13 Functions and expressions

Contents

13.1 Overview
13.2 Operators
  13.2.1 Arithmetic operators
  13.2.2 String operators
  13.2.3 Relational operators
  13.2.4 Logical operators
  13.2.5 Order of evaluation, all operators
13.3 Functions
13.4 System variables (_variables)
13.5 Accessing coefficients and standard errors
  13.5.1 Single-equation models
  13.5.2 Multiple-equation models
  13.5.3 Factor variables and time-series operators
13.6 Accessing results from Stata commands
13.7 Explicit subscripting
  13.7.1 Generating lags and leads
  13.7.2 Subscripting within groups
13.8 Using the Expression Builder
13.9 Indicator values for levels of factor variables
13.10 Time-series operators
  13.10.1 Generating lags, leads, and differences
  13.10.2 Time-series operators and factor variables
  13.10.3 Operators within groups
  13.10.4 Video example
13.11 Label values
13.12 Precision and problems therein
13.13 References

If you have not read **[U] 11 Language syntax**, please do so before reading this entry.

13.1 Overview

Examples of expressions include

```
2+2
miles/gallons
myv+2/oth
(myv+2)/oth
ln(income)
age<25 & income>50000
age<25 | income>50000
age==25
name=="M Brown"
fname + " " + lname
substr(name,1,10)
val[_n-1]
L.gnp
```

Expressions like those above are allowed anywhere *exp* appears in a syntax diagram. One example is [D] **generate**:

`generate` *newvar* = *exp* [*if*] [*in*]

The first *exp* specifies the contents of the new variable, and the optional second expression restricts the subsample over which it is to be defined. Another is [R] **summarize**:

`summarize` [*varlist*] [*if*] [*in*]

The optional expression restricts the sample over which summary statistics are calculated.

Algebraic and string expressions are specified in a natural way using the standard rules of hierarchy. You may use parentheses freely to force a different order of evaluation.

▷ Example 1

`myv+2/oth` is interpreted as `myv+(2/oth)`. If you wanted to change the order of the evaluation, you could type `(myv+2)/oth`.

◁

13.2 Operators

Stata has four different classes of operators: arithmetic, string, relational, and logical. Each type is discussed below.

13.2.1 Arithmetic operators

The *arithmetic operators* in Stata are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `^` (raise to a power), and the prefix `-` (negation). Any arithmetic operation on a missing value or an impossible arithmetic operation (such as division by zero) yields a missing value.

▷ Example 2

The expression `-(`x`+`y`^(`x`-`y`))/(`x`*`y`)` denotes the formula

$$-\frac{x + y^{x-y}}{x \cdot y}$$

and evaluates to *missing* if x or y is missing or zero.

◁

13.2.2 String operators

The `+` and `*` signs are also used as string operators.

`+` is used for the concatenation of two strings. Stata determines by context whether `+` means addition or concatenation. If `+` appears between two numeric values, Stata adds them. If `+` appears between two strings, Stata concatenates them.

▷ Example 3

The expression `"this"+"that"` results in the string `"thisthat"`, whereas the expression `2+3` results in the number `5`. Stata issues the error message “type mismatch” if the arguments on either side of the `+` sign are not of the same type. Thus the expression `2+"this"` is an error, as is `2+"3"`.

The expressions on either side of the `+` can be arbitrarily complex:

```
substr(string(20+2),1,1) + strupper(substr("rf",1+1,1))
```

The result of the above expression is the string `"2F"`. See [FN] **String functions** for a description of the `substr()`, `string()`, and `strupper()` functions.

◁

`*` is used to duplicate a string 0 or more times. Stata determines by context whether `*` means multiplication or string duplication. If `*` appears between two numeric values, Stata multiplies them. If `*` appears between a string and a numeric value, Stata duplicates the string as many times as the numeric value indicates.

▷ Example 4

The expression `"this"*3` results in the string `"thisthisthis"`, whereas the expression `2*3` results in the number `6`. Stata issues the error message “type mismatch” if the arguments on either side of the `*` sign are both strings. Thus the expression `"this"*"that"` is an error.

As with string concatenation above, the arguments can be arbitrarily complex.

◁

13.2.3 Relational operators

The *relational operators* are `>` (greater than), `<` (less than), `>=` (greater than or equal), `<=` (less than or equal), `==` (equal), and `!=` (not equal). Observe that the relational operator for equality is a pair of equal signs. This convention distinguishes relational equality from the `=`*exp* assignment phrase.

❑ Technical note

You may use `~` anywhere `!` would be appropriate to represent the logical operator "not". Thus the not-equal operator may also be written as `~=`.

❑

Relational expressions are either *true* or *false*. Relational operators may be used on either numeric or string subexpressions; thus, the expression `3>2` is *true*, as is `"zebra">"cat"`. In the latter case, the relation merely indicates that `"zebra"` comes after the word `"cat"` in the dictionary. All uppercase letters precede all lowercase letters in Stata's book, so `"cat">"Zebra"` is also *true*.

Missing values may appear in relational expressions. If `x` were a numeric variable, the expression `x>=.` is *true* if `x` is missing and *false* otherwise. A missing value is greater than any nonmissing value; see [U] **12.2.1 Missing values**.

▷ Example 5

You have data on `age` and `income` and wish to list the subset of the data for persons aged 25 years or less. You could type

```
. list if age<=25
```

If you wanted to list the subset of data of persons aged exactly 25, you would type

```
. list if age==25
```

Note the double equal sign. It would be an error to type `list if age=25`.

