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.

Continuing with the multinomial outcome case: the outcome variable must be numeric. **The syntax**

**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 #*

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 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*.

The number 1.1 in binary form is 1.000110011001 , 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 108. **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 doubles, which are 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.

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*.

An ado-file defines a Stata command, but not all Stata commands are defined by ado-files. When you type summarize to obtain summary statistics, you are using a command built into Stata.  
When you type ci to obtain confidence intervals, you are running an ado-file. The results of using  
a built-in command or an ado-file are indistinguishable.  
An ado-file is a text file that contains a Stata program. When **you type a command that Stata does**  
**not know**, *it looks in certain places for an ado-file of that name*. If **Stata finds it**, *Stata loads and*  
*executes it*, so it appears to you as if the ado-command is just another command built into Stata.  
We just told you that Stata’s ci command is implemented as an ado-file. That means that,  
somewhere, there is a file named ci.ado.  
Ado-files usually come with help files. When **you type help ci (or select Help > Stata command...,**  
**and type ci)**, *Stata looks for ci.sthlp*, just as *it looks for ci.ado* when **you use the ci command**.  
A help file is also a text file that tells Stata’s help system what to display