display — Display strings and values of scalar expressions

Syntax   Description   Remarks and examples   Also see

Syntax

`display` [*display_directive* [*display_directive* [...]]]

where *display_directive* is

`"`*double-quoted string*`"`

`` `" ``*compound double-quoted string*`"'`

[*%fmt*] [=]*exp*

`as` {`text` | `txt` | `result` | `error` | `input`}

`in smcl`

`_asis`

`_skip(`*#*`)`

`_column(`*#*`)`

`_newline`[`(`*#*`)`]

`_continue`

`_dup(`*#*`)`

`_request(`*macname*`)`

`_char(`*#*`)`

`,`

`,,`

Description

`display` displays strings and values of scalar expressions. `display` produces output from the programs that you write.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction
Styles
display used with quietly and noisily
Columns
display and SMCL
Displaying variable names
Obtaining input from the terminal

Introduction

Interactively, `display` can be used as a substitute for a hand calculator; see [R] **display**. You can type things such as `display 2+2`.

`display`'s *display_directive*s are used in do-files and programs to produce formatted output. The directives are

`"`*double-quoted string*`"`	displays the string without the quotes
`` `" ``*compound double-quoted string*`"'`	displays the string without the outer quotes; allows embedded quotes
[`%`*fmt*] [`=`]`exp`	allows results to be formatted; see **[U] 12.5 Formats: Controlling how data are displayed**
`as` *style*	sets the style (“color”) for the directives that follow; there may be more than one `as` *style* per `display`
`in smcl`	switches from `_asis` mode to `smcl` mode
`_asis`	switches from `smcl` mode to `_asis` mode
`_skip(`#`)`	skips # columns
`_column(`#`)`	skips to the #th column
`_newline`	goes to a new line
`_newline(`#`)`	skips # lines
`_continue`	suppresses automatic newline at end of `display` command
`_dup(`#`)`	repeats the next directive # times
`_request(`*macname*`)`	accepts input from the console and places it into the macro *macname*
`_char(`#`)`	displays the character for ASCII code #
`,`	displays one blank between two directives
`,,`	places no blanks between two directives

▷ Example 1

Here is a nonsense program called `silly` that illustrates the directives:

```
. program list silly
silly:
  1.      set obs 10
  2.      gen myvar=runiform()
  3.      di as text _dup(59) "-"
  4.      di "hello, world"
  5.      di %~59s "This is centered"
  6.      di "myvar[1] = " as result myvar[1]
  7.      di _col(10) "myvar[1] = " myvar[1] _skip(10) "myvar[2] = " myvar[2]
  8.      di "myvar[1]/myvar[2] = " %5.4f myvar[1]/myvar[2]
  9.      di "This" _newline _col(5) "That" _newline _col(10) "What"
 10.      di `"She said, "Hello""'
 11.      di substr("abcI can do string expressionsXYZ",4,27)
 12.      di _char(65) _char(83) _char(67) _char(73) _char(73)
 13.      di _dup(59) "-" " (good-bye)"
```

Here is the result of running it:

```
. silly
obs was 0, now 10
-----------------------------------------------------------
hello, world
                     This is centered
myvar[1] = .13698408
         myvar[1] = .13698408          myvar[2] = .64322066
myvar[1]/myvar[2] = 0.2130
This
    That
         What
She said, "Hello"
I can do string expressions
ASCII
----------------------------------------------------------- (good-bye)
```

◁

Styles

Stata has four styles: `text` (synonym `txt`), `result`, `error`, and `input`. Typically, these styles are rendered in terms of color,

`text` = black

`result` = black and bold

`error` = red

`input` = black and bold

or, at least, that is the default in the Results window when the window has a white background. On a black background, the defaults are

`text` = green

`result` = yellow

`error` = red

`input` = white

In any case, users can reset the styles by selecting **Edit > Preferences > General Preferences** in Windows or Unix(GUI) or by selecting **Preferences > General Preferences** in Mac.

