An Introduction to Stata Programming

Christopher F. Baum
Boston College

A Stata Press Publication
StataCorp LP
College Station, Texas

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in LaTeX 2ε
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN-10: 1-59718-045-9
ISBN-13: 978-1-59718-045-0

(Pages omitted)

Contents

(Pages omitted)

Preface

This book is a concise introduction to the art of Stata programming. It covers three types of programming that can be used in working with Stata: do-file programming, ado-file programming, and Mata functions that work in conjunction with do- and ado-files. Its emphasis is on the automation of your work with Stata and how programming on one or more of these levels can help you use Stata more effectively.

In the development of these concepts, I do not assume that you have prior experience with Stata programming, although familiarity with the command-line interface is helpful. Examples are drawn from several disciplines, although my background as an applied econometrician is evident in the selection of some sample problems. The introductory chapter motivates the *why*: why should you invest time and effort into learning Stata programming? In chapter 2, I discuss elementary concepts of the command-line interface and describe some commonly used tools for working with programs and datasets.

The format of the book may be unfamiliar to readers who have some familiarity with other books that help you learn how to use Stata. Beginning with chapter 3, each odd-numbered chapter is followed by a "cookbook" chapter containing several "recipes", 40 in total. Each recipe poses a problem: how can I perform a certain task with Stata programming? The recipe then provides a complete solution to the problem and describes how the features presented in the previous chapter can be put to good use. As in the kitchen, you may not want to follow a recipe exactly from the cookbook; just as in cuisine, a minor variation on the recipe may meet your needs, or the techniques presented in that recipe can help you see how Stata programming applies to your specific problem.

Most Stata users who delve into programming make use of do-files to automate and document their work. Consequently, the major focus of the book is do-file programming, covered in chapters 3, 5, 7, and 9. Some users will find that writing formal Stata programs, or ado-files, meets their needs. Chapter 11 is a concise summary of ado-file programming, with the following cookbook chapter presenting several recipes that contain developed ado-files. Stata's matrix programming language, Mata, can also be helpful in automating certain tasks. Chapter 13 presents a summary of Mata concepts and the key features that allow interchange of variables, scalars, macros, and matrices. The last chapter presents several examples of Mata functions developed to work with ado-files. All the do-files, ado-files, Mata functions, and datasets used in the book's examples and recipes are available from the Stata Press web site, as discussed in *Notation and typography*.

(Pages omitted)

3 Do-file programming: Functions, macros, scalars, and matrices

3.1 Introduction

This chapter describes several elements of *do-file programming*: functions used to generate new variables; macros that store individual results; and lists, scalars, and matrices. Although functions will be familiar to all users of Stata, macros and scalars are often overlooked by interactive users. Because nearly all Stata commands return results in the form of macros and scalars, familiarity with these concepts is useful.

The first section of the chapter deals with several general details: varlists, numlists, `if` *exp* and `in` *range* qualifiers, missing data handling, and string-to-numeric conversion (and vice versa). Subsequent sections present functions for `generate`, functions for `egen` ([D] **egen**), computation with a `by` *varlist*`:`, and an introduction to macros, scalars, and matrices.

3.1.1 What you should learn from this chapter

- Understand varlists, numlists, and `if` and `in` qualifiers
- Know how to handle missing data and conversion of values to missing and vice versa
- Understand string-to-numeric conversion and vice versa
- Be familiar with functions for use with `generate`
- Understand how to recode discrete and continuous variables
- Be familiar with the capabilities of `egen` functions
- Know how to use by-groups effectively
- Understand the use of `local` and `global` macros
- Be familiar with extended macro functions and macro list functions
- Understand how to use numeric and string scalars
- Know how to use matrices to retrieve and store results

