

**SOCIAL SCIENCE COMPUTING COOPERATIVE  
(HTTPS://WWW.SSCC.WISC.EDU/)**

Search

**ACCOUNTS**  
(HTTPS://WWW.SSCC.WISC.EDU/  
/ACCOUNTS/)**GET HELP**  
(HTTPS://WWW.SSCC.WISC.EDU/  
/GET-HELP/)**STATISTICAL CONSULTING**  
(HTTPS://WWW.SSCC.WISC.EDU  
/STATISTICS/STATISTICAL-  
CONSULTING/)**KNOWLEDGE BASE**  
(HTTPS://KB.WISC.EDU  
/SSCC/)**TRAINING**  
(HTTPS://WWW.SSCC.WISC.EDU  
/STATISTICS/TRAINING/)**WINSTAT**  
(HTTPS://KB.WISC.EDU  
/SSCC/USING-  
WINSTAT)

## Stata Programming Tools

This article will introduce you to many Stata programming tools that are not needed by everyone but are very useful in certain circumstances. The intended audience is Stata veterans who are already familiar with and comfortable using Stata syntax and fundamental programming tools like macros, `foreach` and `forvalues`.

If you are new to Stata, our [Stata for Researchers \(sfr-intro.htm\)](https://www.sccc.wisc.edu/sfr-intro.htm) will teach you basic Stata syntax, and [Stata Programming Essentials \(stata\\_prog1.htm\)](https://www.sccc.wisc.edu/stata-prog1.htm) will teach you the fundamental programming tools. Unlike this article, they do not assume any Stata or general programming experience.

Topics discussed in this article include:

- [Compound Double Quotes](#)
- [Capture](#)
- [Variables in Scalar Contexts](#)
- [Advanced Macros](#)
- [Branching If](#)
- [While Loops](#)
- [Programs](#)

If you need to learn about a specific topic, feel free to skip to it. Some of the examples use material covered previously (i.e. a program containing a branching *if*) but the explanations are self-contained.

### Compound Double Quotes

The beginning and ending of a string are normally denoted by double quotes ("string"). This causes problems when the string itself contains double quotes. For example, if you wanted to display the string Hamlet said "To be, or not to be." you could not use the code:

```
display "Hamlet said "To be, or not to be.""
```

The solution is what Stata calls compound double quotes. When Stata sees `"' (left single quote followed by a double quote) it treats what follows as a string until it sees "'` (double quote followed by a right single quote). Thus:

```
display `"'Hamlet said "To be, or not to be."`"'
```

### Capture

On occasion you may expect a command to generate an error under certain circumstances but choose to ignore it. For example, putting `log close` at the beginning of a do file prevents a previously opened log from interfering with your do file, but generates an error if no log is open—even though not having a log open is exactly the situation you want to create.

The `capture` prefix prevents Stata from halting your do file if the ensuing command generates an error. The error is "captured" first. Thus:

```
capture log close
```

will close any open log but not crash a do file if no log file is open.

The `capture` prefix should only be used when you fully understand the error the command sometimes generates and why it does so—and you are very confident that that error can be ignored.

When a command is complete it stores a "return code" in `_rc`. A return code of zero generally means the command executed successfully. You can use `capture` followed by a [branching if](#) based on the return code to have Stata do different things depending on whether an error occurred or not.

### Variables in Scalar Contexts

A Stata variable is a vector: it has many values. If you type `list x` you get a list of all the values of `x`. However, some contexts call for a scalar. For example, the `display` command displays just one thing. If a variable is used in a scalar context, the value of the variable for the first observation is used. Thus if you type:

```
display x
```

you'll get just the first value of `x`, as if you'd typed:

```
display x[1]
```

We suggest not taking advantage of this behavior, because it makes for confusing code. If a command calls for a scalar and you want that scalar to be the first value of `x`, type `x[1]` rather than just `x`.

This behavior can cause real problems if you don't realize a particular context calls for a scalar, as you'll see in the section on [branching if](#).

### Advanced Macros

This section will discuss many tricks you can use to define macros. Type `help local` for even more.

## Storing Lists of Values with `levelsof`

The `levelsof` command lists the valid values of a variable and stores them in a local macro. The syntax is:

```
levelsof variable, local(macro)
```

It is frequently used with an *if* condition to store those values of a variable that appear in a given subsample.

