Title

stata.com

smcl — Stata Markup and Control Language

Description Remarks and examples Also see

Description

SMCL, which stands for Stata Markup and Control Language and is pronounced "smickle", is Stata's output language. SMCL directives, such as "`{it:...}`" in

```
You can output {it:italics} using SMCL
```

affect how output appears:

You can output *italics* using SMCL

All Stata output is processed by SMCL: help files, statistical results, and even the output of `display` (see [P] **display**) in the programs you write.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

You will use SMCL mainly in the programs you compose and in the help files you write to document them, although you can use it in any context. Everything Stata displays on the screen is processed by SMCL. You can even use some of SMCL's features to change how text appears in graphs; see [G-4] ***text***.

Your first encounter with SMCL was probably in the Stata session logs created by the `log using` command. By default, Stata creates logs in SMCL format and gives them the file suffix `.smcl`. The file suffix does not matter; that the output is in SMCL format does. Files containing SMCL can be redisplayed in their original rendition, and SMCL output can be translated to other formats through the `translate` command; see [R] **translate**.

SMCL is mostly just plain text, for instance,

```
. display "this is SMCL"
this is SMCL
```

but that text can contain SMCL directives, which are enclosed in braces. Try the following:

```
. display "{title:this is SMCL, too}"
this is SMCL, too
```

The "`{title:...}`" directive told SMCL to output what followed the colon in title format. Exactly how the title format appears on your screen—or on paper if you print it—will vary, but SMCL will ensure that it always appears as a recognizable title.

Now try this:

```
. display "now we will try {help summarize:clicking}"
now we will try clicking
```

The word *clicking* will appear as a link—probably in some shade of blue. Click on the word. This will bring up Stata's Viewer and show you the help for the `summarize` command. The SMCL `{help:...}` directive is an example of a *link*. The directive `{help summarize:clicking}` displayed the word *clicking* and arranged things so that when the user clicked on the highlighted word, help for `summarize` appeared.

Here is another example of a link:

```
. display "You can also run Stata commands by {stata summarize mpg:clicking}"
You can also run Stata commands by clicking
```

Click on the word, and this time the result will be exactly as if you had typed the command `summarize mpg` into Stata. If you have the automobile data loaded, you will see the summary statistics for the variable `mpg`.

Simply put, you can use SMCL to make your output look better and to add links.

SMCL modes

SMCL is always in one of three modes:

1. SMCL line mode
2. SMCL paragraph mode
3. As-is mode

Modes 1 and 2 are nearly alike—in these two modes, SMCL directives are understood, and the modes differ only in how they treat blanks and carriage returns. In paragraph mode—so called because it is useful for formatting text into paragraphs—SMCL joins one line to the next and splits lines to form output with lines that are of nearly equal length. In line mode, SMCL shows the line much as you entered it. For instance, in line mode, the input text

```
Variable name         mean          standard error
```

(which might appear in a help file) would be spaced in the output exactly as you entered it. In paragraph mode, the above would be output as "Variable name mean standard error", meaning that it would all run together. On the other hand, the text

```
The two main uses of SMCL are in the programs you compose and in the help files
you write to document them, although SMCL may be used in any context.
Everything Stata displays on the screen is processed by SMCL.
```

would display as a nicely formatted paragraph in paragraph mode.

In mode 3, as-is mode, SMCL directives are not interpreted. `{title:...}`, for instance, has no special meaning—it is just the characters open brace, t, i, and so on. If `{title:...}` appeared in SMCL input text,

```
{title:My Title}
```

it would be displayed exactly as it appears: {title:My Title}. In as-is mode, SMCL just displays text as it was entered. As-is mode is useful only for those wishing to document how SMCL works because, with as-is mode, they can show examples of what SMCL input looks like.

Those are the three modes, and the most important of them are the first two, the SMCL modes, and the single most important mode is SMCL line mode—mode 1. Line mode is the mother of all modes in that SMCL continually returns to it, and you can get to the other modes only from line mode. For instance, to enter paragraph mode, you use the {p} directive, and you use it from line mode, although you typically do not think of that. Paragraphs end when SMCL encounters a blank line, and SMCL then returns to line mode. Consider the following lines appearing in some help file:

```
{p}
The two main uses of SMCL are in the programs you compose and the
help files you write to document them, although SMCL may be used in any context.
Everything Stata displays on the screen is processed by SMCL.

{p}
Your first encounter with SMCL was probably the Stata session
...
```

Between the paragraphs above, SMCL returned to line mode because it encountered a blank line. SMCL stayed in paragraph mode as long as the paragraph continued without a blank line, but once the paragraph ended, SMCL returned to line mode. There are ways of ending paragraphs other than using blank lines, but they are the most common. Regardless of how paragraphs end, SMCL returns to line mode.