3.2 Some general programming details

In this section, we use the `census2c` dataset of U.S. state-level statistics to illustrate details of do-file programming:

```
. use census2c
(1980 Census data for NE and NC states)
. list, sep(0)
```

	state	region	pop	popurb	medage	marr	divr
1.	Connecticut	NE	3107.6	2449.8	32.00	26.0	13.5
2.	Illinois	N Cntrl	11426.5	9518.0	29.90	109.8	51.0
3.	Indiana	N Cntrl	5490.2	3525.3	29.20	57.9	40.0
4.	Iowa	N Cntrl	2913.8	1708.2	30.00	27.5	11.9
5.	Kansas	N Cntrl	2363.7	1575.9	30.10	24.8	13.4
6.	Maine	NE	1124.7	534.1	30.40	12.0	6.2
7.	Massachusetts	NE	5737.0	4808.3	31.20	46.3	17.9
8.	Michigan	N Cntrl	9262.1	6551.6	28.80	86.9	45.0
9.	Minnesota	N Cntrl	4076.0	2725.2	29.20	37.6	15.4
10.	Missouri	N Cntrl	4916.7	3349.6	30.90	54.6	27.6
11.	Nebraska	N Cntrl	1569.8	987.9	29.70	14.2	6.4
12.	New Hampshire	NE	920.6	480.3	30.10	9.3	5.3
13.	New Jersey	NE	7364.8	6557.4	32.20	55.8	27.8
14.	New York	NE	17558.1	14858.1	31.90	144.5	62.0
15.	N. Dakota	N Cntrl	652.7	318.3	28.30	6.1	2.1
16.	Ohio	N Cntrl	10797.6	7918.3	29.90	99.8	58.8
17.	Pennsylvania	NE	11863.9	8220.9	32.10	93.7	34.9
18.	Rhode Island	NE	947.2	824.0	31.80	7.5	3.6
19.	S. Dakota	N Cntrl	690.8	320.8	28.90	8.8	2.8
20.	Vermont	NE	511.5	172.7	29.40	5.2	2.6
21.	Wisconsin	N Cntrl	4705.8	3020.7	29.40	41.1	17.5

This dataset, `census2c`, is arranged in tabular format, similarly to a spreadsheet. The table rows are the *observations*, cases, or records. The columns are the Stata *variables*, or fields. We see that there are 21 rows, each corresponding to one U.S. state in the North East or North Central regions, and seven columns, or variables: `state`, `region`, `pop`, `popurb`, `medage`, `marr`, and `divr`. The variables `pop` and `popurb` represent each state's 1980 population and urbanized population, respectively, in thousands. The variable `medage`, median age, is measured in years, while the variables `marr` and `divr` represent the number of marriages and divorces, respectively, in thousands.

The Stata variable names must be distinct and follow certain rules of syntax. For instance, they cannot contain embedded spaces, hyphens (`-`), or characters outside the sets `A-Z`, `a-z`, `0-9`, and `_` . In particular, a full stop, or period (`.`), cannot appear within a variable name. Variable names must start with a letter or an underscore. Most importantly, *case matters*: `STATE`, `State`, and `state` are three different variables to Stata. The Stata convention, which I urge you to adopt, is to use lowercase names for all variables to avoid confusion and to use uppercase only for some special reason. You can always use variable labels to hold additional information.

3.2.1 The varlist

Many Stata commands accept a varlist, a list of one or more variables to be used. A varlist can contain the variable names, or you can use a wild card (*), such as in `*id`. The `*` will stand in for an arbitrary set of characters. In the `census2c` dataset, `pop*` will refer to both `pop` and `popurb`:

```
. summarize pop*
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         pop |         21    5142.903    4675.152    511.456   17558.07
      popurb |         21    3829.776    3851.458    172.735   14858.07
```

A varlist can also contain a hyphenated list, such as `dose1-dose4`. This hyphenated list refers to all variables in the dataset between `dose1` and `dose4`, including those two, in the order the variables appear in the dataset. The order of variables is provided by `describe` and is shown in the Variables window. It can be modified by the `order` command.

