable
label values agecohort agecohort
label var agecohort "Age cohort"
order var agecohort, after(age)
```

Fixed-interval categorical from continuous data II

Stata Snippet: The short way: doing it via a loop

```
* 1. Create and fill 'agecohort' var & value labels based on values of 'age'
gen agecohort = .
local i = 1
forvalues r = 0(5)105 {
    local s = `r' + 4
    replace agecohort = `i' if inrange(age,`r',`s')
    label define agecohort `i' "`r' - `s'", modify
    local ++i
}

* 2. Assign variable and value labels and order variable
label values agecohort agecohort
label var agecohort "Age cohort"
order agecohort, after(age)
```

Converting stored results to matrices to variables

Stata Snippet: Graphing point estimates and confidence intervals with complex survey weighting

Task: Calculate the proportion of females for each race group and graphically illustrate the point estimates along with the 95% confidence intervals

```
* 1. Estimate proportion of each gender by race group
svyset [pweight=weight1], vce(linearized) singleunit(missing)   // svyset
svy: proportion empl, over(race)

* 2. Import results stored in r() into mata and apply transformations
mata _EST = st_matrix("r(table)")[1,5...]'       // get point estimate
mata _EST_L = st_matrix("r(table)")[5,5...]'     // get 95% CI lower bound
mata _EST_U =st_matrix("r(table)")[6,5...]'      // get 95% CI upper bound

* 3. Creat column vector of race categories
mata _XVALS = ("African"\"Coloured"\"Indian"\"White")

* 4. Post mata column vectors as Stata variables
getmata _XVALS _EST _EST_L _EST_U, force

* 5. Encode string variable to labeled numeric for graphing
encode _XVALS, gen(_XCATS)

* 6. Graph bars with 95% confidence intervals
twoway (bar _EST _XCATS) (rcap _EST_L _EST_U _XCATS), xlabel(, valuelabel)
```

Emulating by __: egen using collapse & merge

Stata Snippet: Two ways of performing procedures within/accross groups

Task: Calculate the number of household members and household residents per household

```
* 1. Calculate number of hh members and hh residents using egen
by hhid, sort : egen float hh_residents1 = count(pid)
tempvar counter                     // Designate tempvar alias
gen `counter' = 1                   // Generate temporary counter variable
by hhid, sort : egen float hh_members1 = count(`counter')

* 2. Calculate number of hh members and hh residents using collapse/merge
tempfile original collapsed         // Designate tempfile aliases
save `original'                     // Save current data in tempfile `original'
gen counter = 1                     // Gen counter variable
collapse (count) counter (count) pid, by(hhid)  // Collapse by hhid and count
rename counter hh_members2          // Rename collapsed var (make descriptive)
rename pid hh_residents2            // Rename collapsed var (make descriptive)
save `collapsed'                    // Save collapsed data in tempfile `collapsed'
use `original'                      // Open `original' data
qui merge m:1 hhid using `collapsed'    // Merge in vars from `collapsed'