In another part of our help file, we might have

```
{p}
SMCL, which stands for Stata Markup and Control Language
and is pronounced "smickle", is Stata's output language.
SMCL directives, for example, the {c -(}it:...{c )-} in the following,

        One can output {it:italics} using SMCL

{p} affects how output appears:  ...
```

Between the paragraphs, SMCL entered line mode (again, because SMCL encountered a blank line), so the "One can output..." part will appear as you have spaced it, namely, indented. It will appear that way because SMCL is in line mode.

The other mode is invoked using the {asis} directive and does not end with a blank line. It continues until you enter the {smcl} directive, and here {smcl} must be followed by a carriage return. You may put a carriage return at the end of the {asis} directive—it will make no difference—but to return to SMCL line mode, you must put a carriage return directly after the {smcl} directive.

To summarize, when dealing with SMCL, begin by assuming that you are in line mode; you almost certainly will be. If you wish to enter a paragraph, you will use the {p} directive, but once the paragraph ends, you will be back in line mode and ready to start another paragraph. If you want to enter as-is mode, perhaps to include a piece of text output, use the {asis} directive, and at the end of the piece, use the {smcl}(carriage return) directive to return to line mode.

Command summary—general syntax

Pretend that {xyz} is a SMCL directive, although it is not. {xyz} might have any of the following syntaxes:

Syntax 1: {xyz}

Syntax 2: {xyz:*text*}

Syntax 3: {xyz *args*}

Syntax 4: {xyz *args*:*text*}

Syntax 1 means "do whatever it is that {xyz} does". Syntax 2 means "do whatever it is that {xyz} does, do it on the text *text*, and then stop doing it". Syntax 3 means "do whatever it is that {xyz} does, as modified by *args*". Finally, syntax 4 means "do whatever it is that {xyz} does, as modified by *args*, do it on the text *text*, and then stop doing it".

Not every SMCL directive has all four syntaxes, and which syntaxes are allowed is made clear in the descriptions below.

In syntaxes 3 and 4, *text* may contain other SMCL directives, so the following is valid:

```
{center:The use of {ul:SMCL} in help files}
```

The *text* of one SMCL directive may itself contain other SMCL directives. However, the braces must not only match but also match on the same physical (input) line. Typing

```
{center:The use of {ul:SMCL} in help files}
```

is correct, but

```
{center:The use of {ul:SMCL} in
help files}
```

is an error. When SMCL encounters an error, it simply displays the text in the output it does not understand, so the result of making the error above would be to display

```
{center:The use of SMCL in
help files}
```

SMCL understood {ul:...} but not {center:...} because the braces did not match on the input line, so it displayed only that part. If you see SMCL directives in your output, you have made an error.

Help file preprocessor directive for substituting repeated material

`INCLUDE help` *arg* follows syntax 3.

`INCLUDE` specifies that SMCL substitute the contents of a file named *arg*`.ihlp`. This is useful when you need to include the same text multiple times. This substitution is performed only when the file is viewed using `help`.

Example:
We have several commands that accept the `replace` option. Instead of typing the description under *Options* of each help file, we create the file `replace.ihlp`, which contains something like the following:

```
{* 01apr2019}{...}
{phang}
{opt replace} overwrite existing {it:filename}{p_end}
```

To include the text in our help file, we type

```
INCLUDE help replace
```

Formatting directives for use in line and paragraph modes

`{sf}`, `{it}`, and `{bf}` follow syntaxes 1 and 2.

These directives specify how the font is to appear. `{sf}` indicates standard face, `{it}` italic face, and `{bf}` boldface.

Used in syntax 1, these directives switch to the font face specified, and that rendition will continue to be used until another one of the directives is given.

Used in syntax 2, they display *text* in the specified way and then switch the font face back to whatever it was previously.

Examples:

```
the value of {it}varlist {sf}may be specified ...
the value of {it:varlist} may be specified ...
```

`{input}`, `{error}`, `{result}`, and `{text}` follow syntaxes 1 and 2.

These directives specify how the text should be rendered: in the style that indicates user input, an error, a calculated result, or the text around calculated results.