The following example (taken from [Stata Programming Essentials \(stata\\_prog1.htm\)](#)) uses `levelsof` and a `foreach` loop to run a survey-weighted regression separately for each race. This recreates the functionality of `by`: even though `by`: can't be used with `svy`:

```
levelsof race, local(races)
foreach race of local races {
    display _newline(2) "Race=`race'"
    svy, subpop(if race==`race'): reg income age i.education
}
```

Since `levelsof` doesn't include missing values in its list, the above code will not run a regression for observations with a missing value for `race`. This is different from `by`:, which treats observations with a missing value of the *by* variable as just another group to act on.

Suppose you asked individuals to list the social groups they belong to. The groups are encoded as numbers and stored in variables `group_i` where *i* is an integer. Thus if a person said she belongs to group five and group three (in that order) her value of `group_1` would be 5 and her value of `group_2` would be 3. (If she lists fewer groups than the maximum number allowed, she will have missing values for some `group_i` variables.) You then want to create an indicator variable for each group which has a value of 1 if the person said she is a member of that group and zero otherwise:

```
foreach groupvar of varlist group_* {
    levelsof `groupvar', local(groups)
    foreach group of local groups {
        capture gen mem_`group'=0
        replace mem_`group'=1 if `groupvar'==`group'
    }
}
```

This loops over the `group_i` variables, figures out which groups were listed in each one, loops over them, creates an indicator variable for each group, and sets the indicator to 1 if the person listed that group.

Note that you don't need to know ahead of time how many groups a person can list or the numbers of the groups that can appear. The `gen` command has `capture` in front of it because the indicator for a given group may have already been created: if someone listed group 5 as his first group and someone else listed group 5 as his second group, `mem_5` will be created when processing `group_1` and trying to create `mem_5` again when processing `group_2` gives an error—but you know it's an error you can ignore.

## Expanding Variable Lists with `unab`

The `unab` command "unabbreviates" a *varlist* ("expand" was taken) and puts the results in a macro. The syntax is:

```
unab macro: varlist
```

For example, if you had variables `x1`, `x2` and `x3`, then:

```
unab vars: x*
```

would create a local macro called `vars` and place in it `x1 x2 x3`. This can be useful for commands that require a list of variables but cannot use *varlist* syntax, like `reshape` when going from long to wide.

## Lists of Files

You can place a list of files in a macro by sending `local` the output of a `dir` command. The syntax is:

```
local macroname: dir directory files "pattern they must match"
```

Here *macroname* is the name of the macro you want to create, *directory* is the directory where the files are located and *pattern they must match* is something like `"*"` for all files or `"*.dta"` for all Stata data sets. For example, the following would put a list of all Stata data sets in the current working directory in a macro called `datafiles`:

```
local datafiles: dir . files "*.dta"
```

One complication is that the file names are placed in quotes (so it can handle file names with spaces in them). Thus to display them you have to use compound double quotes:

```
di ``"datafiles"''
```

However, this doesn't cause problems if you want to loop over the list of files:

```
foreach file of local datafiles {
    use `file', clear
    // do something with the file
}
```

## Formatting the Contents of a Macro

You can apply a format to a number before storing it in a macro by sending `local` the output of a `display` command that includes a format. For example, the following command stores the R-squared of the most recently run regression, `e(r2)`, in a macro called `r2`, but using the format `%5.4f` so it has four digits rather than sixteen.

```
local r2: display %5.4f e(r2)
```

See [Including Calculated Results In Stata Graphs \(4-25.htm\)](#) for an example that uses this tool.

## Incrementing and Decrementing Macros

Macros frequently need to be increased by one and less frequently decreased by one. You can do so with the `++` and `--` operators. These go inside the macro quotes either before or after the macro name. If they are placed before, the macro is incremented (or decremented) and

then the result placed in the command. If they are placed after, the current value of the macro is placed in the command and then the macro is incremented (or decremented). Try the following:

```
local x 1
display `++x'
display `x--'
display `x'
```

You can use the increment or decrement operators in a `local` command to change a macro without doing anything else, but be sure to put the operator before the macro's name. For example the following does **not** increase `x`:

```
local x `x++'
```

The macro processor replaces the macro ``x++'` with 1. It then increases `x` to 2, but then when Stata proper executes the command it sees `local x 1` and sets `x` back to 1. The following does increase `x`:

```
local x `++x'
```

