Title

stata.com

macro — Macro definition and manipulation

Description Syntax Remarks and examples References Also see

Description

`global` assigns strings to specified global macro names (*mnames*). `local` assigns strings to local macro names (*lclnames*). Both double quotes (`"` and `"`) and compound double quotes (`‘"` and `"’`) are allowed; see [U] **18.3.5 Double quotes**. If the *string* has embedded quotes, compound double quotes are needed.

`tempvar` assigns names to the specified local macro names that may be used as temporary variable names in a dataset. When the program or do-file concludes, any variables with these assigned names are dropped.

`tempname` assigns names to the specified local macro names that may be used as temporary local macro, scalar, matrix, or frame names. When the program or do-file concludes, any local macros, scalars, matrices, or frames with these assigned names are dropped.

`tempfile` assigns names to the specified local macro names that may be used as names for temporary files. When the program or do-file concludes, any datasets created with these assigned names are erased.

`macro` manipulates global and local macros.

See [U] **18.3 Macros** for information on macro substitution.

Syntax

`global` *mname* [`=` *exp* | `:`*macro_fcn* | `"`[*string*]`"` | `‘"`[*string*]`"’`]

`local` *lclname* [`=` *exp* | `:`*macro_fcn* | `"`[*string*]`"` | `‘"`[*string*]`"’`]

`tempvar` *lclname* [*lclname* [...]]

`tempname` *lclname* [*lclname* [...]]

`tempfile` *lclname* [*lclname* [...]]

`local` { `++`*lclname* | `--`*lclname* }

`macro dir`

`macro drop` { *mname* [*mname* [...]] | *mname*`*` | `_all` }

`macro list` [*mname* [*mname* [...]] | `_all`]

`macro shift` [*#*]

[...]`‘`*expansion_optr*`’`[...]

where *expansion_optr* is

lclname | ++*lclname* | *lclname*++ | --*lclname* | *lclname*-- | =*exp* |
:*macro_fcn* | .*class_directive* | macval(*lclname*)

and where *macro_fcn* is any of the following:

Macro function for extracting program properties

properties *command*

Macro function for extracting program results class

results *command*

Macro functions for extracting data attributes

{ type | format | value label | variable label } *varname*

data label

sortedby

label { *valuelabelname* | (*varname*) } { maxlength | # [#$_2$] } [, strict]

constraint { # | dir }

char { *varname*[] | *varname*[*charname*] } or char { _dta[] | _dta[*charname*] }

Macro functions for extracting attributes of alias variables

isalias *varname*

type *varname*

aliasframe *varname*

aliaslinkname *varname*

aliasvarname *varname*

Macro function for naming variables

permname *suggested_name* [, length(#)]

Macro functions for filenames and file paths

adosubdir ["]*filename*["]

dir ["]*dirname*["] { files | dirs | other }["]*pattern*["] [, nofail respectcase]

sysdir [STATA | BASE | SITE | PLUS | PERSONAL | *dirname*]

Macro function for accessing operating-system parameters

environment *name*

Macro functions for names of stored results

`e(scalars|macros|matrices|functions)`

`r(scalars|macros|matrices|functions)`

`s(macros)`

`all {globals|scalars|matrices} ["`*pattern*`"]`

`all {numeric|string} scalars ["`*pattern*`"]`

Macro function for formatting results

`display` *display_directive*

Macro function for manipulating lists

`list` *macrolist_directive*

Macro functions related to matrices

`{rownames|colnames|rowfullnames|colfullnames}` *matname* `[, quoted]`

`{roweq|coleq}` *matname* `[, quoted]`

`{rownumb|colnumb|roweqnumb|coleqnumb}` *matname string*

`{rownfreeparms|colnfreeparms|rownlfs|colnlfs}` *matname*

`{rowsof|colsof|rowvarlist|colvarlist}` *matname*

`{rowlfnames|collfnames}` *matname* `[, quoted]`

Macro function related to time-series operators

`tsnorm` *string* `[, varname]`

Macro function for copying a macro

`copy {local|global}` *mname*

Macro functions for parsing

`word {count|# of}` *string*

`piece` *#piece_number #length_of_pieces* `of [‘]"`*string*`"[’] [, nobreak]`

`strlen{local|global}` *mname*

`ustrlen{local|global}` *mname*

`udstrlen{local|global}` *mname*

`subinstr {global` *mname2*`|local` *lclname2*`}`
`{"`*from*`"|‘"`*from*`"’} {"`*to*`"|‘"`*to*`"’}`
`[, all count(global` *mname3*`| local` *lclname3*`) word]`

Remarks and examples

stata.com

Remarks are presented under the following headings:

Macros are a tool used in programming Stata, and this entry assumes that you have read **[U] 18 Programming Stata** and especially **[U] 18.3 Macros**. This entry concerns advanced issues not previously covered.

Formal definition of a macro

A *macro* has a *macro name* and *macro contents*. Everywhere a punctuated macro name appears in a command—punctuation is defined below—the macro contents are substituted for the macro name.

Macros come in two types, global and local. Macro names are up to 32 characters long for global macros and up to 31 characters long for local macros. The contents of global macros are defined with the `global` command and those of local macros with the `local` command. Global macros, once defined, are available anywhere in Stata. Local macros exist solely within the program or do-file in which they are defined. If that program or do-file calls another program or do-file, the local macros previously defined temporarily cease to exist, and their existence is reestablished when the calling program regains control. When a program or do-file ends, its local macros are permanently deleted.