These styles are often rendered as color. In the Results window, on a white background, Stata by default shows input in black and bold, error messages in red, calculated results in black and bold, and text in black. However, the relationship between the real colors and `{input}`, `{error}`, `{result}`, and `{text}` may not be the default (the user could reset it), and, in fact, these renditions may not be shown in color at all. The user might have set `{result}`, for instance, to show in yellow, or in highlight, or in something else. However the styles are rendered, SMCL tries to distinguish among `{input}`, `{error}`, `{result}`, and `{text}`.

Examples:

```
{text}the variable mpg has mean {result:21.3} in the sample.
{text}mpg     {c |} {result}21.3
{text}mpg     {c |} {result:21.3}
{error:variable not found}
```

{inp}, {err}, {res}, and {txt} follow syntaxes 1 and 2.
These four commands are synonyms for {input}, {error}, {result}, and {text}.

Examples:

```
{txt}the variable mpg has mean {res:21.3} in the sample.
{txt}mpg     {c |} {res}21.3
{txt}mpg     {c |} {res:21.3}
{err:variable not found}
```

{cmd} follows syntaxes 1 and 2.
{cmd} is similar to the "color" styles and is the recommended way to show Stata commands in help files. Do not confuse {cmd} with {inp}. {inp} is the way commands actually typed are shown, and {cmd} is the recommended way to show commands you might type. We recommend that you present help files in terms of {txt} and use {cmd} to show commands; use any of {sf}, {it}, or {bf} in a help file, but we recommend that you not use any of the "colors" {inp}, {err}, or {res}, except where you are showing actual Stata output.

Example:

```
When using the {cmd:summarize} command, specify ...
```

{cmdab:*text1*:*text2*} follows a variation on syntax 2 (note the double colons).
{cmdab} is the recommended way to show minimum abbreviations for Stata commands and options in help files; *text1* represents the minimum abbreviation, and *text2* represents the rest of the text. When the entire command or option name is the minimum abbreviation, you may omit *text2* along with the extra colon. {cmdab:*text*} is then equivalent to {cmd:*text*}; it makes no difference which you use.

Examples:

```
{cmdab:su:mmarize} [{it:varlist}] [{it:weight}] [{cmdab:if} {it:exp}]
the option {cmdab:ef:orm}{cmd:({it:varname})} ...
```

{opt *option*}, {opt *option*(*arg*)}, {opt *option*(*a*,*b*)}, and {opt *option*(*a*|*b*)} follow syntax 3; alternatives to using {cmd}.
{opt *option1*:*option2*}, {opt *option1*:*option2*(*arg*)}, {opt *option1*:*option2*(*a*,*b*)}, and {opt *option1*:*option2*(*a*|*b*)} follow syntaxes 3 and 4; alternatives to using {cmdab}.
{opt} is the recommended way to show options. {opt} allows you to easily include arguments.

SMCL directive ...	is equivalent to typing ...
{opt *option*}	{cmd:*option*}
{opt *option*(*arg*)}	{cmd:*option*(}{it:*arg*}{cmd:)}
{opt *option*(*a*,*b*)}	{cmd:*option*(}{it:*a*}{cmd:,}{it:*b*}{cmd:)}
{opt *option*(*a*|*b*)}	{cmd:*option*(}{it:*a*}{cmd:|}{it:*b*}{cmd:)}
{opt *option1*:*option2*}	{cmd:*option1*:*option2*}
{opt *option1*:*option2*(*arg*)}	{cmd:*option1*:*option2*(}{it:*arg*}{cmd:)}
{opt *option1*:*option2*(*a*,*b*)}	{cmd:*option1*:*option2*(}{it:*a*}{cmd:,}{it:*b*}{cmd:)}
{opt *option1*:*option2*(*a*|*b*)}	{cmd:*option1*:*option2*(}{it:*a*}{cmd:|}{it:*b*}{cmd:)}

option1 represents the minimum abbreviation, and *option2* represents the rest of the text.

a,*b* and *a*|*b* may have any number of elements. Available elements that are displayed in {cmd} style are ,, =, :, *, %, and (). Several elements are displayed in plain text style: |, { }, and [ ].

Also, {opth *option*(*arg*)} is equivalent to {opt}, except that *arg* is displayed as a link to help; see *Link directives for use in line and paragraph modes* for more details.

Examples:

```
{opt replace}
{opt bseunit(varname)}
{opt f:ormat}
{opt sep:arator(#)}
```

`{hilite}` and `{hi}` follow syntaxes 1 and 2.