The `display` directives `as text`, `as result`, `as error`, and `as input` allow you, the programmer, to specify in which rendition subsequent items in the `display` statement are to be displayed. So if a piece of your program reads

```
        quietly summarize mpg
        display as text "mean of mpg = " as result r(mean)
```

what might be displayed is

```
    mean of mpg = 21.432432
```

where, above, our use of boldface for the 21.432432 is to emphasize that it would be displayed differently from the “mean of mpg =” part. In the Results window, if we had a black background, the “mean of mpg =” part would be in green and the 21.432432 would be in yellow.

You can switch back and forth among styles within a `display` statement and between `display` statements. Here is how we recommend using the styles:

`as result` should be used to display things that depend on the data being used. For statistical output, think of what would happen if the names of the dataset remained the same but all the data changed. Clearly, calculated results would change. That is what should be displayed `as result`.

`as text` should be used to display the text around the results. Again think of the experiment where you change the data but not the names. Anything that would not change should be displayed `as text`. This will include not just the names but also table lines and borders, variable labels, etc.

`as error` should be reserved for displaying error messages. `as error` is special in that it not only displays the message as an error (probably meaning that the message is displayed in red) but also forces the message to display, even if output is being suppressed. (There are two commands for suppressing output: `quietly` and `capture`. `quietly` will not suppress `as error` messages but `capture` will, the idea being that `capture`, because it captures the return code, is anticipating errors and will take the appropriate action.)

`as input` should never be used unless you are creating a special effect. `as input` (white on a black background) is reserved for what the user types, and the output your program is producing is by definition not being typed by the user. Stata uses `as input` when it displays what the user types.

display used with quietly and noisily

`display`’s output will be suppressed by `quietly` at the appropriate times. Consider the following:

```
    . program list example1

    example1:
      1. di "hello there"

    . example1
    hello there

    . quietly example1

    . _
```

The output was suppressed because the program was run `quietly`. Messages displayed `as error`, however, are considered error messages and are always displayed:

```
. program list example2
example2:
  1.      di as error "hello there"
. example2
hello there
. quietly example2
hello there
```

Even though the program was run `quietly`, the message `as error` was displayed. Error messages should always be displayed `as error` so that they will always be displayed at the terminal.

Programs often have parts of their code buried in `capture` or `quietly` blocks. `display`s inside such blocks produce no output:

```
. program list example3
example3:
  1. quietly {
  2.      display "hello there"
  3. }
. example3

. _
```

If the `display` had included `as error`, the text would have been displayed, but only error output should be displayed that way. For regular output, the solution is to precede the `display` with `noisily`:

```
. program list example4
example4:
  1. quietly {
  2.       noisily display "hello there"
  3. }
. example4
hello there
```

This method also allows Stata to correctly treat a `quietly` specified by the caller:

```
. quietly example4

. _
```

Despite its name, `noisily` does not really guarantee that the output will be shown—it restores the output only if output would have been allowed at the instant the program was called.

For more information on `noisily` and `quietly`, see [P] **quietly**.

Columns

`display` can move only forward and downward. The directives that take a numeric argument allow only nonnegative integer arguments. It is not possible to back up to make an insertion in the output.

```
. program list cont
cont:
  1.      di "Stuff" _column(9) "More Stuff"
  2.      di "Stuff" _continue
  3.      di _column(9) "More Stuff"
. cont
Stuff   More Stuff
Stuff   More Stuff
```

display and SMCL

Stata Markup and Control Language (SMCL) is Stata's output formatter, and all Stata output passes through SMCL. See [P] **smcl** for a description. All the features of SMCL are available to `display` and so motivate you to turn to the SMCL section of this manual.