To substitute the macro contents of a global macro name, the macro name is typed (punctuated) with a dollar sign (`$`) in front. To substitute the macro contents of a local macro name, the macro name is typed (punctuated) with surrounding left and right single quotes (`‘’`). In either case, braces (`{}`) can be used to clarify meaning and to form nested constructions. When the contents of an undefined macro are substituted, the macro name and punctuation are removed, and nothing is substituted in its place.

For example,

The input ...	is equivalent to ...
`global a "myvar"` `generate $a = oldvar`	`generate myvar = oldvar`
`generate a = oldvar`	`generate a = oldvar`
`local a "myvar"` `generate ‘a’ = oldvar`	`generate myvar = oldvar`
`generate a = oldvar`	`generate a = oldvar`

`global a "newvar"`
`global i = 2`
`generate $a$i = oldvar`
	`generate newvar2 = oldvar`

`local a "newvar"`
`local i = 2`
`generate ‘a’‘i’ = oldvar`
	`generate newvar2 = oldvar`

`global b1 "newvar"`
`global i=1`
`generate ${b$i} = oldvar`
	`generate newvar = oldvar`

`local b1 "newvar"`
`local i=1`
`generate ‘b‘i’’ = oldvar`
	`generate newvar = oldvar`

`global b1 "newvar"`
`global a "b"`
`global i = 1`
`generate ${$a$i} = oldvar`
	`generate newvar = oldvar`

`local b1 "newvar"`
`local a "b"`
`local i = 1`
`generate ‘‘a’‘i’’ = oldvar`
	`generate newvar = oldvar`

Global and local macro names

What we say next is an exceedingly fine point: global macro names that begin with an underscore are really local macros; this is why local macro names can have only 31 characters. The `local` command is formally defined as equivalent to `global _`. Thus the following are equivalent:

```
local x                      global _x
local i=1                    global _i=1
local name "Bill"            global _name "Bill"
local fmt : format myvar     global _fmt : format myvar
local 3 ‘2’                  global _3 $_2
```

`tempvar` is formally defined as equivalent to `local` *name* `: tempvar` for each name specified after `tempvar`. Thus

```
tempvar a b c
```

is equivalent to

```
local a : tempvar
local b : tempvar
local c : tempvar
```

which in turn is equivalent to

```
global _a : tempvar
global _b : tempvar
global _c : tempvar
```

`tempfile` is defined similarly.

Macro assignment

When you type

```
. local name "something"
```

or

```
. local name ‘"something"’
```

something becomes the contents of the macro. The compound double quotes (‘" and "’) are needed when *something* itself contains quotation marks. In fact, if the string is anything more complex than a single word, it is safest to enclose the string in compound quotes (‘" "’). The outermost compound quotes will be stripped, and all that remains will be assigned to `name`. Note that any embedded macro references in *something* are expanded before assignment to `name` whether or not compound quotes are used.

When you type

```
. local name = something
```

something is evaluated as an expression, and the result becomes the contents of the macro. Note the presence and lack of the equal sign. That is, if you type

```
. local problem "2+2"
. local result = 2+2
```

then `problem` contains `2+2`, whereas `result` contains `4`.

Finally, when you type

```
. local name : something
```

something is interpreted as a macro function. (Note the colon rather than nothing or the equal sign.) Of course, all of this applies to `global` as well as to `local`.

`local` `++`*lclname*, or `local` `--`*lclname*, is used to increment, or decrement, *lclname*. For instance, typing

```
. local ++x
```

is equivalent to typing

```
. local x = ‘x’ + 1
```

Macro functions

Macro functions are of the form

```
. local mname : ...
```

For instance,

```
. local x : type mpg
. local y : sortedby
. local z : display %9.4f sqrt(2)
```

We document the macro functions below. Macro functions are typically used in programs, but you can experiment with them interactively. For instance, if you are unsure what ‘`local x : type mpg`’ does, you could type

```
. local x : type mpg
. display "‘x’"
int
```

Macro function for extracting program properties

`properties` *command*
returns the properties declared for *command*; see [P] **program properties**.

Macro function for extracting program results class

`results` *command*
returns the results class—`nclass`, `eclass`, `rclass`, or `sclass`—declared for *command*; see [P] **program**.

Macro functions for extracting data attributes

`type` *varname*
returns the storage type of *varname*, which might be `int`, `long`, `float`, `double`, `str1`, `str2`, etc. If *varname* is an alias variable, `type` returns the storage type of the linked variable or `unknown` if the linked variable cannot be found.

`format` *varname*
returns the display format associated with *varname*, for instance, `%9.0g` or `%12s`.

`value label` *varname*
returns the name of the value label associated with *varname*, which might be "" (meaning no label), or, for example, `make`, meaning that the value label's name is `make`.

`variable label` *varname*
returns the variable label associated with *varname*, which might be "" (meaning no label), or, for example, `Repair Record 1978`.

`data label`
returns the dataset label associated with the dataset currently in memory, which might be "" (meaning no label), or, for example, `1978 automobile data`. See [D] **label**.

`sortedby`
returns the names of the variables by which the data in memory are currently sorted, which might be "" (meaning not sorted), or, for example, `foreign mpg`, meaning that the data are in the order of the variable `foreign`, and, within that, in the order of `mpg` (the order that would be obtained from the Stata command `sort foreign mpg`). See [D] **sort**.