## Branching If

You're familiar with *if* conditions at the end of commands, meaning "only carry out this command for the observations where this condition is true." This is a subsetting *if*. When *if* starts a command, it is a branching *if* and has a very different meaning: "don't execute the following command or commands at all unless this condition is true." The syntax is for a single command is:

```
if condition command
```

For a block of commands, it's:

```
if condition {
    commands
}
```

An *if* block can be followed by an *else* block, meaning "commands to be executed if the condition is not true." The *else* can precede another *if*, allowing for *else if* chains of any length:

```
if condition1 {
    commands to execute if condition1 is true
}
else if condition2 {
    commands to execute if condition one is false and condition2 is true
}
else {
    commands to execute if both condition1 and condition2 are false
}
```

Consider trying to demean a list of variables, where the list is contained in a macro called `varlist` which was generated elsewhere and could contain string variables:

```
foreach var of local varlist {
    capture confirm numeric variable `var'
    if _rc==0 {
        sum `var', meanonly
        replace `var'=`var'-r(mean)
    }
    else display as error "`var' is not a numeric variable and cannot be demeaned."
}
```

The command `confirm numeric variable `var'` checks that ``var'` is a numeric variable, but when preceded by `capture` it does not crash the program if the variable is a string. Instead, `if _rc==0 {` checks whether the `confirm` command succeeded (implying ``var'` is in fact numeric). If it did, the program proceeds to demean ``var'`. If not, the program displays an error message (`as error` makes the message red) and then proceeds to the next variable.

The condition for a branching *if* is only evaluated once, so a branching *if* is a scalar context and the rule that only the first value of a variable will be used applies. In particular, a branching *if* cannot be used for subsetting. Imagine a data set made up of observations from multiple census years where the data from 1960 is coded in a different way and thus has to be handled differently. What you should **not** write is something like:

```
if year==1960 { //don't do this!
    code for handling observations from 1960
}
else {
    code for handling observations from other years
}
```

Since this is a scalar context, the condition `year==1960` is only evaluated once, and the only value of `year` Stata will look at is that of the first observation. Thus if the first observation happens to be from 1960, the entire data set will be coded as if it came from 1960. If not, the entire data set will be coded as if it came from other years. This is not a job for branching *if*, it is a job for a standard subsetting *if* at the end of each command.

The above code might make sense if it were embedded in a loop that processed multiple data sets, where each data set came from a single year. Then `year` would be the same for all the observations in a given data set, and the first observation could stand in for all of them. But in that case we'd suggest writing something like:

```
if year[1]==1960 { // if year for the first observation is 1960 this is a 1960 data set
```

Conditions for branching *if* frequently involve macros. If the macros contain text rather than numbers, the values on both sides of the equals sign need to be placed in quotes:

```
foreach school in West East Memorial {
    if "`school'"=="West" {
        commands that should only be carried out for West High School
    }
}
```

```
    commands that should be carried out for all schools
}
```

## While Loops

`foreach` and `forvalues` loops repeat a block of commands a set number of times, but `while` loops repeat them until a given condition is no longer true. For example:

```
local i 1
while `i'<=5 {
    display `i++'
}
```

is equivalent to:

```
forval i=1/5 {
    display `i'
}
```

Note that `i` is increased by 1 each time through the `while` loop--if you left out that step the loop would never end.

The following code is a more typical use of `while`:

```
local xnew 1
local xold 0
local iteration 1
while abs(`xnew'-`xold')>.001 & `iteration'<100 {
    local xold `xnew'
    local xnew=`xold'-(3-`xold'^3)/(-3*`xold'^2)
    display "Iteration: `iteration++', x: `xnew'"
}
```

The above uses the Newton-Raphson method to solve the equation  $3-x^3=0$ . The algorithm proceeds until the result from the current iteration differs from that of the last iteration by less than .001, or it completes 100 iterations. The second condition acts as a failsafe in case the algorithm does not converge. Note that the initial value of `xold` is not used, but if it were the same as the initial value of `xnew` then the `while` condition would be false immediately and the loop would never be executed.

## Programs

A Stata program is a block of code which can be executed at any time by invoking the program's name. They are useful when you need to perform a task repeatedly, but not all at once. To begin defining a program, type:

```
program define name
```

where *name* is replaced by the name of the program to be defined. Subsequent commands are considered part of the program, until you type

```
end
```

Thus a basic "Hello World" program is:

```
program define hello
    display "Hello World"
end
```

To run this program, type `hello`.