◁

Although it is convenient to think of relational expressions as evaluating to *true* or *false*, they actually evaluate to numbers. A result of *true* is defined as 1 and *false* is defined as 0.

▷ Example 6

The definition of *true* and *false* makes it easy to create indicator, or dummy, variables. For instance,

```
generate incgt10k=income>10000
```

creates a variable that takes on the value 0 when `income` is less than or equal to $10,000, and 1 when `income` is greater than $10,000. Because missing values are greater than all nonmissing values, the new variable `incgt10k` will also take on the value 1 when `income` is *missing*. It would be safer to type

```
generate incgt10k=income>10000 if income<.
```

Now, observations in which `income` is *missing* will also contain *missing* in `incgt10k`. See [U] **25 Working with categorical data and factor variables** for more examples.

◁

❑ Technical note

Although you will rarely wish to do so, because arithmetic and relational operators both evaluate to numbers, there is no reason you cannot mix the two types of operators in one expression. For instance, `(2==2)+1` evaluates to 2, because `2==2` evaluates to 1, and $1 + 1$ is 2.

Relational operators are evaluated after all arithmetic operations. Thus the expression `(3>2)+1` is equal to 2, whereas `3>2+1` is equal to 0. Evaluating relational operators last guarantees the *logical* (as opposed to the *numeric*) interpretation. It should make sense that `3>2+1` is *false*.

❑

13.2.4 Logical operators

The *logical operators* are `&` (and), `|` (or), and `!` (not). The logical operators interpret any nonzero value (including *missing*) as *true* and zero as *false*.

▷ Example 7

If you have data on `age` and `income` and wish to `list` data for persons making more than $50,000 along with persons under the age of 25 making more than $30,000, you could type

```
list if income>50000 | income>30000 & age<25
```

The `&` takes precedence over the `|`. If you were unsure, however, you could have typed

```
list if income>50000 | (income>30000 & age<25)
```

In either case, the statement will also `list` all observations for which `income` is *missing*, because *missing* is greater than 50,000.

◁

❑ Technical note

Like relational operators, logical operators return 1 for *true* and 0 for *false*. For example, the expression `5 & .` evaluates to 1. Logical operations, except for `!`, are performed after all arithmetic and relational operations; the expression `3>2 & 5>4` is interpreted as `(3>2) & (5>4)` and evaluates to 1.

❑

13.2.5 Order of evaluation, all operators

The order of evaluation (from first to last) of all operators is `!` (or `~`), `^`, `-` (negation), `/`, `*`, `-` (subtraction), `+`, `!=` (or `~=`), `>`, `<`, `<=`, `>=`, `==`, `&`, and `|`.

13.3 Functions

Stata provides mathematical functions, probability and density functions, matrix functions, string functions, functions for dealing with dates and time series, and a set of special functions for programmers. You can find all of these documented in the *Stata Functions Reference Manual*. Stata's matrix programming language, Mata, provides more functions and those are documented in the *Mata Reference Manual* or in the help documentation (type `help mata functions`).

Functions are merely a set of rules; you supply the function with arguments, and the function evaluates the arguments according to the rules that define the function. Because functions are essentially subroutines that evaluate arguments and cause no action on their own, functions must be used in conjunction with a Stata command. Functions are indicated by the function name, an open parenthesis, an expression or expressions separated by commas, and a close parenthesis.

For example,

```
. display sqrt(4)
2
```

or

```
. display sqrt(2+2)
2
```

demonstrates the simplest use of a function. Here we have used the mathematical function, `sqrt()`, which takes one number (or expression) as its argument and returns its square root. The function was used with the Stata command `display`. If we had simply typed

```
. sqrt(4)
```

Stata would have returned the error message

```
command sqrt is unrecognized
r(199);
```

Functions can operate on variables, as well. For example, suppose that you wanted to generate a random variable that has observations drawn from a lognormal distribution. You could type

```
. set obs 5
number of observations (_N) was 0, now 5
. generate y = runiform()
. replace y = invnormal(y)
(5 real changes made)
. replace y = exp(y)
(5 real changes made)
. list
```

	y
1.	.686471
2.	2.380994
3.	.2814537
4.	1.215575
5.	.2920268

You could have saved yourself some typing by typing just

```
. generate y = exp(rnormal())
```

Functions accept expressions as arguments.

All functions are defined over a specified domain and return values within a specified range. Whenever an argument is outside a function's domain, the function will return a missing value or issue an error message, whichever is most appropriate. For example, if you supplied the `log()` function with an argument of zero, the `log(0)` would return a missing value because zero is outside the natural logarithm function's domain. If you supplied the `log()` function with a string argument, Stata would issue a "type mismatch" error because `log()` is a numerical function and is undefined

for strings. If you supply an argument that evaluates to a value that is outside the function's range, the function will return a missing value. Whenever a function accepts a string as an argument, the string must be enclosed in double quotes, unless you provide the name of a variable that has a string storage type.

13.4 System variables (_variables)

Expressions may also contain *_variables* (pronounced "underscore variables"), which are built-in system variables that are created and updated by Stata. They are called *_variables* because their names all begin with the underscore character, '_'.

The *_variables* are

[*eqno*]_b[*varname*] (synonym: [*eqno*]_coef[*varname*]) contains the value (to machine precision) of the coefficient on *varname* from the most recently fit model (such as ANOVA, regression, Cox, logit, probit, and multinomial logit). See **[U] 13.5 Accessing coefficients and standard errors** below for a complete description.