`label` *valuelabelname* { `maxlength` | *#* [*#*$_2$] } [`, strict`]
returns the label value of *#* in *valuelabelname*. For instance, `label forlab 1` might return `Foreign cars` if `forlab` were the name of a value label and 1 mapped to "Foreign cars". If 1 did not correspond to any mapping within the value label, or if the value label `forlab` were not defined, `1` (the *#* itself) would be returned.

#$_2$ optionally specifies the maximum length of the label to be returned. If `label forlab 1` would return `Foreign cars`, then `label forlab 1 6` would return `Foreig`.

`maxlength` specifies that, rather than looking up a number in a value label, `label` return the maximum length of the labelings. For instance, if value label `yesno` mapped 0 to `no` and 1 to `yes`, then its `maxlength` would be 3 because `yes` is the longest label and it has three characters.

`strict` specifies that nothing is to be returned if there is no value label for *#*.

`label (`*varname*`) {maxlength | # [#2]} [, strict]`
works exactly as the above, except that rather than specifying the *valuelabelname* directly, you indirectly specify it. The value label name associated with *varname* is used, if there is one. If not, it is treated just as if *valuelabelname* were undefined, and the number itself is returned.

`constraint {#|dir}`
gives information on constraints.

`constraint` # puts constraint # in *mname* or returns "" if constraint # is not defined. `constraint` # for # < 0 is an error.

`constraint dir` returns an unsorted numerical list of those constraints that are currently defined. For example,

```
. constraint 1 price = weight
. constraint 2 mpg > 20
. local myname : constraint 2
. macro list _myname
_myname:        mpg > 20
. local aname : constraint dir
. macro list _aname
_aname:         2 1
```

`char {`*varname*`[] | `*varname*`[`*charname*`]}` or `char {_dta[] | _dta[`*charname*`]}`
returns information on the characteristics of a dataset; see [P] **char**. For instance,

```
. sysuse auto
(1978 automobile data)
. char mpg[one] "this"
. char mpg[two] "that"
. local x : char mpg[one]
. di "`x'"
this
. local x : char mpg[nosuch]
. di "`x'"
. local x : char mpg[]
. di "`x'"
two one
```

Macro functions for extracting attributes of alias variables

In the following setup, we link the `default` frame to a `target` frame and create alias variables using this linkage. Specifically, we create a new frame named `target`, then populate it with the auto data and a new variable `id` that uniquely identifies the observations. In the current frame named `default`, we set the observations, create a variable named `id` that identifies observations in the `target` frame, then use `frlink` and `id` to create a link to frame `target`, naming the linking variable `link`. Finally, we use `fralias add` and the linking variable `link` to create alias variables that are linked to frame `target`. With option `prefix(l_)`, the names of the new alias variables are `l_make` and `l_headroom`.

```
. frame create target
. frame target {
.         quietly sysuse auto
.         generate id = _n
. }
```

```
. set obs 74
Number of observations (_N) was 0, now 74.
. generate id = _n
. frlink 1:1 id, frame(target) generate(link)
(all observations in frame default matched)
. fralias add make headroom, from(link) prefix(l_)
(2 variables aliased from linked frame)
```

`isalias` *varname*

returns `1` when *varname* is an alias variable, `0` otherwise.

```
. local x : isalias l_make
. display "`x'"
1
. local x : isalias l_headroom
. display "`x'"
1
. local x : isalias link
. display "`x'"
0
```

`type` *varname*

returns the storage type of the variable that *varname* is linked to, when *varname* is an alias variable. If the linked variable cannot be found, then `type` returns `unknown`.

```
. local x : type l_make
. display "`x'"
str18
. local x : type l_headroom
. display "`x'"
float
. * break the link
. rename link junk
. local x : type l_make
. display "`x'"
unknown
. * restore the link
. rename junk link
```

`aliasframe` *varname*

returns the name of the frame that *varname* is linked to. If *varname* is not an alias variable, or the linking variable cannot be found, then `aliasframe` returns an empty string.

```
. local x : aliasframe l_make
. display "`x'"
target
. * break the link
. rename link junk
. local x : aliasframe l_make
. display "`x'"
. * restore the link
. rename junk link
```

`aliaslinkname` *varname*

returns the name of the linking variable that was used to create *varname*. If *varname* is not an alias variable, then `aliaslinkname` returns an empty string.

```
. local x : aliaslinkname l_make
. display "`x'"
link
```

`aliasvarname` *varname*
returns the name of the variable that *varname* is linked to. If *varname* is not an alias variable, then `aliasvarname` returns an empty string.

```
. local x : aliasvarname l_make
. display "`x'"
make
. local x : aliasvarname l_headroom
. display "`x'"
headroom
```

Macro function for naming variables

`permname` *suggested_name* [`, length(#)`]
returns a valid new variable name based on *suggested_name* in *mname*, where *suggested_name* must follow naming conventions but may be too long or correspond to an already existing variable.

`length(#)` specifies the maximum length of the returned variable name, which must be between 8 and 32. `length(32)` is the default. For instance,

```
. local myname : permname foreign
. macro list _myname
_myname:        foreign1
.local aname : permname displacement, length(8)
. macro list _aname
_aname:         displace
```

Macro functions for filenames and file paths

`adosubdir` [`"`] *filename* [`"`]
puts in *mname* the subdirectory in which Stata would search for this file along the ado-path. Typically, the directory name would be the first letter of *filename*. However, certain files may result in a different name depending on their extension.