3.2.2 The numlist

Many Stata commands require the use of a numlist, a list of numeric arguments. A numlist can be provided in several ways. It can be spelled out explicitly, as in `0.5 1.0 1.5`. It may involve a range of values, such as `1/4` or `-3/3`; these lists would include the integers between those limits. You could also specify `10 15 to 30`, which would count from 10 to 30 by 5s, or you could use a colon to say the same thing: `10 15:30`. You can count by steps, as in `1(2)9`, which is a list of the first five odd integers, or `9(-2)1`, which is the same list in reverse order. Square brackets can be used in place of parentheses.

One thing that generally should not appear in a numlist is a comma. A comma in a numlist will usually cause a syntax error. Other programming languages' loop constructs often spell out a range with an expression, such as `1,10`. In Stata, such an expression will involve a numlist of `1/10`. One of the primary uses of the numlist is for the `forvalues` ([P] **forvalues**) statement, which is described in section 7.3 (but not all valid numlists are acceptable in `forvalues`).

3.2.3 The if exp and in range qualifiers

Stata commands operate on all the observations in memory by default. Almost all Stata commands accept *qualifiers*: `if` *exp* and `in` *range* clauses that restrict the command to a subset of the observations. If we wanted to apply a transformation to a subset of the dataset or wanted to `list` ([D] **list**) only certain observations or `summarize` only those observations that met some criterion, we would use an `if` *exp* or an `in` *range* clause on the command.

In many problems, the desired subset of the data is not defined in terms of observation numbers (as specified with `in` *range*) but in terms of some logical condition. Then it is more useful to use the `if` *exp* qualifier. We could, of course, use `if` *exp* to express an `in` *range* condition. But the most common use of `if` *exp* involves the transformation of data or the specification of a statistical procedure for a subset of data identified by `if` *exp* as a logical condition. Here are some examples to illustrate these qualifiers:

```
. list state pop in 1/5
```

	state	pop
1.	Connecticut	3107.6
2.	Illinois	11426.5
3.	Indiana	5490.2
4.	Iowa	2913.8
5.	Kansas	2363.7

```
. list state pop medage if medage >= 32
```

	state	pop	medage
1.	Connecticut	3107.6	32.00
13.	New Jersey	7364.8	32.20
17.	Pennsylvania	11863.9	32.10

3.2.4 Missing data handling

Stata possesses 27 numeric missing value codes: the system missing value `.` and 26 others from `.a` through `.z`. They are treated as large positive values, and they sort in that order; plain `.` is the smallest missing value (see [U] **12.2.1 Missing values**). This allows qualifiers such as `if` *variable* `<.` to exclude all possible missing values.[1] To make your code as readable as possible, use the `missing()` ([D] **functions**) function described below.

Stata's standard practice for missing data handling is to omit those observations from any computation. For `generate` or `replace`, missing values are typically propagated so that any function of missing data is missing. In univariate statistical computations (such as `summarize`) computing a mean or standard deviation, only nonmissing cases are considered. For multivariate statistical commands, Stata generally practices *casewise deletion*, which is when an observation in which any variable is missing is deleted from the computation. The `missing(`x_1`,`x_2`,...,`x_n`)` function returns 1 if any of the arguments are missing, and 0 otherwise; that is, it provides the user with a casewise deletion indicator.

1. Before version 8, Stata user code often used qualifiers like `if` *variable* `!= .` to rule out missing values. That is now dangerous practice because that qualifier will capture only the `.` missing data code. If any of the additional codes are present in the data (for instance, by virtue of having used Stat/Transfer to convert an SPSS or SAS dataset to Stata format), they will be handled properly only when `if` *variable* `<.` or `if !missing(`*variable*`)` is used.