_cons is always equal to the number 1 when used directly and refers to the intercept term when used indirectly, as in _b[_cons].

_n contains the number of the current observation.

_N contains the total number of observations in the dataset or the number of observations in the current by() group.

_rc contains the value of the return code from the most recent capture command.

[*eqno*]_se[*varname*] contains the value (to machine precision) of the standard error of the coefficient on *varname* from the most recently fit model (such as ANOVA, regression, Cox, logit, probit, and multinomial logit). See **[U] 13.5 Accessing coefficients and standard errors** below for a complete description.

13.5 Accessing coefficients and standard errors

After fitting a model, you can access the coefficients and standard errors and use them in subsequent expressions. Also see **[R] predict** (and **[U] 20 Estimation and postestimation commands**) for an easier way to obtain predictions, residuals, and the like.

13.5.1 Single-equation models

First, let's consider estimation methods that yield one estimated equation with a one-to-one correspondence between coefficients and variables such as logit, ologit, oprobit, probit, regress, and tobit. _b[*varname*] (synonym _coef[*varname*]) contains the coefficient on *varname* and _se[*varname*] contains its standard error, and both are recorded to machine precision. Thus _b[age] refers to the calculated coefficient on the age variable after typing, say, regress response age sex, and _se[age] refers to the standard error on the coefficient. _b[_cons] refers to the constant and _se[_cons] to its standard error. Thus you might type

```
. regress response age sex
. generate asif = _b[_cons] + _b[age]*age
```

13.5.2 Multiple-equation models

The syntax for referring to coefficients and standard errors in multiple-equation models is the same as in the simple-model case, except that `_b[]` and `_se[]` are preceded by an equation number in square brackets. There are, however, many alternatives in how you may type requests. The way that you are supposed to type requests is

[*eqno*]`_b`[*varname*]
[*eqno*]`_se`[*varname*]

but you may substitute `_coef[]` for `_b[]`. In fact, you may omit the `_b[]` altogether, and most Stata users do:

[*eqno*][*varname*]

You may also omit the second pair of square brackets:

[*eqno*]*varname*

You may retain the `_b[]` or `_se[]` and insert a colon between *eqno* and *varname*:

`_b`[*eqno*`:`*varname*]

There are two ways to specify the equation number *eqno*: either as an absolute equation number or as an "indirect" equation number. In the absolute form, the number is preceded by a '`#`' sign. Thus `[#1]displ` refers to the coefficient on `displ` in the first equation (and `[#1]_se[displ]` refers to its standard error). You can even use this form for simple models, such as `regress`, if you prefer. `regress` estimates one equation, so `[#1]displ` refers to the coefficient on `displ`, just as `_b[displ]` does. Similarly, `[#1]_se[displ]` and `_se[displ]` are equivalent. The logic works both ways—in the multiple-equation context, `_b[displ]` refers to the coefficient on `displ` in the first equation and `_se[displ]` refers to its standard error. `_b`[*varname*] (`_se`[*varname*]) is just another way of saying `[#1]`*varname* (`[#1]_se`[*varname*]).

Equations may also be referred to indirectly. `[res]displ` refers to the coefficient on `displ` in the equation named `res`. Equations are often named after the corresponding dependent variable name if there is such a concept in the fitted model, so `[res]displ` might refer to the coefficient on `displ` in the equation for variable `res`.

For multinomial logit (`mlogit`), multinomial probit (`mprobit`), and similar commands, equations are named after the levels of the single dependent categorical variable. In these models, there is one dependent variable, and there is an equation corresponding to each of the outcomes (values taken on) recorded in that variable, except for the one that is taken to be the base outcome. `[res]displ` would be interpreted as the coefficient on `displ` in the equation corresponding to the outcome `res`. If outcome `res` is the base outcome, Stata treats `[res]displ` as zero (and Stata does the same for `[res]_se[displ]`).

Continuing with the multinomial outcome case: the outcome variable must be numeric. The syntax `[res]displ` would be understood only if there were a value label associated with the numeric outcome variable and `res` were one of the labels. If your data are not labeled, then you can use the usual multiple-equation syntax `[##]`*varname* and `[##]_se`[*varname*] to refer to the coefficient and standard error for variable *varname* in the #th equation.

For `mlogit`, if your data are not labeled, you can also use the syntax [*#*]*varname* and [*#*]`_se`[*varname*] (without the '`#`') to refer to the coefficient and standard error for *varname* in the equation for outcome *#*.

13.5.3 Factor variables and time-series operators

We refer to time-series–operated variables exactly as we refer to normal variables. We type the name of the variable, which for time-series–operated variables includes the operators; see [U] 11.4.4 Time-series varlists. You might type

```
. regress open L.close LD.volume
. display _b[L.close]
. display _b[LD.volume]
```