`dir` [`"`] *dirname* [`"`] { `files` | `dirs` | `other` } [`"`] *pattern* [`"`] [`, nofail respectcase`]
puts in *mname* the specified files, directories, or entries that are neither files nor directories, from directory *dirname* and matching pattern *pattern*, where the pattern matching is defined by Stata's `strmatch(`s_1`,`s_2`)` function; see [FN] **String functions**. The quotes in the command are optional but recommended, and they are nearly always required surrounding *pattern*. The returned string will contain each of the names, separated one from the other by spaces and each enclosed in double quotes. If *mname* is subsequently used in a quoted context, it must be enclosed in compound double quotes: `` `"`mname'"' ``.

The `nofail` option specifies that if the directory contains too many filenames to fit into a macro, rather than issuing an error message, the filenames that fit into *mname* should be returned. `nofail` should rarely, if ever, be specified.

In Windows only, the `respectcase` option specifies that `dir` respect the case of filenames when performing matches. Unlike other operating systems, Windows has, by default, case-insensitive filenames. `respectcase` is ignored in operating systems other than Windows.

For example,

`local list : dir . files "*"` makes a list of all regular files in the current directory. In `list` might be returned `"subjects.dta" "step1.do" "step2.do" "reest.ado"`.

`local list : dir . files "s*", respectcase` in Windows makes a list of all regular files in the current directory that begin with a lowercase "s". The case of characters in the filenames is preserved. In Windows, without the `respectcase` option, all filenames would be converted to lowercase before being compared with *pattern* and possibly returned.

`local list : dir . dirs "*"` makes a list of all subdirectories of the current directory. In `list` might be returned `"notes" "subpanel"`.

`local list : dir . other "*"` makes a list of all things that are neither regular files nor directories. These files rarely occur and might be, for instance, Unix device drivers.

`local list : dir "\mydir\data" files "*"` makes a list of all regular files that are to be found in `\mydir\data`. Returned might be `"example.dta" "make.do" "analyze.do"`.

It is the names of the files that are returned, not their full path names.

`local list : dir "subdir" files "*"` makes a list of all regular files that are to be found in `subdir` of the current directory.

`sysdir` [`STATA` | `BASE` | `SITE` | `PLUS` | `PERSONAL`]
returns the various Stata system directory paths; see [P] **sysdir**. The path is returned with a trailing separator; for example, `sysdir STATA` might return `D:\PROGRAMS\STATA\`.

`sysdir` *dirname*
returns *dirname*. This function is used to code `local x : sysdir 'dir'`, where `'dir'` might contain the name of a directory specified by a user or a keyword, such as `STATA` or `BASE`. The appropriate directory name will be returned. The path is returned with a trailing separator.

Macro function for accessing operating-system parameters

`environment` *name*
returns the contents of the operating system's environment variable named *name*, or "" if *name* is undefined.

Macro functions for names of stored results

`e(scalars | macros | matrices | functions)`
returns the names of all the stored results in `e()` of the specified type, with the names listed one after the other and separated by one space. For instance, `e(scalars)` might return `N ll_0 ll df_m chi2 r2_p`, meaning that scalar stored results `e(N)`, `e(ll_0)`, ... exist.

`r(scalars | macros | matrices | functions)`
returns the names of all the stored results in `r()` of the specified type.

`s(macros)`
returns the names of all the stored results in `s()` of type macro, which is the only type that exists within `s()`.

`all` { `globals` | `scalars` | `matrices` } ["*pattern*"]
puts in *mname* the specified globals, scalars, or matrices that match the *pattern*, where the pattern matching is defined by Stata's `strmatch(`s_1`,`s_2`)` function; see [FN] **String functions**.

`all {numeric|string} scalars ["`*pattern*`"]`
puts in *mname* the specified numeric or string scalars that match the *pattern*, where the pattern matching is defined by Stata's `strmatch(`s_1`,`s_2`)` function; see [FN] **String functions**.

Macro function for formatting results

`display` *display_directive*
returns the results from the `display` command. The `display` function is the `display` command, except that the output is rerouted to a macro rather than to the screen.

You can use all the features of `display` that make sense. That is, you may not set styles with `as` *style* because macros do not have colors, you may not use `_continue` to suppress going to a new line on the real display (it is not being displayed), you may not use `_newline` (for the same reason), and you may not use `_request` to obtain input from the console (because input and output have nothing to do with macro definition). Everything else works. See [P] **display**.

Example:
```
local x : display %9.4f sqrt(2)
```

Macro function for manipulating lists

`list` *macrolist_directive*
fills in *mname* with the *macrolist_directive*, which specifies one of many available commands or operators for working with macros that contain lists; see [P] **macro lists**.

Macro functions related to matrices

In understanding the functions below, remember that the *fullname* of a matrix row or column is defined as *eqname*`:`*name*. For instance, *fullname* might be `outcome:weight`, and then the *eqname* is `outcome` and the *name* is `weight`. Or the *fullname* might be `gnp:L.cpi`, and then the *eqname* is `gnp` and the *name* is `L.cpi`. Or the *fullname* might be `mpg`, in which case the *eqname* is "" and the *name* is `mpg`. Or the *fullname* might be `gnp:1.south#1.smsa`, and then the *eqname* is `gnp` and the *name* is `1.south#1.smsa`. For more information, see [P] **matrix define**.

`rownames` *matname* `[, quoted]`
returns the names of the rows of *matname*, listed one after another and separated by one space. As many names are listed as there are rows of *matname*. `quoted` specifies that row names be enclosed in double quotes.

`colnames` *matname* `[, quoted]`
is like `rownames` but returns the names of the columns.

`rowfullnames` *matname* `[, quoted]`
returns the full names of the rows of *matname*, listed one after another and separated by one space. As many full names are listed as there are rows of *matname*. `quoted` specifies that full names be enclosed in double quotes.