A program cannot be modified after it is defined; to change it you must first drop the existing version with `program drop` and then define it again. Since a do file run in an interactive session can't be sure what's been defined previously, it's best to capture `program drop` a program before you define it:

```
capture program drop hello
program define hello
    display "Hello World Again"
end
```

## Arguments

Programs can be controlled by passing in *arguments*. An argument can be anything you can put in a macro: numbers, text, names of variables, etc. You pass arguments into a program by typing them after it's program name. Thus:

```
hello Russell Dimond
```

runs the `hello` program with two arguments: `Russell` and `Dimond`. But arguments only matter if the program does something with them--the current version of `hello` will completely ignore them.

Programs that use arguments should first use the `args` command to assign them to local macros. The command:

```
args fname lname
```

puts the first argument the program received in the macro `fname` and the second in the macro `lname`. You can then use those macros in subsequent commands:

```
capture program drop hello
program define hello
    args fname lname
    display "Hello `fname' `lname'"
end
```

If you then type:

```
hello Russell Dimond
```

the output will be:

```
Hello Russell Dimond
```

Of course if you type:

```
hello Dimond Russell
```

the output will be:

```
Hello Dimond Russell
```

It's up to you to make sure the arguments you pass in match what the program is expecting.

The macro `_rc` is always defined, and contains a list of all the arguments that were passed into the program. This is useful for handling lists of unknown length. For example, you could take the code you wrote earlier for demeaning lists of variables and turn it into a program:

```
program define demean
    foreach var of local 0 {
        capture confirm numeric variable `var'
        if _rc==0 {
            sum `var', meanonly
            replace `var'=`var'-(mean)
        }
        else display as error "`var' is not a numeric variable and cannot be demeaned."
    }
end
```

To run this program, you'd type `demean` and then a list of variables to be demeaned:

```
demean x y z
```

You might also want to look at the `syntax` command, which makes it relatively easy to write a program that understands standard Stata syntax, but `syntax` is beyond the scope of this article.

## Returning Values

Your program can return values in the `r()` or `e()` vectors, just like official Stata commands. This is critical if your program is intended for use with `bootstrap` or `simulate`. To do so, first declare in your program `define` command that the program is either `rclass` (puts results in the `r()` vector) or `eclass` (puts results in the `e()` vector):

```
program define myprogram, rclass
```

When you have a result to return, use the `return` command. The general syntax is:

```
return type name=value
```

where `type` can be scalar, local or matrix, `value` is what you want to return, and `name` is what you want it to be called. As a trivial example:

```
return scalar x=3
```

When the program is complete, you can refer to the result as `r(name)` or `e(name)`. Thus if you've just run a program containing the above `return` command, typing:

```
gen var=r(x)
```

creates a variable `var` with the value 3.

For a more realistic example, see the last section of [Bootstrapping in Stata \(4-27.htm#BootstrappingResultsYouveCalculated\)](#).


---

Last Revised: 12/14/2010



### Contact Us

1180 Observatory Dr. Rm 4226  
Madison, WI 53706

Map  (<https://www.google.com/maps/place/University+of+Wisconsin+-+William+H+Sewell+Social+Sciences+Building/@43.0754567,-89.4098389,17z/data=!4m2!1m6!3m5!1s0x8807ac950a7f481d:0xc9e065e091733a642sUniversity+of+Wisconsin-Madison!8m2!3d43.076592!4d-89.4124875!3m4!1s0x8807ac950a7f481d:0xf6ca8f24bb55e09c!8m2!3d43.0764116!4d-89.4052577>)

Email: [helpdesk@ssc.wisc.edu](mailto:helpdesk@ssc.wisc.edu) (mailto:[helpdesk@ssc.wisc.edu](mailto:helpdesk@ssc.wisc.edu))

Phone: 608-262-9917 (tel:608-262-9917)

Feedback, questions or accessibility issues: [helpdesk@ssc.wisc.edu](mailto:helpdesk@ssc.wisc.edu) (mailto:[helpdesk@ssc.wisc.edu](mailto:helpdesk@ssc.wisc.edu)).

This site was built using the [UW Theme \(https://uwtheme.wordpress.wisc.edu/\)](https://uwtheme.wordpress.wisc.edu/). © 2021 Board of Regents of the [University of Wisconsin System](http://www.wisconsin.edu). (<http://www.wisconsin.edu>)