We cannot refer to factor variables such as i.group in expressions. Assuming that i.group has three levels, i.group represents three virtual indicator variables—1b.group, 2.group, and 3.group. We can refer to the indicator variables in expressions by typing, for example, _b[i2.group] or just _b[2.group]. That is to say, we include the operators and the levels of the factor variables when typing the indicator-variable name. Consider a regression using factor variables:

```
. use http://www.stata-press.com/data/r14/fvex, clear
(Artificial factor variables' data)

. regress y i.sex i.group sex#group age sex#c.age

      Source |       SS           df       MS      Number of obs   =     3,000
-------------+----------------------------------   F(7, 2992)      =     80.84
       Model |  221310.507         7  31615.7868   Prob > F        =    0.0000
    Residual |   1170122.5     2,992  391.083723   R-squared       =    0.1591
-------------+----------------------------------   Adj R-squared   =    0.1571
       Total |  1391433.01     2,999  463.965657   Root MSE        =    19.776

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         sex |
     female  |   32.29378   3.782064     8.54   0.000     24.87807    39.70949
             |
       group |
          2  |   9.477077   1.624075     5.84   0.000     6.292659    12.66149
          3  |   18.31292   1.776337    10.31   0.000     14.82995    21.79588
             |
   sex#group |
   female#2  |  -6.621804   2.021384    -3.28   0.001    -10.58525   -2.658361
   female#3  |  -10.48293      3.209    -3.27   0.001      -16.775   -4.190858
             |
         age |   -.212332   .0538345    -3.94   0.000    -.3178884   -.1067756
             |
   sex#c.age |
     female  |   -.226838   .0745707    -3.04   0.002    -.3730531   -.0806229
             |
       _cons |   60.48167   2.842955    21.27   0.000     54.90732    66.05601
------------------------------------------------------------------------------
```

If we want to use the coefficient for level 2 of group in an expression, we type _b[2.group]; for level 3, we type _b[3.group]. To refer to the coefficient of an interaction of two levels of two factor variables, we specify the interaction operator and the level of each variable. For example, to use the coefficient for sex = 1 (female) and group = 2, we type _b[1.sex#2.group]. (We determined that 1 was the level corresponding to female by typing label list.) When one of the variables in an interaction is continuous, we can either make that explicit, _b[1.sex#c.age], or we can leave off the c., _b[1.sex#age].

Referring to interactions is more challenging than referring to normal variables. It is also more challenging to refer to coefficients from estimators that use multiple equations. If you find it difficult to know what to type for a coefficient, replay your estimation results using the coeflegend option.

```
. regress, coeflegend
      Source |       SS           df       MS      Number of obs   =     3,000
-------------+----------------------------------   F(7, 2992)      =     80.84
       Model |  221310.507         7  31615.7868   Prob > F        =    0.0000
    Residual |   1170122.5     2,992  391.083723   R-squared       =    0.1591
-------------+----------------------------------   Adj R-squared   =    0.1571
       Total |  1391433.01     2,999  463.965657   Root MSE        =    19.776

------------------------------------------------------------------------------
           y |      Coef.  Legend
-------------+----------------------------------------------------------------
         sex |
     female  |   32.29378  _b[1.sex]
             |
       group |
          2  |   9.477077  _b[2.group]
          3  |   18.31292  _b[3.group]
             |
   sex#group |
   female#2  |  -6.621804  _b[1.sex#2.group]
   female#3  |  -10.48293  _b[1.sex#3.group]
             |
         age |   -.212332  _b[age]
             |
   sex#c.age |
     female  |   -.226838  _b[1.sex#c.age]
             |
       _cons |   60.48167  _b[_cons]
------------------------------------------------------------------------------
```

The `Legend` column shows you exactly what to type to refer to any coefficient in the estimation.

If your estimation results have both equations and factor variables, nothing changes from what we said in [U] **13.5.2 Multiple-equation models** above. What you type for *varname* is just a little more complicated.

13.6 Accessing results from Stata commands

Most Stata commands—not just estimation commands—store results so that you can access them in subsequent expressions. You do that by referring to `e(`*name*`)`, `r(`*name*`)`, `s(`*name*`)`, or `c(`*name*`)`.

```
. summarize age
. generate agedev = age-r(mean)
. regress mpg weight
. display "The number of observations used is " e(N)
```

Most commands are categorized as r-class, meaning that they store results in `r()`. The returned results—such as `r(mean)`—are available immediately following the command, and if you are going to refer to them, you need to refer to them soon because the next command will probably replace what is in `r()`.

e-class commands are Stata's estimation commands—commands that fit models. Results in `e()` remain available until the next model is fit.

s-class commands are parsing commands—commands used by programmers to interpret commands you type. Few commands store anything in `s()`.

There are no c-class commands. `c()` contains values that are always available, such as `c(current_date)` (today's date), `c(pwd)` (the current directory), `c(N)` (the number of observations), and so on. There are many `c()` values and they are documented in [P] **creturn**.

Every command of Stata is designated r-class, e-class, or s-class, or, if the command stores nothing, n-class. r stands for return as in returned results, e stands for estimation as in estimation results, s stands for string, and, admittedly, this last acronym is weak, n stands for null.

You can find out what is stored where by looking in the *Stored results* section for the particular command in the *Reference* manual. If you know the class of a command—and it is easy enough to guess—you can also see what is stored by typing `return list`, `ereturn list`, or `sreturn list`:

See [R] **stored results** and [U] **18.8 Accessing results calculated by other programs**.

13.7 Explicit subscripting

Individual observations on variables can be referred to by subscripting the variables. Explicit subscripts are specified by following a variable name with square brackets that contain an expression. The result of the subscript expression is truncated to an integer, and the value of the variable for the indicated observation is returned. If the value of the subscript expression is less than `1` or greater than `_N`, a missing value is returned.