`colfullnames` *matname* `[, quoted]`
is like `rowfullnames` but returns the full names of the columns.

`roweq` *matname* `[, quoted]`
returns the equation names of the columns of *matname*, listed one after another and separated by one space. As many names are listed as there are columns of *matname*. If the eqname of a column

is blank, _ (underscore) is substituted. Thus `roweq` might return "`Poor Poor Poor Average Average Average`" for one matrix and "`_ _ _ _ _ _`" for another. `quoted` specifies that equation names be enclosed in double quotes.

`coleq` *matname* [`, quoted`]
is like `roweq` but returns the equation names of the columns.

`rownumb` *matname string*
returns the row number of *matname* that matches *string*.

`colnumb` *matname string*
is like `rownumb` but returns the column number of *matname*.

`roweqnumb` *matname string*
returns the row equation number of *matname* that matches *string*.

`coleqnumb` *matname string*
is like `roweqnumb` but returns the column equation number of *matname*.

`rownfreeparms` *matname*
returns the number of free parameters in rows of *matname*.

`colnfreeparms` *matname*
returns the number of free parameters in columns of *matname*.

`rownlfs` *matname*
returns the number of linear forms among the rows of *matname*.

`colnlfs` *matname*
returns the number of linear forms among the columns of *matname*.

`rowsof` *matname*
returns the number of rows of *matname*.

`colsof` *matname*
returns the number of columns of *matname*.

`rowvarlist` *matname*
returns the variable list corresponding to the rows of *matname*.

`colvarlist` *matname*
returns the variable list corresponding to the columns of *matname*.

`rowlfnames` *matname* [`, quoted`]
returns the list of names corresponding to the linear forms in the rows of *matname*.

`collfnames` *matname* [`, quoted`]
returns the list of names corresponding to the linear forms in the columns of *matname*.

In all cases, *matname* may be either a Stata matrix name or a matrix stored in `e()` or `r()`, such as `e(b)` or `e(V)`.

Macro function related to time-series operators

`tsnorm` *string*
returns the canonical form of *string* when *string* is interpreted as a time-series operator. For instance, if *string* is `ldl`, then `L2D` is returned, or if *string* is `l.ldl`, then `L3D` is returned. If *string* is nothing, "" is returned.

`tsnorm` *string*`, varname`
returns the canonical form of *string* when *string* is interpreted as a time-series–operated variable. For instance, if *string* is `ldl.gnp`, then `L2D.gnp` is returned, or if string is `l.ldl.gnp`, then `L3D.gnp` is returned. If *string* is just a variable name, then the variable name is returned.

Macro function for copying a macro

`copy` { `local` | `global` } *mname*
returns a copy of the contents of *mname*, or an empty string if *mname* is undefined.

Macro functions for parsing

`word count` *string*
returns the number of tokens in *string*. A token is a word (characters separated by spaces) or set of words enclosed in quotes. Do not enclose *string* in double quotes because `word count` will return `1`.

`word #  of` *string*
returns the #th token of *string*. Do not enclose *string* in double quotes.

`piece` $\#_1$ $\#_2$ `of "`*string*`"` [`, nobreak`]
returns a piece of *string*. This macro function provides a smart method of breaking a string into pieces of roughly the specified display columns. $\#_1$ specifies which piece to obtain. $\#_2$ specifies the maximum number of display columns of each piece. Each piece is built trying to fill to the maximum number of display columns without breaking in the middle of a word. However, when a word takes more display columns than $\#_2$, the word will be split unless `nobreak` is specified. `nobreak` specifies that words not be broken, even if that would result in a string being displayed in more than $\#_2$ columns.

Compound double quotes may be used around *string* and must be used when *string* itself might contain double quotes.

`strlen` { `local` | `global` } *mname*
returns the length of the contents of *mname* in bytes. If *mname* is undefined, then `0` is returned. For instance,

```
. constraint 1 price = weight
. local myname : constraint 1
. macro list _myname
_myname         price = weight
. local lmyname : strlen local myname
. macro list _lmyname
_lmyname:       14
```

`ustrlen` { `local` | `global` } *mname*
returns the length of the contents of *mname* in Unicode characters. If *mname* is undefined, then `0` is returned.

`udstrlen` { `local` | `global` } *mname*
returns the length of the contents of *mname* in display columns. If *mname* is undefined, then `0` is returned.

`subinstr local` *mname* `"`*from*`" "`*to*`"`
returns the contents of *mname*, with the first occurrence of "*from*" changed to "*to*".

`subinstr local` *mname* `"`*from*`" "`*to*`", all`
does the same thing but changes all occurrences of "*from*" to "*to*".

`subinstr local` *mname* "*from*" "*to*"`, word`
returns the contents of *mname*, with the first occurrence of the word "*from*" changed to "*to*". A word is defined as a space-separated token or a token at the beginning or end of the string.

`subinstr local` *mname* "*from*" "*to*"`, all word`
does the same thing but changes all occurrences of the word "*from*" to "*to*".

`subinstr global` *mname* ...
is the same as the above but obtains the original string from the global macro `$mname` rather than from the local macro *mname*.

`subinstr` ... `global` *mname* ..., ... `count(`{`global`|`local`} *mname2*`)`
in addition to the usual, places a count of the number of substitutions in the specified global or in local macro *mname2*.