Several Stata commands handle missing data in nonstandard ways. The functions `max()` and `min()` and the `egen` rowwise functions (`rowmax()`, `rowmean()`, `rowmin()`, `rowsd()`, and `rowtotal()`) all ignore missing values (see section 3.4). For example, `rowmean(x1,x2,x3)` will compute the mean of three, two, or one of the variables, returning missing only if all three variables' values are missing for that observation. The `egen` functions `rownonmiss()` and `rowmiss()` return, respectively, the number of nonmissing and missing elements in their varlists. Although `correlate` *varlist* ([R] **correlate**) uses casewise deletion to remove any observation containing missing values in any variable of the varlist from the computation of the correlation matrix, the alternative command `pwcorr` computes pairwise correlations using all the available data for each pair of variables.

We have discussed missing values in numeric variables, but Stata also provides for missing values in string variables. The empty, or null, string (`""`) is taken as missing. There is an important difference in Stata between a string variable containing one or more spaces and a string variable containing no spaces (although they will appear identical to the naked eye). This suggests that you should not include one or more spaces as a possible value of a string variable; take care if you do.

Recoding missing values: The mvdecode and mvencode commands

When importing data from another statistical package, spreadsheet, or database, differing notions of missing data codes can hinder the proper rendition of the data within Stata. Likewise, if the data are to be used in another program that does not use the `.` notation for missing data codes, there may be a need to use an alternative representation of Stata's missing data. The `mvdecode` and `mvencode` commands (see [D] **mvencode**) can be useful in those circumstances. The `mvdecode` command permits you to recode various numeric values to missing, as would be appropriate when missing data have been represented as $-99, -999,\ 0.001$, and so on. Stata's full set of 27 numeric missing data codes can be used, so that -9 can be mapped to `.a`, -99 can be mapped to `.b`, etc. The `mvencode` command provides the inverse function, allowing Stata's missing values to be revised to numeric form. Like `mvdecode`, `mvencode` can map each of the 27 numeric missing data codes to a different numeric value.

Many of the thorny details involved with the reliable transfer of missing data values between packages are handled competently by Stat/Transfer. This third-party application (remarketed by StataCorp) can handle the transfer of variable and value labels between major statistical packages and can create subsets of files' contents (e.g., only selected variables are translated into the target format); it is well worth the cost for those researchers who frequently import or export datasets.

3.2.5 String-to-numeric conversion and vice versa

Stata has two major kinds of variables: string and numeric. Quite commonly, a variable imported from an external source will be misclassified as string when it should be

considered as numeric. For instance, if the first value read by `insheet` is `NA`, that variable will be classified as a string variable. Stata provides several methods for converting string variables to numeric.

First, if the variable has merely been misclassified as string, you can apply the brute force approach of the `real()` function, e.g., `generate patid = real(patientid)`. This will create missing values for any observations that cannot be interpreted as numeric.

Second, a more subtle approach is given by the `destring` command, which can transform variables in place (with the `replace` option) and can be used with a varlist to apply the same transformation to an entire set of variables with one command. This is useful if there are several variables that require conversion. However, `destring` should be used only for variables that have genuine numeric content but happen to have been misclassified as string variables.

Third, if the variable truly has string content and you need a numeric equivalent, you can use the `encode` command. You should not apply `encode` to a string variable that has purely numeric content (for instance, one that has been misclassified as a string variable) because `encode` will attempt to create a value label for each distinct value of the variable. As an example, we create a numeric equivalent of the `state` variable:

```
. encode state, generate(stateid)
. describe state stateid
              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------------------------
state           str13  %-13s                  State
stateid         long   %13.0g      stateid    State
. list state stateid in 1/5

     +-------------------------+
     | state           stateid |
     |-------------------------|
  1. | Connecticut   Connecticut |
  2. | Illinois         Illinois |
  3. | Indiana           Indiana |
  4. | Iowa                 Iowa |
  5. | Kansas             Kansas |
     +-------------------------+
```

Although `stateid` is numeric, it has automatically been given the value label of the values of `state`. To see the numeric values, use `list` with the `nolabel` option.

Numeric-to-string conversion

You may also need to generate the string equivalent of a numeric variable. Often it is easier to parse the contents of string variables and extract substrings that may have some particular significance. Such transformations can be applied to integer numeric variables by means of integer division and remainders, but these transformations are generally more cumbersome and error-prone. The limits to exact representation of

numeric values, such as integers, with many digits are circumvented by placing those values in string form. A thorough discussion of these issues is given in Cox (2002c).