`{hilite}` and `{hi}` are synonyms. `{hilite}` is the recommended way to highlight (draw attention to) something in help files. You might highlight, for example, a reference to a manual, the *Stata Journal*, or a book.

Examples:

```
see {hilite:[R] anova} for more details.
see {hi:[R] anova} for more details.
```

`{ul}` follows syntaxes 2 and 3.

`{ul on}` starts underlining mode. `{ul off}` ends it. `{ul:`*text*`}` underlines *text*.

Examples:

```
You can {ul on}underline{ul off} this way or
you can {ul:underline} this way
```

`{*}` follows syntaxes 2 and 4.

`{*}` indicates a comment. What follows it (inside the braces) is ignored.

Examples:

```
{* this text will be ignored}
{*:as will this}
```

`{hline}` follows syntaxes 1 and 3.

`{hline}` (syntax 1) draws a horizontal line the rest of the way across the page.
`{hline #}` (syntax 3) draws a horizontal line of # characters.
`{hline}` (either syntax) is generally used in line mode.

Examples:

```
{hline}
{hline 20}
```

`{.-}` follows syntax 1.

`{.-}` is a synonym for `{hline}` (syntax 1).

Example:

```
{.-}
```

`{dup #:`*text*`}` follows syntax 4.

`{dup}` repeats *text* # times.

Examples:

```
{dup 20:A}
{dup 20:ABC}
```

{char *code*} and {c *code*} are synonyms and follow syntax 3.
These directives display the specified characters that otherwise might be difficult to type on your keyboard. See *Displaying characters using ASCII and extended ASCII codes* below.

Examples:
```
C{c o'}rdoba es una joya arquitect{c o'}nica.
{c S|}57.20
The ASCII character 206 in the current font is {c 206}
The ASCII character 5a (hex) is {c 0x5a}
{c -(} is open brace and {c )-} is close brace
```

{reset} follows syntax 1.
{reset} is equivalent to coding {txt}{sf}.

Example:
```
{reset}
```

Link directives for use in line and paragraph modes

All the link commands share the feature that when syntax 4 is allowed,

Syntax 4: {xyz *args*:*text*}

then syntax 3 is also allowed,

Syntax 3: {xyz *args*}

and if you specify syntax 3, Stata treats it as if you specified syntax 4, inserting a colon and then repeating the argument. For instance, {help} is defined below as allowing syntaxes 3 and 4. Thus the directive

```
{help summarize}
```

is equivalent to the directive

```
{help summarize:summarize}
```

Coding {help summarize} or {help summarize:summarize} both display the word *summarize*, and if the user clicks on that, the action of help summarize is taken. Thus you might code

```
See help for {help summarize} for more information.
```

This would display "See help for **summarize** for more information" and make the word *summarize* a link. To make the words describing the action different from the action, use syntax 4,

```
You can also {help summarize:examine the summary statistics} if you wish.
```

which results in "You can also **examine the summary statistics** if you wish."

The link directives, which may be used in either line mode or paragraph mode, are the following:

{help *args*[:*text*]} follows syntaxes 3 and 4.
{help} displays *args* as a link to help *args*; see [R] **help**. If you also specify the optional :*text*, *text* is displayed instead of *args*, but you are still directed to the help file for *args*.

Examples:
```
{help epitab}
{help summarize:the mean}
```

{helpb *args*[:*text*]} follows syntaxes 3 and 4.
{helpb} is equivalent to {help}, except that *args* or *text* is displayed in boldface.

Examples:
```
{helpb summarize}
{helpb generate}
```

{manhelp *args1 args2*[:*text*]} follows syntaxes 3 and 4.
{manhelp} displays [*args2*] *args1* as a link to help *args1*; thus our first example below would display [R] summarize as a link to help summarize. Specifying the optional :*text* displays *text* instead of *args1*, but you are still directed to the help file for *args1*.

Examples:
```
{manhelp summarize R}
{manhelp weight U:14 Language syntax}
{manhelp graph_twoway G:graph twoway}
```

{manhelpi *args1 args2*[:*text*]} follows syntaxes 3 and 4.
{manhelpi} is equivalent to {manhelp}, except that *args* or *text* is displayed in italics.