13.7.1 Generating lags and leads

When you type something like

```
. generate y = x
```

Stata interprets it as if you typed

```
. generate y = x[_n]
```

which means that the first observation of `y` is to be assigned the value from the first observation of `x`, the second observation of `y` is to be assigned the value from the second observation on `x`, and so on. If you instead typed

```
. generate y = x[1]
```

you would set each observation of `y` equal to the first observation on `x`. If you typed

```
. generate y = x[2]
```

you would set each observation of `y` equal to the second observation on `x`. If you typed

```
. generate y = x[0]
```

Stata would merely copy a missing value into every observation of `y` because observation 0 does not exist. The same would happen if you typed

```
. generate y = x[100]
```

and you had fewer than 100 observations in your data.

When you type the square brackets, you are specifying explicit subscripts. Explicit subscripting combined with the *_variable* `_n` can be used to create lagged values on a variable. The lagged value of a variable `x` can be obtained by typing

```
. generate xlag = x[_n-1]
```

If you are really interested in lags and leads, you probably have time-series data and would be better served by using the time-series operators, such as `L.x`. Time-series operators can be used with varlists and expressions and they are safer because they account for gaps in the data; see [U] **11.4.4 Time-series varlists** and [U] **13.10 Time-series operators**. Even so, it is important that you understand how the above works.

The built-in underscore variable _n is understood by Stata to mean the observation number of the current observation. That is why

```
. generate y = x[_n]
```

results in observation 1 of x being copied to observation 1 of y and similarly for the rest of the observations. Consider

```
. generate xlag = x[_n-1]
```

_n-1 evaluates to the observation number of the previous observation. For the first observation, _n-1 = 0 and therefore xlag[1] is set to missing. For the second observation, _n-1 = 1 and xlag[2] is set to the value of x[1], and so on.

Similarly, the lead of x can be created by

```
. generate xlead = x[_n+1]
```

Here the last observation on the new variable xlead will be *missing* because _n+1 will be greater than _N (_N is the total number of observations in the dataset).

13.7.2 Subscripting within groups

When a command is preceded by the by *varlist*: prefix, subscript expressions and the underscore variables _n and _N are evaluated relative to the subset of the data currently being processed. For example, consider the following (admittedly not very interesting) data:

```
. use http://www.stata-press.com/data/r14/gxmpl6
. list
```

	bvar	oldvar
1.	1	1.1
2.	1	2.1
3.	1	3.1
4.	2	4.1
5.	2	5.1

To see how _n, _N, and explicit subscripting work, let's create three new variables demonstrating each and then list their values:

```
. generate small_n = _n
. generate big_n = _N
. generate newvar = oldvar[1]
. list
```

	bvar	oldvar	small_n	big_n	newvar
1.	1	1.1	1	5	1.1
2.	1	2.1	2	5	1.1
3.	1	3.1	3	5	1.1
4.	2	4.1	4	5	1.1
5.	2	5.1	5	5	1.1

small_n (which is equal to _n) goes from 1 to 5, and big_n (which is equal to _N) is 5. This should not be surprising; there are 5 observations in the data, and _n is supposed to count observations, whereas _N is the total number. newvar, which we defined as oldvar[1], is 1.1. Indeed, we see that the first observation on oldvar is 1.1.

Now, let's repeat those same three steps, only this time preceding each step with the prefix by bvar:. First, we will drop the old values of small_n, big_n, and newvar so that we start fresh:

```
. drop small_n big_n newvar
. by bvar, sort: generate small_n=_n
. by bvar: generate big_n =_N
. by bvar: generate newvar=oldvar[1]
. list
```

	bvar	oldvar	small_n	big_n	newvar
1.	1	1.1	1	3	1.1
2.	1	2.1	2	3	1.1
3.	1	3.1	3	3	1.1
4.	2	4.1	1	2	4.1
5.	2	5.1	2	2	4.1

The results are different. Remember that we claimed that _n and _N are evaluated relative to the subset of data in the by-group. Thus small_n (_n) goes from 1 to 3 for bvar = 1 and from 1 to 2 for bvar = 2. big_n (_N) is 3 for the first group and 2 for the second. Finally, newvar (oldvar[1]) is 1.1 and 4.1.

▷ Example 8

You now know enough to do some amazing things.

Suppose that you have data on individual states and you have another variable in your data called region that divides the states into the four census regions. You have a variable x in your data, and you want to make a new variable called avgx to include in your regressions. This new variable is to take on the average value of x for the region in which the state is located. Thus, for California, you will have the observation on x and the observation on the average value in the region, avgx. Here is how:

```
. by region, sort: generate avgx=sum(x)/_n
. by region: replace avgx=avgx[_N]
```

First, by region, we generate avgx equal to the running sum of x divided by the number of observations so far. The , sort ensures that the data are in region order. We have, in effect, created the running average of x within region. It is the last observation of this running average, the overall average within the region, that interests us. So, by region, we replace every avgx observation in a region with the last observation within the region, avgx[_N].

Here is what we will see when we type these commands:

```
. use http://www.stata-press.com/data/r14/gxmpl7, clear
. by region, sort: generate avgx=sum(x)/_n
. by region: replace avgx=avgx[_N]
(46 real changes made)
```

In our example, there are no missing observations on x. If there had been, we would have obtained the wrong answer. When we created the running average, we typed

```
. by region, sort: generate avgx=sum(x)/_n
```

The problem is not with the sum() function. When sum() encounters a missing, it adds zero to the sum. The problem is with _n. Let's assume that the second observation in the first region has recorded a missing for x. When Stata processes the third observation in that region, it will calculate the sum of two elements (remember that one is missing) and then divide the sum by 3 when it should be divided by 2. There is an easy solution:

```
. by region: generate avgx=sum(x)/sum(x<.)
```

Rather than divide by _n, we divide by the total number of nonmissing observations seen on x so far, namely, the sum(x<.).