We discussed three methods for string-to-numeric conversion. For each method, the inverse function or command is available for numeric-to-string conversion: the `string()` function, `tostring`, and `decode`. The `string()` function is useful in allowing a numeric display format ([D] **format**) to be used. This would allow, for instance, the creation of a variable with leading zeros, which are integral in some ID-number schemes. The `tostring` command provides a more comprehensive approach: it contains various safeguards to prevent the loss of information and can be used with a particular display format. Like `destring`, `tostring` can be applied to a varlist to alter an entire set of variables.

A common task is the restoration of leading zeros in a variable that has been transferred from a spreadsheet. For instance, U.S. zip (postal) codes and Social Security numbers can start with zero. The `tostring` command is useful here. Say, for example, that we have a variable, `zip`:

```
tostring zip, format(%05.0f) generate(zipstring)
```

The variable `zipstring` will contain strings of five-digit numbers with leading zeros included, as specified by the `format()` option.

To illustrate `decode`, let's say that you have the `stateid` numeric variable defined above (with its value label) in your dataset but you do not have the variable in string form. You can create the string variable `statename` by using `decode`:

```
. decode stateid, generate(statename)
. list stateid statename in 1/5
```

	stateid	statename
1.	Connecticut	Connecticut
2.	Illinois	Illinois
3.	Indiana	Indiana
4.	Iowa	Iowa
5.	Kansas	Kansas

To use `decode`, the numeric variable to be decoded must have a value label.

Working with quoted strings

You may be aware that `display "this is a quoted string"` will display the contents of that quoted string. What happens, though, if your string itself contains quotation marks? Then you must resort to *compound double quotes*. A command such as

```
. display ‘"This is a "quoted" string."’
```

will properly display the string, with the inner quotation marks intact. If ordinary double quotes are used, Stata will produce an error message. Compound double quotes can often be used advantageously when there is any possibility that the contents of string variables might include quotation marks.

3.3 Functions for the generate command

The fundamental commands for data transformation are `generate` and `replace`. They function in the same way, but two rules govern their use. `generate` can be used only to create a *new* variable, one whose name is not currently in use. On the other hand, `replace` can be used only to revise the contents of an *existing* variable. Unlike other Stata commands whose names can be abbreviated, `replace` must be spelled out for safety's sake.

We illustrate the use of `generate` by creating a new variable in our dataset that measures the fraction of each state's population living in urban areas in 1980. We need only specify the appropriate formula, and Stata will automatically apply that formula to every observation that is specified by the `generate` command, using the rules of algebra. For instance, if the formula would result in a division by zero for a given state, the result for that state would be flagged as missing. We generate the fraction, `urbanized`, and use the `summarize` command to display its descriptive statistics:

```
. generate urbanized = popurb / pop
. summarize urbanized
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
   urbanized |         21    .6667691    .1500842   .3377319   .8903645
```

We see that the average state in this part of the United States is 66.7% urbanized, with that fraction ranging from 34% to 89%.

If the `urbanized` variable already existed, but we wanted to express it as a percentage rather than a decimal fraction, we must use `replace`:

```
. replace urbanized = 100 * urbanized
(21 real changes made)
. summarize urbanized
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
   urbanized |         21    66.67691    15.00843   33.77319   89.03645
```

`replace` reports the number of changes it made; here it changed all 21 observations.

The concern for efficiency of a do-file is first a concern for *human* efficiency. You should write the data transformations as a simple, succinct set of commands that can readily be audited and modified. You may find that there are several ways to create the same variable by using `generate` and `replace`. It is usually best to stick with the simplest and clearest form of these statements.