Examples:
```
{manhelpi twoway_options G}
{manhelpi mata M:Mata Reference Manual}
```

{help *args*##*markername*[|*viewername*] [:*text*]} and {marker *markername*} follow syntax 3.
They let the user jump to a specific location within a file, not just to the top of the file. {help *args*##*markername*} displays *args*##*markername* as a link that will jump to the location marked by {marker *markername*}. Specifying the optional |*viewername* will display the results of {marker *markername*} in a new Viewer window named *viewername*; _new is a valid *viewername* that assigns a unique name for the new Viewer. Specifying the optional :*text* displays *text* instead of *args*##*markername*. *args* represents the name of the file where the {marker} is located. If *args* contains spaces, be sure to specify it within quotes.

We document the directive as {help ...}; however, view, net, ado, and update may be used in place of help, although you would probably want to use only help or view.

Examples:

```
{pstd}You can change the style of the text using the {cmd}
directive; see {help example##cmd} below.

You can underline a word or phrase with the {ul} directive;
see {help example##ul:below}.

{marker cmd}{...}
{phang}{cmd} follows syntaxes 1 and 2.{break}
{cmd} is another style not unlike the  ...

{marker ul}{...}
{phang}{ul} follows syntaxes 2 and 3.{break}
{ul on} starts underlining mode.  {ul} ...
```

{help_d:*text*} follows syntax 2.
{help_d} displays *text* as a link that will display a help dialog box from which the user may obtain interactive help on any Stata command.

Example:
```
... using the {help_d:help system} ...
```

`{newvar[:`*args*`]}` follows syntaxes 1 and 2.
`{newvar}` displays *newvar* as a link to `help newvar`. If you also specify the optional `:`*args*, Stata concatenates *args* to *newvar* to display *newvarargs*.

Examples:
```
{newvar}
{newvar:2}
```

`{var[:`*args*`]}` and `{varname[:`*args*`]}` follow syntaxes 1 and 2.
`{var}` and `{varname}` display *varname* as a link to `help varname`. If you also specify the optional `:`*args*, Stata concatenates *args* to *varname* to display *varnameargs*.

Examples:
```
{var}
{var:1}
{varname}
{varname:2}
```

`{vars[:`*args*`]}` and `{varlist[:`*args*`]}` follow syntaxes 1 and 2.
`{vars}` and `{varlist}` display *varlist* as a link to `help varlist`. If you also specify the optional `:`*args*, Stata concatenates *args* to *varlist* to product *varlistargs*.

Examples:
```
{vars}
{vars:1}
{varlist}
{varlist:2}
```

`{depvar[:`*args*`]}` follows syntaxes 1 and 2.
`{depvar}` displays *depvar* as a link to `help depvar`. If you also specify the optional `:`*args*, Stata concatenates *args* to *depvar* to display *depvarargs*.

Examples:
```
{depvar}
{depvar:1}
```

`{depvars[:`*args*`]}` and `{depvarlist[:`*args*`]}` follow syntaxes 1 and 2.
`{depvars}` and `{depvarlist}` display *depvarlist* as a link to `help depvarlist`. If you also specify the optional `:`*args*, Stata concatenates *args* to *depvarlist* to display *depvarlistargs*.

Examples:
```
{depvars}
{depvars:1}
{depvarlist}
{depvarlist:2}
```

`{indepvars[:`*args*`]}` follows syntaxes 1 and 2.
`{indepvars}` displays *indepvars* as a link to `help varlist`. If you also specify the optional `:`*args*, Stata concatenates *args* to *indepvars* to display *indepvarsargs*.

Examples:
```
{indepvars}
{indepvars:1}
```

{ifin} follows syntax 1.
{ifin} displays [*if*] and [*in*], where *if* is a link to the help for the if qualifier and *in* is a link to the help for the in qualifier.

Example:
{ifin}

{weight} follows syntax 1.
{weight} displays [*weight*], where *weight* is a link to the help for the *weight* specification.

Example:
{weight}

{dtype} follows syntax 1.
{dtype} displays [*type*], where *type* is a link to help data types.

Example:
{dtype}

{search *args*[:*text*]} follows syntaxes 3 and 4.
{search} displays *text* as a link that will display the results of search on *args*; see [R] **search**.

Examples:
{search anova:click here} for the latest information on ANOVA
Various programs are available for {search anova}

{search_d:*text*} follows syntax 2.
{search_d} displays *text* as a link that will display a *Keyword Search* dialog box from which the user can obtain interactive help by entering keywords of choice.

Example:
... using the {search_d:search system} ...