If our goal were simply to obtain the mean, we could have more easily accomplished it by typing egen avgx=mean(x), by(region); see [D] **egen**. egen, however, is written in Stata, and the above is how egen's mean() function works. The general principles are worth understanding.

◁

▷ Example 9

You have some patient data recording vital signs at various times during an experiment. The variables include patient, an ID number or name of the patient; time, a variable recording the date or time or epoch of the vital-sign reading; and vital, a vital sign. You probably have more than one vital sign, but one is enough to illustrate the concept. Each observation in your data represents a patient-time combination.

Let's assume that you have 1,000 patients and, for every observation on the same patient, you want to create a new variable called orig that records the patient's initial value of this vital sign.

```
. use http://www.stata-press.com/data/r14/gxmpl8, clear
. sort patient time
. by patient: generate orig=vital[1]
```

Observe that vital[1] refers not to the first reading on the first patient but to the first reading on the current patient, because we are performing the generate command by patient.

◁

▷ Example 10

Let's do one more example with these patient data. Suppose that we want to create a new dataset from our patient data that record not only the patient's identification, the time of the reading of the first vital sign, and the first vital sign reading itself, but also the time of the reading of the last vital sign and its value. We want 1 observation per patient. Here's how:

```
. sort patient time
. by patient: generate lasttime=time[_N]
. by patient: generate lastvital=vital[_N]
. by patient: drop if _n!=1
```

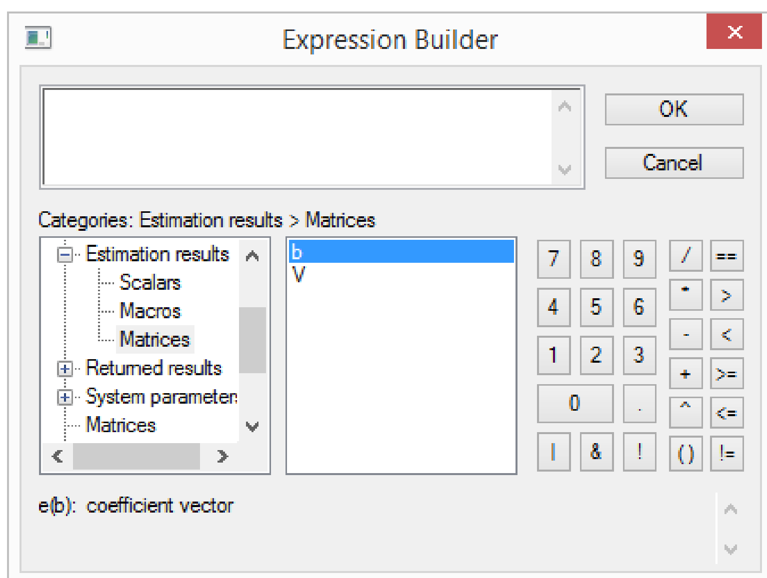
◁

See Mitchell (2010, chap. 7) for numerous examples of subscripting and subscripting within groups

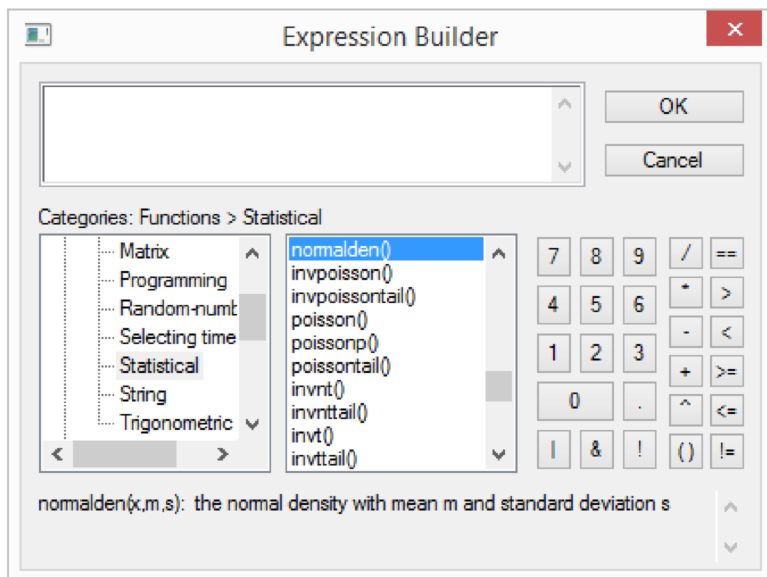
13.8 Using the Expression Builder

The Expression Builder in Stata provides a convenient way to create expressions using any of the methods described above. To access the Expression Builder, click on the **Create** button in a dialog box of any command that allows an *exp*.

Within the Expression Builder, you can interactively browse and then select almost anything you would want to add to an expression: mathematical constants, variables, system limits, local and global macros, dataset and variable notes, and more. This is especially useful for accessing estimation results and system values when you may not immediately know the name.



You may also find the Expression Builder helpful if you want to use a function because a description of each function, as well as the order of the arguments for each function, is provided at the bottom of the dialog box when it is selected.