In our opening silly example, we included the line

```
        di as text _dup(59) "-"
```

That line would have better read

```
        di as text "{hline 59}"
```

The first `display` produces this:

```
-----------------------------------------------------------
```

and the second produces this:

```
───────────────────────────────────────────────────────────
```

It was not `display` that produced that solid line—`display` just displayed the characters `{hline 59}`. Output of Stata, however, passes through SMCL, and SMCL interprets what it hears. When SMCL heard `{hline 59}`, SMCL drew a horizontal line 59 characters wide.

SMCL has many other capabilities, including creating clickable links in your output that, when you click on them, can even execute other Stata commands.

If you carefully review the SMCL documentation, you will discover many overlap in the capabilities of SMCL and `display` that will lead you to wonder whether you should use `display`'s capabilities or SMCL's. For instance, in the section above, we demonstrated the use of `display`'s `_column()` feature to skip forward to a column. If you read the SMCL documentation, you will discover that SMCL has a similar feature, `{col}`. You can type

```
display "Stuff" _column(9) "More Stuff"
```

or you can type

```
display "Stuff{col 9}More Stuff"
```

So, which should you type? The answer is that it makes no difference and that when you use `display`'s `_column()` directive, `display` just outputs the corresponding SMCL `{col}` directive for you. This rule generalizes beyond `_column()`. For instance,

```
display as text "hello"
```

and

```
display "{text}hello"
```

are equivalent. There is, however, one important place where `display` and SMCL are different:

```
display as error "error message"
```

is not the same as

```
display "{error}error message"
```

Use `display as error`. The SMCL `{error}` directive sets the rendition to that of errors, but it does not tell Stata that the message is to be displayed, even if output is otherwise being suppressed. `display as error` both sets the rendition and tells Stata to override output suppression if that is relevant.

❑ Technical note

All Stata output passes through SMCL, and one side effect of that is that open and close brace characters, `{` and `}`, are treated oddly by `display`. Try the following:

```
display as text "{1, 2, 3}"
{1, 2, 3}
```

The result is just as you expect. Now try

```
display as text "{result}"
```

The result will be to display nothing because `{result}` is a SMCL directive. The first displayed something, even though it contained braces, because `{1, 2, 3}` is not a SMCL directive.

You want to be careful when displaying something that might itself contain braces. You can do that by using `display`'s `_asis` directive. Once you specify `_asis`, whatever follows in the display will be displayed exactly as it is, without SMCL interpretation:

```
display as text _asis "{result}"
{result}
```

You can switch back to allowing SMCL interpretation within the line by using the `in smcl` directive:

```
display as text _asis "{result}" in smcl "is a {bf:smcl} directive"
{result} is a smcl directive
```

Every `display` command in your program starts off in SMCL mode.

❑

Displaying variable names

Let's assume that a program we are writing is to produce a table that looks like this:

```
    Variable |       Obs        Mean   Std. Dev.        Min         Max
-------------+-----------------------------------------------------
         mpg |        74     21.2973   5.785503          12          41
      weight |        74    3019.459   777.1936        1760        4840
       displ |        74    197.2973   91.83722          79         425
```

Putting out the header in our program is easy enough:

```
di as text "    Variable {c |}     Obs" /*
        */ _col(37) "Mean   Std. Dev.       Min        Max"
di as text "{hline 13}{c +}{hline 53}"
```

We use the SMCL directive `{hline}` to draw the horizontal line, and we use the SMCL characters `{c |}` and `{c +}` to output the vertical bar and the "plus" sign where the lines cross.

Now let's turn to putting out the rest of the table. Variable names can be of unequal length and can even be long. If we are not careful, we might end up putting out something that looks like this:

```
    Variable |     Obs        Mean    Std. Dev.       Min        Max
-------------+------------------------------------------------------
miles_per_gallon |      74     21.2973   5.785503         12         41
weight |      74    3019.459   777.1936       1760       4840
displacement |      74   197.2973   91.83722         79        425
```

If it were not for the too-long variable name, we could avoid the problem by displaying our lines with something like this:

```
display as text %12s "`vname'" " {c |}" /*
        */ as result /*
        */ %8.0g `n' "    " /*
        */ %9.0g `mean' "  " %9.0g `sd'    "  " /*
        */ %9.0g `min'  "  " %9.0g `max'
```