{dialog *args*[:*text*]} follows syntaxes 3 and 4.
{dialog} displays *text* as a link that will launch the dialog box for *args*. *args* must contain the name of the dialog box and may optionally contain , message(*string*), where *string* is the message to be passed to the dialog box.

Example:
... open the {dialog regress:regress dialog box} ...

{browse *args*[:*text*]} follows syntaxes 3 and 4.
{browse} displays *text* as a link that will launch the user's browser pointing at *args*. Because *args* is typically a URL containing a colon, *args* usually must be specified within quotes.

Example:
... you can {browse "https://www.stata.com":visit the Stata website} ...

{view *args*[:*text*]} follows syntaxes 3 and 4.
{view} displays *text* as a link that will present in the Viewer the filename *args*. If *args* is a URL, be sure to specify it within quotes. {view} is seldom used in a SMCL file (such as a help file) because you will seldom know of a fixed location for the file unless it is a URL. {view} is sometimes used from programs because the program knows the location of the file it created.

{view} can also be used with {marker}; see {help *args*##*markername*[|*viewername*] [:*text*]} and {marker *markername*}, earlier in this section.

Examples:

```
see {view "https://www.stata.com/man/readme.smcl"}
display `"{view "`newfile'":click here} to view the file created"'
```

`{view_d:`*text*`}` follows syntax 2.

`{view_d}` displays *text* as a link that will display the *Choose File to View* dialog box in which the user may type the name of a file or a URL to be displayed in the Viewer.

Example:

```
{view_d:Click here} to view your current log
```

`{manpage` *args*[`:`*text*]`}` follows syntaxes 3 and 4.

`{manpage}` displays *text* as a link that will launch the user's PDF viewer pointing at *args*. *args* are a Stata manual (such as `R` or `SVY`) and a page number. The page number is optional. If the page number is not specified, the PDF viewer will open to the first page of the file.

Example:

```
The formulas are given on {manpage R 342:page 342 of [R] manual}.
```

`{mansection` *args*[`:`*text*]`}` follows syntaxes 3 and 4.

`{mansection}` displays *text* as a link that will launch the user's PDF viewer pointing at *args*. *args* are a Stata manual (such as `R` or `SVY`) and a named destination within that manual (such as `predict` or `regress postestimation`). The named destination is optional. If the named destination is not specified, the PDF viewer will open to the first page of the file.

Example:

```
See {mansection R clogitpostestimation:[R] clogit postestimation}.
```

`{manlink` *man entry*`}` and `{manlinki` *man entry*`}` follow syntax 3.

`{manlink}` and `{manlinki}` display *man* and *entry* using the `{mansection}` directive as a link that will launch the user's PDF viewer pointing at that manual entry. *man* is a Stata manual (such as `R` or `SVY`) and *entry* is the name of an entry within that manual (such as `predict` or `regress postestimation`). The named destination should be written as it appears in the title of the manual entry.

SMCL directive ...	is equivalent to typing ...
`{manlink` *man entry*`}`	`{bf:{mansection` *man entry_ns*`:[`*man*`]` *entry*`}}`
`{manlinki` *man entry*`}`	`{bf:{mansection` *man entry_ns*`:[`*man*`]` `{it:`*entry*`}}}`

entry_ns is *entry* with the following characters removed: space, left and right quotes (`‘` and `’`), `#`, `$`, `~`, `{`, `}`, `[`, and `]`.

{net *args*[:*text*]} follows syntaxes 3 and 4.

{net} displays *args* as a link that will display in the Viewer the results of net *args*; see [R] **net**. Specifying the optional :*text*, displays *text* instead of *args*. For security reasons, net get and net install cannot be executed in this way. Instead, use {net describe ...} to show the page, and from there, the user can click on the appropriate links to install the materials. Whenever *args* contains a colon, as it does when *args* is a URL, be sure to enclose *args* within quotes.

{net cd .:*text*} displays *text* as a link that will display the contents of the current net location.

{net} can also be used with {marker}; see {help *args*##*markername*[|*viewername*] [:*text*]} and {marker *markername*}, earlier in this section.

Examples:

```
programs are available from {net "from https://www.stata.com":Stata}
Nicholas Cox has written a series of matrix commands which you can obtain
by {net "describe https://www.stata.com/stb/stb56/dm79":clicking here}.
```

{net_d:*text*} follows syntax 2.

{net_d} displays *text* as a link that will display a *Keyword Search* dialog box from which the user can search the Internet for additions to Stata.