Watch a video example of using the Expression Builder.

13.9 Indicator values for levels of factor variables

Stata's factor-variable features let us access virtual indicator variables for categorical variables and their interactions; see **[U] 11.4.3 Factor variables** and **[U] 25 Working with categorical data and factor variables**. We can use those virtual indicator variables in expressions just as though the virtual variables existed in our data. If you have not read about factor-variable varlists in **[U] 11.4.3 Factor variables**, do so now.

If `group` is a categorical variable taking on the value 1, 2, or 3, consider the expression

```
. generate group1 = 1.group
```

We have taken the virtual indicator variable that is 1 when `group` $= 1$ and 0 when `group` $\neq 1$ and made it into a real variable—`group1`. That is strictly true only if `group` is never missing. If group can be missing, we need to add that `1.group` is missing when `group` is missing.

These virtual variables extend to interactions. If we also have a variable, `sex`, that is 0 for males and 1 for females, then

```
. generate sex0grp2 = 0.sex#2.group
```

creates the variable `sex0grp2`, which is `1` when `sex` $= 0$ and `group` $= 2$, `.` (missing) when `sex` or `group` is missing, and `0` otherwise.

Virtual indicator variables can be used in any expression, including `if` expressions.

❑ Technical note

We have been using the shorthand notation for virtual indicators that drops the `i` prefix. We have written `2.group` rather than `i2.group`. There are three cases where we cannot drop the `i` prefix—when our variable name is `e`, `d`, or `x`. These three letters can be used to construct numbers such as `1e-3`, which can also be typed `1.e-3`. If we have a variable named `e`, are we to interpret `1.e-3` as the number 0.001 or as the virtual indicator variable `1.e` with the number 3 subtracted? Because of longstanding precedent, it is interpreted as the number 0.001. If we want `1.e` interpreted as a virtual indicator, we must include the `i` prefix—`i1.e`.

❑

13.10 Time-series operators

Time-series operators allow you to refer to the lag of `gnp` by typing `L.gnp`, the second lag by typing `L2.gnp`, etc. There are also operators for lead (`F`), difference `D`, and seasonal difference `S`.

Time-series operators can be used with varlists and with expressions. See **[U] 11.4.4 Time-series varlists** if you have not read it already. This section has to do with using time-series operators in expressions such as with `generate`. You do not have to create new variables; you can use the time-series operated variables directly.

13.10.1 Generating lags, leads, and differences

In a time-series context, referring to `L2.gnp` is better than referring to `gnp[_n-2]` because there might be missing observations. Pretend that observation 4 contains data for $t = 25$ and observation 5 data for $t = 27$. `L2.gnp` will still produce correct answers; `L2.gnp` for observation 5 will be the value from observation 4 because the time-series operators look at t to find the relevant observation. The more mechanical `gnp[_n-2]` just goes 2 observations back, which, here, would not produce the desired result.

This same idea holds for differences. In our example, `D.gnp` will produce a missing value in observation 5 ($t = 27$) because there is no data recorded for $t = 26$, and therefore there is no first difference for $t = 27$.

Time-series operators can be used with varlists or with expressions, so you can type

```
. regress val L.gnp r
```

or

```
. generate gnplagged = L.gnp
. regress val gnplagged
```

Before you can type either one, however, you must use the `tsset` command to tell Stata the identity of the time variable; see **[TS] tsset**. Once you have `tsset` the data, anyplace you see an *exp* in a syntax diagram, you may type time series–operated variables, so you can type

```
. summarize r if F.gnp < gnp
```

or

```
. generate grew = 1 if gnp > L.gnp & L.gnp < .
. replace grew = 0 if grew >= . & L.gnp < .
```

or

```
. generate grew = (gnp > L.gnp) if L.gnp < .
```

13.10.2 Time-series operators and factor variables

As with varlists, factor variables may be combined with the L. (lag) and F. (lead) time-series operators in expressions. We can generate a variable containing the lag of the level 2 indicator of group (group = 2) by typing

```
. generate lag2group = 2L.group
```

The operators can be combined anywhere expressions are allowed. We can select observations for which the lag of the second level of group is 1 by typing if i2L.group.

They can be combined in interactions. We can generate the lag of the interaction of sex = 1 with group = 3 by typing

```
. generate lag1sexX3grp = 1L.sex#2L.group
```

See **[U] 11.4.3 Factor variables** and **[U] 11.4.4 Time-series varlists** for more on factor variables and time-series operators.

13.10.3 Operators within groups

Stata also understands panel or cross-sectional time-series data. For instance, if you type

```
. tsset country time
```

you are declaring that you have time-series data. The time variable is time, and you have time-series data for separate countries.

Once you have tsset both cross-sectional and time identifiers, you proceed just as you would if you had a simple time series.

```
. generate grew = (gnp > L.gnp) if L.gnp < .
```

would produce correct results. The L. operator will not confuse the observation at the end of one panel with the beginning of the next.

13.10.4 Video example

Time series, part 3: Time-series operators

13.11 Label values

If you have not read **[U] 12.6 Dataset, variable, and value labels**, please do so. You may use labels in an expression in place of the numeric values with which they are associated. To use a label in this way, type the label in double quotes followed by a colon and the name of the value label.