Example 1

```
. local string "a or b or c or d"
. global newstr : subinstr local string "c" "sand"
. display "$newstr"
a or b or sand or d
. local string2 : subinstr global newstr "or" "and", all count(local n)
. display "`string2'"
a and b and sand and d
. display "`n'"
3
. local string3: subinstr local string2 "and" "x", all word
. display "`string3'"
a x b x sand x d
```

The "and" in "sand" was not replaced by "x" because the `word` option was specified.

Macro expansion operators and function

There are five macro expansion operators that may be used within references to local (not global) macros.

‘*lclname*++’ and ‘++*lclname*’ provide inline incrementation of local macro *lclname*. For example,

```
. local x 5
. display "‘x++’"
5
. display "‘x’"
6
```

++ can be place before *lclname*, in which case *lclname* is incremented before ‘*lclname*’ is evaluated.

```
. local x 5
. display "‘++x’"
6
. display "‘x’"
6
```

‘*lclname*--’ and ‘--*lclname*’ provide inline decrementation of local macro *lclname*.

‘=*exp*’ provides inline access to Stata’s expression evaluator. The Stata expression *exp* is evaluated and the result substituted. For example,

```
. local alpha = 0.05
. regress mpg weight, level(‘=100*(1-‘alpha’)’)
```

‘:*macro_fcn*’ provides inline access to Stata’s macro functions. ‘:*macro_fcn*’ evaluates to the results of the macro function *macro_fcn*. For example,

```
. format ‘:format gear_ratio’ headroom
```

will set the display format of `headroom` to that of `gear_ratio`, which was obtained via the macro function `format`.

‘.*class_directive*’ provides inline access to class-object values. See [P] **class** for details.

The macro expansion function ‘`macval(name)`’ expands local macro `name` but not any macros contained within `name`. For instance, if `name` contained “`example ‘of’ macval`”, ‘`name`’ would expand to “`example macval`” (assuming that ‘`of`’ is not defined), whereas ‘`macval(name)`’ would expand to “`example ‘of’ macval`”. The ‘`of`’ would be left just as it is.

❑ Technical note

To store an unexpanded macro within another macro, use “`\`” to prevent macro expansion. This is useful when defining a formula with elements that will be substituted later in the program. To save the formula `sqrt(‘A’ + 1)`, where ‘`A`’ is a macro you would like to fill in later, you would use the command

```
. local formula sqrt(\‘A’ + 1)
```

which would produce

```
. macro list _formula
_formula:       sqrt(‘A’ + 1)
```

Because the statement `\‘A’` was used, it prevented Stata from expanding the macro ‘`A`’ when it stored it in the macro ‘`formula`’.

Now you can fill in the macro `‘A’` with different statements and have this be reflected when you call `‘formula’`.

```
. local A 2^3
. display "formula ‘formula’: " ‘formula’
formula sqrt(2^3 + 1): 3
. local A log10((‘A’ + 2)^3)
. display "formula ‘formula’: " ‘formula’
formula sqrt(log10((2^3 + 2)^3) + 1): 2
```

❑

The tempvar, tempname, and tempfile commands

The `tempvar`, `tempname`, and `tempfile` commands create names that may be used for temporary variables, temporary scalars and matrices, and temporary files. A temporary element exists while the program or do-file is running but, once it concludes, automatically ceases to exist.

Temporary variables

You are writing a program, and in the middle of it you need to calculate a new variable equal to $\texttt{var1}^2 + \texttt{var2}^2$ for use in the calculation. You might be tempted to write

```
(code omitted)
generate sumsq = var1^2 + var2^2
(code continues)
(code uses sumsq in subsequent calculations)
drop sumsq
```

This would be a poor idea. First, users of your program might already have a variable called `sumsq`, and if they did, your program would break at the `generate` statement with the error “sumsq already defined”. Second, your program in the subsequent code might call some other program, and perhaps that program also attempts (poorly) to create the variable `sumsq`. Third, even if nothing goes wrong, if users press *Break* after your code executes `generate` but before `drop`, you would confuse them by leaving behind the `sumsq` variable.

The way around these problems is to use temporary variables. Your code should read

```
(code omitted)
tempvar sumsq
generate ‘sumsq’ = var1^2 + var2^2
(code continues)
(code uses ‘sumsq’ in subsequent calculations)
(you do not bother to drop ‘sumsq’)
```

The `tempvar sumsq` command creates a local macro called `sumsq` and stores in it a name that is different from any name currently in the data. Subsequently, you then use `‘sumsq’` with single quotes around it rather than `sumsq` in your calculation, so that rather than naming your temporary variable `sumsq`, you are naming it whatever Stata wants you to name it. With that small change, your program works just as before.

Another advantage of temporary variables is that you do not have to drop them—Stata will do that for you when your program terminates, regardless of the reason for the termination. If a user presses *Break* after the `generate`, your program is stopped, the temporary variables are dropped, and things really are just as if the user had never run your program.

❑ Technical note

What do these temporary variable names assigned by Stata look like? It should not matter to you; however they look, they are guaranteed to be unique (`tempvar` will not hand out the same name to more than one concurrently executing program). Nevertheless, to satisfy your curiosity,

```
. tempvar var1 var2
. display "‘var1’ ‘var2’"
__000009 __00000A
```

Although we reveal the style of the names created by `tempvar`, you should not depend on this style. All that is important is that

- The names are unique; they differ from one call to the next.
- You should not prefix or suffix them with additional characters.
- Stata keeps track of any names created by `tempvar` and, when the program or do-file ends, searches the data for those names. Any variables found with those names are automatically dropped. This happens regardless of whether your program ends with an error.