Example:

```
To search the Internet for the latest additions to Stata available,
{net_d:click here}.
```

{netfrom_d:*text*} follows syntax 2.

{netfrom_d} displays *text* as a link that will display a *Choose Download Site* dialog box into which the user may enter a URL and then see the contents of the site. This directive is seldom used.

Example:

```
If you already know the URL, {netfrom_d:click here}.
```

{ado *args*[:*text*]} follows syntaxes 3 and 4.

{ado} displays *text* as a link that will display in the Viewer the results of ado *args*; see [R] **net**. For security reasons, ado uninstall cannot be executed in this way. Instead, use {ado describe ...} to show the package, and from there, the user can click to uninstall (delete) the material.

{ado} can also be used with {marker}; see {help *args*##*markername*[|*viewername*] [:*text*]} and {marker *markername*}, earlier in this section.

Example:

```
You can see the community-contributed packages you have installed (and uninstall
any that you wish) by {ado dir:clicking here}.
```

{ado_d:*text*} follows syntax 2.

{ado_d} displays *text* as a link that will display a *Search Installed Programs* dialog box from which the user can search for community-contributed routines previously installed (and uninstall them if desired).

Example:

```
You can search the community-contributed ado-files you have installed
by {ado_d:clicking here}.
```

{update *args*[:*text*]} follows syntaxes 3 and 4.

{update} displays *text* as a link that will display in the Viewer the results of update *args*; see [R] **update**. If *args* contains a URL, be careful to place the *args* in quotes.

args can be omitted because the `update` command is valid without arguments. `{update:`*text*`}` is really the best way to use the `{update}` directive because it allows the user to choose whether and from where to update their Stata.

`{update}` can also be used with `{marker}`; see `{help` *args*`##`*markername*[`|`*viewername*] [`:`*text*]`}` and `{marker` *markername*`}`, earlier in this section.

Examples:

```
Check whether your Stata is {update:up to date}.
Check whether your Stata is {update "from https://www.stata.com":up to date}.
```

`{update_d:`*text*`}` follows syntax 2.

`{update_d}` displays *text* as a link that will display a *Choose Official Update Site* dialog box into which the user may type a source (typically https://www.stata.com, but perhaps a local CD drive) from which to install official updates to Stata.

Example:

```
If you are installing from CD or some other source,
{update_d:click here}.
```

`{back:`*text*`}` follows syntax 2.

`{back}` displays *text* as a link that will take an action equivalent to pressing the Viewer's **Back** button.

Example:

```
{back:go back to the previous page}
```

`{clearmore:`*text*`}` follows syntax 2.

`{clearmore}` displays *text* as a link that will take an action equivalent to pressing Stata's **Clear –more– Condition** button. `{clearmore}` is of little use to anyone but the developers of Stata.

Example:

```
{clearmore:{hline 2}more{hline 2}}
```

`{stata` *args*[`:`*text*]`}` follows syntaxes 3 and 4.

`{stata}` displays *text* as a link that will execute the Stata command *args* in the Results window. Stata will first ask before executing a command that is displayed in a web browser. If *args* (the Stata command) contains a colon, remember to enclose the command in quotes.

Example:

```
... {stata summarize mpg:to obtain the mean of mpg}...
```

Remember, like all SMCL directives, `{stata}` can be used in programs as well as files. Thus you could code

```
display "... {stata summarize mpg:to obtain the mean of mpg}..."
```

or, if you were in the midst of outputting a table,

```
di "{stata summarize mpg:mpg}       {c |}" ...
```

However, it is more likely that, rather than being hardcoded, the variable name would be in a macro, say, `‘vn’`:

```
di "{stata summarize ‘vn’:‘vn’}       {c |}" ...
```

Here you probably would not know how many blanks to put after the variable name because it could be of any length. Thus you might code

```
di "{ralign 12:{stata summ ‘vn’:‘vn’}} {c |}" ...
```

thus allocating 12 spaces for the variable name, which would be followed by a blank and the vertical bar. Then you would want to allow for a ‘vn’ longer than 12 characters:

```
local vna = abbrev(‘vn’,12)
di "{ralign 12:{stata summ ‘vn’:‘vna’}} {c |}" ...
```

There you have a line that will output a part of a table, with the linked variable name on the left and with the result of clicking on the variable name being to `summ ‘vn’`. Of course, you could make the action whatever else you wanted.

`{matacmd` *args*[`:`*text*]`}` follows syntaxes 3 and 4.