▷ Example 11

If the value label yesno associates the label yes with 1 and no with 0, then "yes":yesno (said aloud as the value of yes under yesno) is evaluated as 1. If the double-quoted label is not defined in the indicated value label, or if the value label itself is not found, a missing value is returned. Thus the expression "maybe":yesno is evaluated as *missing*.

```
. use http://www.stata-press.com/data/r14/gxmpl9, clear
. list

     +--------------------+
     |      name   answer |
     |--------------------|
  1. |   Mikulin       no |
  2. |    Gaines       no |
  3. |     Hilbe      yes |
  4. |    DeLeon       no |
  5. |      Cain       no |
     |--------------------|
  6. |      Wann      yes |
  7. | Schroeder       no |
  8. |       Cox       no |
  9. |    Bishop       no |
 10. |    Hardin      yes |
     |--------------------|
 11. | Lancaster      yes |
 12. |     Poole       no |
     +--------------------+

. list if answer=="yes":yesno

     +--------------------+
     |      name   answer |
     |--------------------|
  3. |     Hilbe      yes |
  6. |      Wann      yes |
 10. |    Hardin      yes |
 11. | Lancaster      yes |
     +--------------------+
```

In the above example, the variable `answer` is not a string variable; it is a numeric variable that has the associated value label `yesno`. Because `yesno` associates `yes` with 1 and `no` with 0, we could have typed `list if answer==1` instead of what we did type. We could not have typed `list if answer=="yes"` because `answer` is not a string variable. If we had, we would have received the error message "type mismatch".

◁

13.12 Precision and problems therein

Examine the following short Stata session:

```
. drop _all
. input x y
                x          y
  1. 1 1.1
  2. 2 1.2
  3. 3 1.3
  4. end
. count if x==1
    1
. count if y==1.1
    0
```

```
. list

     +---------+
     | x     y |
     |---------|
  1. | 1   1.1 |
  2. | 2   1.2 |
  3. | 3   1.3 |
     +---------+
```

We created a dataset containing two variables, x and y. The first observation has x equal to 1 and y equal to 1.1. When we asked Stata to count the number of times that the variable x took on the value 1, we were told that it occurred once. Yet when we asked Stata to count the number of times y took on the value 1.1, we were told zero—meaning that it never occurred. What has gone wrong? When we list the data, we see that the first observation has y equal to 1.1.

Despite appearances, Stata has not made a mistake. Stata stores numbers internally in binary form, and the number 1.1 has no exact binary representation—that is, there is no finite string of binary digits that is equal to 1.1.

❑ Technical note

The number 1.1 in binary form is 1.0001100110011 . . . , where the period represents the binary point. The problem binary computers have with storing numbers like 1/10 is much like the problem we base-10 users have in precisely writing 1/11, which is 0.0909090909 . . . .

For detailed information about precision on binary computers and how Stata stores binary floating-point numbers, see Gould (2011a).

❑

The number that appears as 1.1 in the listing above is actually 1.1000000238419, which is off by roughly 2 parts in 10^8. Unless we tell Stata otherwise, it stores all numbers as floats, which are also known as *single-precision* or *4-byte reals*. On the other hand, Stata performs all internal calculations in double, which is also known as *double-precision* or *8-byte reals*. This is what leads to the difficulty.

In the above example, we compared the number 1.1, stored as a float, with the number 1.1 stored as a double. The double-precision representation of 1.1 is more accurate than the single-precision representation, but it is also different. Those two numbers are not equal.

There are several ways around this problem. The problem with 1.1 apparently not equaling 1.1 would never arise if the storage precision and the precision of the internal calculations were the same. Thus you could store all your data as doubles. This takes more computer memory, however, and it is unlikely that your data are really that accurate and the extra digits would meaningfully affect any calculated result, even if the data were that accurate.

❑ Technical note

This is unlikely to affect any calculated result because Stata performs all internal calculations in double precision. This is all rather ironic, because the problem would also not arise if we had designed Stata to use single precision for its internal calculations. Stata would be less accurate, but the problem would have been completely disguised from the user, making this entry unnecessary.

❑

Another solution is to use the float() function. float(*x*) rounds *x* to its float representation. If we had typed count if y==float(1.1) in the above example, we would have been informed that there is one such value.

13.13 References

Cox, N. J. 2006. Stata tip 33: Sweet sixteen: Hexadecimal formats and precision problems. *Stata Journal* 6: 282–283.

——. 2011a. Speaking Stata: Compared with . . . . *Stata Journal* 11: 305–314.

——. 2011b. Stata tip 96: Cube roots. *Stata Journal* 11: 149–154.

Crow, K. 2012. Building complicated expressions the easy way. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2012/02/07/building-complicated-expressions-the-easy-way/.

Gould, W. W. 2006. Mata Matters: Precision. *Stata Journal* 6: 550–560.

——. 2011a. How to read the %21x format, part 2. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/02/10/how-to-read-the-percent-21x-format-part-2/.

——. 2011b. Precision (yet again), Part I. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/06/17/precision-yet-again-part-i/.

——. 2011c. Precision (yet again), Part II. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/06/23/precision-yet-again-part-ii/.

——. 2012. The penultimate guide to precision. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2012/04/02/the-penultimate-guide-to-precision/.

Linhart, J. M. 2008. Mata Matters: Overflow, underflow and the IEEE floating-point format. *Stata Journal* 8: 255–268.

Mitchell, M. N. 2010. *Data Management Using Stata: A Practical Handbook*. College Station, TX: Stata Press.

Weiss, M. 2009. Stata tip 80: Constructing a group variable with specified group sizes. *Stata Journal* 9: 640–642.