❑

Temporary scalars and matrices

`tempname` is the equivalent of `tempvar` for obtaining names for scalars and matrices. This use is explained, with examples, in [P] **scalar**.

❑ Technical note

The temporary names created by `tempname` look just like those created by `tempvar`. The same cautions and features apply to `tempname` as `tempvar`:

- The names are unique; they differ from one call to the next.
- You should not prefix or suffix them with additional characters.
- Stata keeps track of any names created by `tempname` and, when the program or do-file ends, searches for scalars or matrices with those names. Any scalars or matrices so found are automatically dropped; see [P] **scalar**. This happens regardless of whether your program ends with an error.

❑

Temporary files

`tempfile` is the equivalent of `tempvar` for obtaining names for disk files. Before getting into that, let's discuss how you should not use `tempfile`. Sometimes, in the midst of your program, you will find it necessary to destroy the user's data to obtain your desired result. You do not want to change the data, but it cannot be helped, and therefore you would like to arrange things so that the user's original data are restored at the conclusion of your program.

You might then be tempted to save the user's data in a (temporary) file, do your damage, and then restore the data. You can do this, but it is complicated, because you then have to worry about the user pressing *Break* after you have stored the data and done the damage but have not yet restored the data. Working with `capture` (see [P] **capture**), you can program all of this, but you do not have to. Stata's `preserve` command (see [P] **preserve**) will handle saving and restoring the user's data, regardless of how your program ends.

Still, there may be times when you need temporary files. For example,

```
(code omitted)
preserve                          // preserve user's data
keep var1 var2 xvar
save master, replace
drop var2
save part1, replace
use master, clear
drop var1
rename var2 var1
append using part1
erase master.dta
erase part1.dta
(code continues)
```

This is poor code, even though it does use `preserve` so that, regardless of how this code concludes, the user's original data will be restored. It is poor because datasets called `master.dta` and `part1.dta` might already exist, and, if they do, this program will replace the user's (presumably valuable) data. It is also poor because, if the user presses *Break* before both (temporary) datasets are erased, they will be left behind to consume (presumably valuable) disk space.

Here is how the code should read:

```
(code omitted)
preserve                          // preserve user's data
keep var1 var2 xvar
tempfile master part1             // declare temporary files
save "`master'"
drop var2
save "`part1'"
use "`master'", clear
drop var1
rename var2 var1
append using "`part1'"
(code continues; temporary files are not erased)
```

In this version, Stata was asked to provide the names of temporary files in local macros named `master` and `part1`. We then put single quotes around `master` and `part1` wherever we referred to them so that, rather than using the names `master` and `part1`, we used the names Stata handed us. At the end of our program, we no longer bother to erase the temporary files. Because Stata gave us the temporary filenames, it knows that they are temporary and erases them for us if our program completes, has an error, or the user presses *Break*.

❑ Technical note

What do the temporary filenames look like? Again it should not matter to you, but for the curious,

```
. tempfile file1 file2
. display "`file1' `file2'"
/tmp/St13310.0001 /tmp/St13310.0002
```

We were using the Unix version of Stata; had we been using the Windows version, the last line might read

```
. display "`file1' `file2'"
C:\WIN\TEMP\ST_0a00000c.tmp C:\WIN\TEMP\ST_00000d.tmp
```

Under Windows, Stata uses the environment variable `TEMP` to determine where temporary files are to be located. This variable is typically set in your `autoexec.bat` file. Ours is set to `C:\WIN\TEMP`. If the variable is not defined, Stata places temporary files in your current directory.

Under Unix, Stata uses the environment variable TMPDIR to determine where temporary files are to be located. If the variable is not defined, Stata locates temporary files in /tmp.

Although we reveal the style of the names created by tempfile, just as with tempvar, you should not depend on it. tempfile produces names the operating system finds pleasing, and all that is important is that

- The names are unique; they differ from one call to the next.
- You should assume that they are so long that you cannot prefix or suffix them with additional characters and make use of them.
- Stata keeps track of any names created by tempfile, and, when your program or do-file ends, looks for files with those names. Any files found are automatically erased. This happens regardless of whether your program ends with an error.

❑

Manipulation of macros

macro dir and macro list list the names and contents of all defined macros; both do the same thing:

```
. macro list
S_FNDATE:       13 Apr 2020 17:45
S_FN:           C:\Program Files\Stata18\ado\base/a/auto.dta
tofname:        str18
S_level:        95
F1:             help advice;
F2:             describe;
F7:             save
F8:             use
S_MACH:         PC (64-bit x86-64)
S_OS:           Windows
S_OSDTL:        64-bit
S_StataSE:      SE
S_StataMP:      MP
S_ADO:          BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
_file2:         C:\WIN\Temp\ST_0a00000d.tmp
_file1:         C:\WIN\Temp\ST_0a00000c.tmp
_var2:          __00000A
_var1:          __000009
_str3:          a x b x sand x d
_dl:            Employee Data
_lbl:           Employee name
_vl:            sexlbl
_fmt:           %9.0g
```

macro drop eliminates macros from memory, although it is rarely used because most macros are local and automatically disappear when the program ends. Macros can also be eliminated by defining their contents to be nothing using global or local, but macro drop is more convenient.

Typing macro drop *base** drops all global macros whose names begin with *base*.

Typing macro drop _all eliminates all macros except system macros—those with names that begin with "S_".

Typing macro drop S_* does not drop all system macros that begin with "S_". It leaves certain macros in place that should not be casually deleted.