What we are imagining here is that we write a subroutine to display a line of output and that the display line above appears in that subroutine:

```
program output_line
        args vname n mean sd min max
        display as text %12s "`vname'" " {c |}" /*
                */ as result /*
                */ %8.0g `n' "    " /*
                */ %9.0g `mean' "  " %9.0g `sd'    "  " /*
                */ %9.0g `min'  "  " %9.0g `max'
end
```

In our main routine, we would calculate results and then just call output_line with the variable name and results to be displayed. This subroutine would be sufficient to produce the following output:

```
    Variable |     Obs        Mean    Std. Dev.       Min        Max
-------------+------------------------------------------------------
miles_per_gallon |      74     21.2973   5.785503         12         41
      weight |      74   3019.459   777.1936       1760       4840
displacement |      74   197.2973   91.83722         79        425
```

The short variable name weight would be spaced over because we specified the %12s format. The right way to handle the miles_per_gallon variable is to display its abbreviation with Stata's abbrev() function:

```
program output_line
        args vname n mean sd min max
        display as text %12s abbrev("`vname'",12) " {c |}" /*
                */ as result /*
                */ %8.0g `n' "    " /*
                */ %9.0g `mean' "  " %9.0g `sd'    "  " /*
                */ %9.0g `min'  "  " %9.0g `max'
end
```

With this improved subroutine, we would get the following output:

```
    Variable |     Obs        Mean    Std. Dev.       Min        Max
-------------+------------------------------------------------------
miles_per_~n |      74     21.2973   5.785503         12         41
      weight |      74   3019.459   777.1936       1760       4840
displacement |      74   197.2973   91.83722         79        425
```

The point of this is to persuade you to learn about and use Stata's abbrev() function. abbrev("`vname'",12) returns `vname' abbreviated to 12 characters.

If we now wanted to modify our program to produce the following output,

```
     Variable │        Obs        Mean    Std. Dev.       Min        Max
─────────────┼─────────────────────────────────────────────────────
miles_per_~n │         74     21.2973    5.785503         12         41
      weight │         74    3019.459    777.1936       1760       4840
displacement │         74    197.2973    91.83722         79        425
─────────────┴─────────────────────────────────────────────────────
```

all we would need to do is add a `display` at the end of the main routine that reads

```
di as text "{hline 13}{c BT}{hline 53}"
```

Note the use of `{c BT}`. The characters that we use to draw lines in and around tables are summarized in [P] **smcl**.

❑ Technical note

Much of the output of Stata's official commands and of user-written commands is formatted to look good in a Results window that is 80 characters wide. If you write a Stata program that you want to share with others, we recommend that you design it such that its output will fit in an 80-character-wide Results window. The `abbrev()` function described above is useful for abbreviating variable names such that output tables fit within 80 columns.

Your program can determine the current width of the Results window by checking the value of `c(linesize)`. Some Stata commands, such as official estimation commands that output a coefficient table, use the value of `c(linesize)` to determine by how much, if at all, they need to abbreviate variable names.

We can modify the `output_line` program above to respect `c(linesize)`. For every character the Results window is wider than 80, we can allow our variable name abbreviation to be one character longer. If the Results window is 100 or more characters wide, we do not need to abbreviate variable names at all, because the maximum length of a variable name is 32 characters, and we were already able to display 12 characters of the variable name at a line size of 80.

```
program output_line
     args vname n mean sd min max
     if (c(linesize) >= 100)
         local abname = "‘vname’"

     else if (c(linesize) > 80)
         local abname = abbrev("‘vname’", 12+(c(linesize)-80))

     else
         local abname = abbrev("‘vname’", 12)

     local abname = abbrev("‘vname’",12)
     display as text %12s "‘abname’" " c |" /*
          */ as result                      /*
          */ %8.0g ‘n’ " "                  /*
          */ %9.0g ‘mean’ " " %9.0g ‘sd’ " "  /*
          */ %9.0g ‘min’ " " %9.0g ‘max’
 end
```