A variety of useful functions are located in Stata's programming functions category (`help programming functions` or [D] **functions**). For instance, several `replace` statements might themselves be replaced with one call to the `inlist()` or `inrange()` functions (see Cox [2006c]). The former will allow the specification of a variable and a list of values. It returns 1 for each observation if the variable matches one of the elements of the list, and 0 otherwise. The function can be applied to either numeric or string variables. For string variables, up to 10 string values can be specified in the list. For example,

```
. generate byte newengland = inlist(state, "Connecticut", "Maine",
> "Massachusetts", "New Hampshire", "Rhode Island", "Vermont")
. sort medage
. list state medage pop if newengland, sep(0)
```

	state	medage	pop
6.	Vermont	29.40	511.5
13.	New Hampshire	30.10	920.6
14.	Maine	30.40	1124.7
16.	Massachusetts	31.20	5737.0
17.	Rhode Island	31.80	947.2
19.	Connecticut	32.00	3107.6

The `inrange()` function allows the specification of a variable and an interval on the real line and returns 1 or 0 to indicate whether the variable's values fall within the interval (which can be open, i.e., one limit can be $\pm\infty$). For example,

```
. list state medage pop if inrange(pop, 5000, 9999), sep(0)
```

	state	medage	pop
2.	Michigan	28.80	9262.1
4.	Indiana	29.20	5490.2
16.	Massachusetts	31.20	5737.0
21.	New Jersey	32.20	7364.8

Several data transformations involve the use of *integer division*, that is, truncating the remainder. For instance, four-digit U.S. Standard Industrial Classification (SIC) codes 3211–3299 divided by 100 must each yield 32. This is accomplished with the `int()` function (defined in `help math functions`).[2] A common task involves extracting one or more digits from an integer code; for instance, the third and fourth digits of the codes above can be defined as

```
generate digit34 = SIC - int(SIC / 100) * 100
```

or

```
generate mod34 = mod(SIC,100)
```

2. Also see the discussion of the `floor()` and `ceil()` functions in section 3.3.3.

where the second construct makes use of the modulo (`mod()`) function (see Cox [2007d]). The third digit alone could be extracted with

```
generate digit3 = int((SIC - int(SIC / 100) * 100) / 10)
```

or

```
generate mod3 = (mod(SIC, 100) - mod(SIC, 10)) / 10
```

or even

```
generate sub3 = real(substr(string(SIC), 3, 1))
```

using the `string()` function to express `SIC` as a string, the `substr()` function to extract the desired piece of that string, and the `real()` function to convert the extracted string into numeric form.

As discussed in section 2.4, you should realize the limitations of this method in dealing with very long integers, such as U.S. Social Security numbers of nine digits or ID codes of 10 or 12 digits. The functions `maxbyte()`, `maxint()`, and `maxlong()` are useful here. An excellent discussion of these issues is given in Cox (2002c).

Lastly, we must mention one exceedingly useful function for `generate`: the `sum()` function, which produces cumulative or running sums. That capability is useful in the context of time-series data, where it can be used to convert a flow or other rate variable into a stock or other amount variable. If we have an initial capital stock value and a net investment series, the `sum()` of investment plus the initial capital stock defines the capital stock at each point in time. This function does not place the single sum of the series into the new variable. If that is what you want, use the `egen` function `total()`.

3.3.1 Using if exp with indicator variables

A key element of many empirical research projects is the *indicator variable*: a variable taking on the values $(0, 1)$ to indicate whether a particular condition is satisfied. These are also commonly known as *dummy variables* or *Boolean variables*. The creation of indicator variables is best accomplished by using a *Boolean condition*: an expression that evaluates to true or false for each observation. The `if` *exp* qualifier has an important role here as well. Using our dataset, it would be possible to generate indicator variables for small and large states with the following commands. As I note below, we must take care to handle potentially missing values by using the `missing()` function.

```
. generate smallpop = 0
. replace smallpop = 1 if pop <= 5000 & !missing(pop)
(13 real changes made)
. generate largepop = 0
. replace largepop = 1 if pop > 5000 & !missing(pop)
(8 real changes made)
```