▷ Example 2

```
. macro drop _var* _lbl tofname _fmt
. macro list
S_FNDATE:       13 Apr 2020 17:45
S_FN:           C:\Program Files\Stata18\ado\base/a/auto.dta
S_level:        95
F1:             help advice;
F2:             describe;
F7:             save
F8:             use
S_MACH:         PC (64-bit x86-64)
S_OS:           Windows
S_OSDTL:        64-bit
S_StataSE:      SE
S_StataMP:      MP
S_ADO:          BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
_file2:         C:\WIN\Temp\ST_0a00000d.tmp
_file1:         C:\WIN\Temp\ST_0a00000c.tmp
_str3:          a x b x sand x d
_dl:            Employee Data
_vl:            sexlbl
. macro drop _all
. macro list
S_FNDATE:       13 Apr 2020 17:45
S_FN:           C:\Program Files\Stata18\ado\base/a/auto.dta
S_level:        95
S_MACH:         PC (64-bit x86-64)
S_OS:           Windows
S_OSDTL:        64-bits
S_StataSE:      SE
S_StataMP:      MP
S_ADO:          BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
. macro drop S_*
. macro list
S_level:        95
S_MACH:         PC (64-bit x86-64)
S_OS:           Windows
S_OSDTL:        64-bit
S_StataSE:      SE
S_StataMP:      MP
S_ADO:          BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
```

◁

❑ Technical note

Stata usually requires that you explicitly drop something before redefining it. For instance, before redefining a value label with the `label define` command or redefining a program with the `program define` command, you must type `label drop` or `program drop`. This way, you are protected from accidentally replacing something that might require considerable effort to reproduce.

Macros, however, may be redefined freely. It is *not* necessary to `drop` a macro before redefining it. Macros typically consist of short strings that could be easily reproduced if necessary. The inconvenience of the protection is not justified by the small benefit.

❑

Macros as arguments

Sometimes programs have in a macro a list of things—numbers, variable names, etc.—that you wish to access one at a time. For instance, after parsing (see **[U] 18.4 Program arguments**), you might have in the local macro `‘varlist’` a list of variable names. The `tokenize` command (see [P] **tokenize**) will take any macro containing a list and assign the elements to local macros named `‘1’`, `‘2’`, and so on. That is, if `‘varlist’` contained “`mpg weight displ`”, then coding

```
tokenize ‘varlist’
```

will make `‘1’` contain “`mpg`”, `‘2’` contain “`weight`”, `‘3’` contain “`displ`”, and `‘4’` contain “” (nothing). The empty fourth macro marks the end of the list.

`macro shift` can be used to work through these elements one at a time in constructs like

```
while "‘1’" != "" {
        do something based on ‘1’
        macro shift
}
```

`macro shift` discards `‘1’`, shifts `‘2’` to `‘1’`, `‘3’` to `‘2’`, and so on. For instance, in our example, after the first `macro shift`, `‘1’` will contain “`weight`”, `‘2’` will contain “`displ`”, and `‘3’` will contain “” (nothing).

It is better to avoid `macro shift` and instead code

```
local i = 1
while "‘‘i’’" != "" {
        do something based on ‘‘i’’
        local i = ‘i’ + 1
}
```

This second approach has the advantage that it is faster. Also what is in `‘1’`, `‘2’`, ... remains unchanged so that you can pass through the list multiple times without resetting it (coding “`tokenize ‘varlist’`” again).

It is even better to avoid `tokenize` and the numbered macros altogether and to instead loop over the variables in `‘varlist’` directly:

```
foreach var of local varlist {
        do something based on ‘var’
}
```

This is easier to understand and executes even more quickly; see [P] **foreach**.

`macro shift` # performs multiple macro shifts, or if # is 0, none at all. That is, `macro shift 2` is equivalent to two `macro shift` commands. `macro shift 0` does nothing.

Also see [P] **macro lists** for other list-processing commands.

References

Buis, M. L. 2015. Stata tip 124: Passing temporary variables to subprograms. *Stata Journal* 15: 597–598.

Cox, N. J. 2020. Stata tip 138: Local macros have local scope. *Stata Journal* 20: 499–503.

Also see

[P] **char** — Characteristics

[P] **creturn** — Return c-class values

[P] **display** — Display strings and values of scalar expressions

[P] **gettoken** — Low-level parsing

[P] **macro lists** — Manipulate lists

[P] **matrix** — Introduction to matrix commands

[P] **numlist** — Parse numeric lists

[P] **preserve** — Preserve and restore data

[P] **program** — Define and manipulate programs

[P] **return** — Return stored results

[P] **scalar** — Scalar variables

[P] **syntax** — Parse Stata syntax

[P] **tokenize** — Divide strings into tokens

[D] **fralias** — Alias variables from linked frames

[D] **frlink** — Link frames

[M-5] **st_global( )** — Obtain strings from and put strings into global macros

[M-5] **st_local( )** — Obtain strings from and put strings into Stata macros

[U] 12.8 Characteristics

[U] 18 Programming Stata

[U] 18.3 Macros

Stata Functions Reference Manual

Stata, Stata Press, and Mata are registered trademarks of StataCorp LLC. Stata and Stata Press are registered trademarks with the World Intellectual Property Organization of the United Nations. StataNow and NetCourseNow are trademarks of StataCorp LLC. Other brand and product names are registered trademarks or trademarks of their respective companies. Copyright © 1985–2023 StataCorp LLC, College Station, TX, USA. All rights reserved.

For suggested citations, see the FAQ on citing Stata documentation.