❑

❑ Technical note

Let’s now consider outputting the table in the form

```
     Variable │      Obs       Mean    Std. Dev.       Min        Max
──────────────┼──────────────────────────────────────────────────────
miles_per_~n  │       74    21.2973    5.785503        12         41
      weight  │       74   3019.459    777.1936      1760       4840
displacement  │       74   197.2973    91.83722        79        425
──────────────┴──────────────────────────────────────────────────────
```

where the boldfaced entries are clickable and, if you click on them, the result is to execute `summarize` followed by the variable name. We assume that you have already read [P] **smcl** and so know that the relevant SMCL directive to create the link is `{stata}`, but continue reading even if you have not read [P] **smcl**.

The obvious fix to our subroutine would be simply to add the `{stata}` directive, although to do that we will have to store `abbrev("‘vname’",12)` in a macro so that we can refer to it:

```
program output_line
        args vname n mean sd min max
        local abname = abbrev("‘vname’,12)
        display as text %12s "{stata summarize ‘vname’:‘abname’}" /*
                */ " {c |}" /*
                */ as result /*
                */ %8.0g ‘n’ "    " /*
                */ %9.0g ‘mean’ "   " %9.0g ‘sd’    "   " /*
                */ %9.0g ‘min’   "   " %9.0g ‘max’
end
```

The SMCL directive `{stata summarize ‘vname’:‘abname’}` says to display `‘abname’` as clickable, and, if the user clicks on it, to execute `summarize ‘vname’`. We used the abbreviated name to display and the unabbreviated name in the command.

The one problem with this fix is that our table will not align correctly because `display` does not know that “`{stata summarize ‘vname’:‘abname’}`” displays only `‘abname’`. To `display`, the string looks long and is not going to fit into a `%12s` field. The solution to that problem is

```
program output_line
        args vname n mean sd min max
        local abname = abbrev("‘vname’,12)
        display as text "{ralign 12:{stata summarize ‘vname’:‘abname’}}" /*
                */ " {c |}" /*
                */ as result /*
                */ %8.0g ‘n’ "    " /*
                */ %9.0g ‘mean’ "   " %9.0g ‘sd’    "   " /*
                */ %9.0g ‘min’   "   " %9.0g ‘max’
end
```

The SMCL `{ralign` *#*`:`*text*`}` macro right-aligns *text* in a field 12 wide and so is equivalent to `%12s`. The *text* that we are asking be aligned is “`{stata summarize ‘vname’:‘abname’}`”, but SMCL understands that the only displayable part of the string is `‘abname’` and so will align it correctly.

If we wanted to duplicate the effect of a `%-12s` format by using SMCL, we would use `{lalign 12:`*text*`}`.

❑

Obtaining input from the terminal

display's _request(*macname*) option accepts input from the console and places it into the macro *macname*. For example,

```
. display "What is Y? " _request(yval)
What is Y? i don't know

. display "$yval"
i don't know
```

If yval had to be a number, the code fragment to obtain it might be

```
global yval "junk"
capture confirm number $yval
while _rc!=0 {
     display "What is Y? " _request(yval)
     capture confirm number $yval
}
```

You will typically want to store such input into a local macro. Local macros have names that really begin with a '_':

```
local yval "junk"
capture confirm number `yval'
while _rc!=0 {
     display "What is Y? " _request(_yval)
     capture confirm number `yval'
}
```

Also see

[P] **capture** — Capture return code

[P] **quietly** — Quietly and noisily perform Stata command

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

[P] **smcl** — Stata Markup and Control Language

[D] **list** — List values of variables

[D] **outfile** — Export dataset in text format

[U] 12.5 Formats: Controlling how data are displayed

[U] 18 Programming Stata