* 3. Establish that egen and collapse-merge method yield same results
assert hh_members1 == hh_members2
assert hh_residents1 == hh_residents2
```

Thank you

Stata Reference Manuals

The most complete and comprehensive reference

- Stata's series of reference manuals are a fantastic, yet often underutilised resource for learning how to work with and program in Stata
- These manuals contain far more complete information than Stata's help files and are generally easier to read and understand
  - The *"Remarks and examples"* sections (available for most of the entries in the manual), in particular, are useful when trying to learn how to use a specific command or function
- Reference manuals can be accessed in *pdf* format online or directly from Stata by navigating to *File* ▹ *PDF Documentation*

Stata Press publications and books about Stata

Further resources for learning Stata available from www.stata.com

- A number of excellent books on Stata and statistics can be purchase from the Stata Bookstore
  - These books cover a wide array of topics and applications ranging from basic data management and workflow in Stata to advanced econometric modelling and programming
- The Stata Journal is a quarterly publication of peer-reviewed articles, tips, and notes on various applications in Stata
  - This is one of the best resources for keeping abreast of modern econometric/analytical techniques and how they can be implemented in Stata
  - The topics dealt with range from simple trips and ticks that can be used to improve workflow in Stata to involved applications of econometric techniques at the forefront of the discipline
  - A subscription is required to gain access to the most recent issues of the journal, but articles from back issues can often be accessed for free

Stata Press publications and books about Stata

Further resources for learning Stata available from www.stata.com

- A number of excellent books on Stata and statistics can be purchase from the Stata Bookstore
  - These books cover a wide array of topics and applications ranging from basic data management and workflow in Stata to advanced econometric modelling and programming
- The Stata Journal is a quarterly publication of peer-reviewed articles, tips, and notes on various applications in Stata
  - This is one of the best resources for keeping abreast of modern econometric/analytical techniques and how they can be implemented in Stata
  - The topics dealt with range from simple tips and ticks that can be used to improve workflow in Stata to involved applications of econometric techniques at the forefront of the discipline
  - A subscription is required to gain access to the most recent issues of the journal, but articles from back issues can often be accessed for free

NetCourses

Training on Stata, by Stata

- StataCorp offers a number of web-based courses for learning Stata
- These NetCourses are comprehensive and are offered at various levels and different times throughout the year (schedule)
- Available course include
  - NetCourse 101, Introduction to Stata
  - NetCourse 151, Introduction to Stata Programming
  - NetCourse 152, Advanced Stata Programming
  - NetCourse 461, Introduction to Univariate Time Series with Stata
  - NetCourse 471, Introduction to Panel Data Using Stata
  - NetCourse 631, Introduction to Survival Analysis Using Stata

Availability of online resources

Excellent Stata training for free

- There are many excellent third-party resources for learning Stata programming freely available online
  - These resources vary in terms of their coverage, comprehensiveness, focus, level, target audience, and structure
  - Taken together, they are an incredibly useful for learning Stata and should be more than sufficient for gaining familiarity with the vast majority of applications most users (including advanced users) may be interested in
- Many of these free resources have proved invaluable in my own understanding and command of Stata
- I strongly encourage all interested users to exploit the availability of the resources listed below

Some epic free resources I

14 seriously awesome resources for learning Stata, in no particular order (Trust me, these are all fantastic)

- The list below includes my favourite free, third-party resources for learning Stata, but is by no means an exhaustive list of what is available online

1. Germán Rodríguez's *Stata Tutorial* at Princeton University (introductory - advanced)
    - Well-structured overview of Stata's interface and workflow, data management, graphics, and programming with good examples
2. *Resources to help you learn and use Stata* by the Institute for Digital Research and Education at UCLA (intro-advanced)
    - One of the most comprehensive resources for learning both basic and advanced Stata
3. *Introduction to Stata* by the Carolina Population Center at the University of North Carolina (introductory - intermediate)

Some epic free resources II

14 seriously awesome resources for learning Stata, in no particular order (Trust me, these are all fantastic)

- ▹ Introduction to basic and most frequently used commands in Stata and illustration of their uses

4. *Stata Programming Essentials* by the Social Science Computing Cooperative at UW-Madison (intermediate)
   - ▹ Good overview of and introduction to programming concepts (macros, loops, etc)
5. *Stataman's The Stata Project-Oriented Guide* (intro-advanced)
   - ▹ As the name implies, this site offers a project-oriented approach to learning data manipulation, results manipulation, command automation, and results presentation in Stata
6. Alexander C. Lembecke's *Advanced Stata Topics* notes at the London School of Economics (advanced)
   - ▹ One of the best introductions (brief, yet thourough) to programming basics, syntax, maximum likelihood methods, and Mata that I have seen. This is one of my permanent reference resources

Some epic free resources III

14 seriously awesome resources for learning Stata, in no particular order (Trust me, these are all fantastic)

7. The *Stata Daily* blog (basic - advanced)
   - Great resource for creative and powerful segments of Stata code and diverse applications
8. The Stata section of Francis Smart's *Econometrics by Simulation* blog (intermediate - advanced)
   - Contains many snippets of Stata code on advanced topics including programming, simulation, and Mata
9. Ulrich Kohler's *Introduction to Programming Stata* notes (advanced)
   - Very concise treatment of programming in Stata that covers a lot of useful code in only 24 pages (not for those who require an explanation of every single line of code)
10. Florian Wendespiess Chávez Juárez's *A guide to Stata* notes (intro - intermediate)
    - Very good notes with excellent explanations covering a lot of ground in only 64 pages. Useful for beginner and advanced users alike.

Some epic free resources IV

14 seriously awesome resources for learning Stata, in no particular order (Trust me, these are all fantastic)

11. Anything by Christopher F Baum (e.g.) at Boston College (Intermediate - Advanced)
    - Christopher is the author of An Introduction to Stata Programming. Many of the topics dealt with in this book are also given treatment in his lecture slides which can be found via a simple search in Google. Some of the most novel and creative programs and tricks can be found in these slides
12. Kurt Schmidheiny's Coding with Mata in Stata notes (advanced)
    - Excellent (my favourite) introduction to Mata (Stata's matrix programming language) and permanent personal reference
13. William Gould's Mata, the missing manual notes (advanced)
    - Good introduction to incorporating Mata into do-files and ado programs and particularly useful for learning how to define Mata functions
14. YouTube (basic - advanced)

Some epic free resources V

14 seriously awesome resources for learning Stata, in no particular order (Trust me, these are all fantastic)

- Increasingly, Stata users (and StataCorp) are uploading video tutorials to YouTube. These tutorials vary in quality, usefulness, and scope, but are useful for those who prefer the screencast medium

Thank you

References I

Baum, C. F. (2005). SUGUK 2005 invited lecture: A little bit of Stata programming goes a long way... Lecture, Boston College.

Driver, S. (2005). Stata tip 18: Making keys functional. *The Stata Journal* **5**(1), pp. 137 -- 138.

Schechter, C. (2011). Stata tip 99: Taking extra care with encode. *The Stata Journal* **11**(2), pp. 321 -- 322.

StataCorp (2013*a*). *Getting Started Stata for Windows Release 13*. StataCorp, College Station, TX: Stata Press.

--------- (2013*b*). *Stata Programming Reference Manual Release 13*. StataCorp, College Station, TX: Stata Press.
[online] available from `http://www.stata.com/manuals13/p.pdf`

--------- (2013*c*). *Stata User's Guide Release 13*. StataCorp, College Station, TX: Stata Press.
[online] available from `http://www.stata.com/manuals13/pmacro.pdf`