`{matacmd}` works the same as `{stata}`, except that it submits a command to Mata. If Mata is not already active, the command will be prefixed with `mata` to allow Stata to execute it.

Formatting directives for use in line mode

`{title:`*text*`}`(carriage return) follows syntax 2.

`{title:`*text*`}` displays *text* as a title. `{title:...}` should be followed by a carriage return and, usually, by one more blank line so that the title is offset from what follows. (In help files, we precede titles by two blank lines and follow them by one.)

Example:

```
{title:Command summary -- general syntax}

{p}
Pretend that {cmd:{c -({xyz}c )-}} is a SMCL directive, although ...
```

`{center:`*text*`}` and `{centre:`*text*`}` follow syntax 2.
`{center` #`:`*text*`}` and `{centre` #`:`*text*`}` follow syntax 4.

`{center:`*text*`}` and `{centre:`*text*`}` are synonyms; they center the text on the line. `{center:`*text*`}` should usually be followed by a carriage return; otherwise, any text that follows it will appear on the same line. With syntax 4, the directives center the text in a field of width #.

Examples:

```
{center:This text will be centered}
{center:This text will be centered} and this will follow it
{center 60:This text will be centered within a width of 60 columns}
```

`{rcenter:`*text*`}` and `{rcentre:`*text*`}` follow syntax 2.
`{rcenter` #`:`*text*`}` and `{rcentre` #`:`*text*`}` follow syntax 4.

`{rcenter:`*text*`}` and `{rcentre:`*text*`}` are synonyms. `{rcenter}` is equivalent to `{center}`, except that *text* is displayed one space to the right when there are unequal spaces left and right. `{rcenter:`*text*`}` should be followed by a carriage return; otherwise, any text that follows it will appear on the same line. With syntax 4, the directives center the text in a field of width #.

Example:

```
{rcenter:this is shifted right one character}
```

`{right:`*text*`}` follows syntax 2.

`{right}` displays *text* with its last character aligned on the right margin. `{right:`*text*`}` should be followed by a carriage return.

Examples:

```
{right:this is right-aligned}
{right:this is shifted left one character }
```

{lalign #:*text*} and {ralign #:*text*} follow syntax 4.
{lalign} left-aligns *text* in a field # characters wide, and {ralign} right-aligns *text* in a field # characters wide.

Example:
```
{lalign 12:mpg}{ralign 15:21.2973}
```

{dlgtab [# [#]]:*text*} follows syntaxes 2 and 4.
{dlgtab} displays *text* as a dialog tab. The first # specifies how many characters to indent the dialog tab from the left-hand side, and the second # specifies how much to indent from the right-hand side. The default is {dlgtab 4 2:*text*}.

Examples:
```
{dlgtab:Model}
{dlgtab 8 2:Model}
```

{...} follows syntax 1.
{...} specifies that the next carriage return be treated as a blank.

Example:
```
Sometimes you need to type a long line and, while {...}
that is fine with SMCL, some word processors balk. {...}
In line mode, the above will appear as one long line to SMCL.
```

{col #} follows syntax 3.
{col #} skips forward to column #. If you are already at or beyond that column in the output, then {col #} does nothing.

Example:
```
mpg{col 20}21.3{col 30}5.79
```

{space #} follows syntax 3.
{space} is equivalent to typing # blank characters.

Example:
```
20.5{space 20}17.5
```

{tab} follows syntax 1.
{tab} has the same effect as typing a tab character. Tab stops are set every eight spaces.

Examples:
```
{tab}This begins one tab stop in
{tab}{tab}This begins two tab stops in
```

Note: SMCL also understands tab characters and treats them the same as the {tab} command, so you may include tabs in your files.

Formatting directives for use in paragraph mode

{p} follows syntax 3. The full syntax is {p # # # #}.
{p # # # #} enters paragraph mode. The first # specifies how many characters to indent the first line; the second #, how much to indent the second and subsequent lines; the third #, how much to bring in the right margin on all lines; and the fourth # is the total width for the paragraph. Numbers, if not specified, default to zero, so typing {p} without numbers is equivalent to typing {p 0 0 0 0}, {p #} is equivalent to {p # 0 0 0}, and so on. A zero for the fourth # means use the default paragraph width; see set linesize in [R] **log**. {p} (with or without numbers) may be followed by a carriage return or not; it makes no difference.

Paragraph mode ends when a blank line is encountered, the {p_end} directive is encountered, or {smcl}(carriage return) is encountered.

Examples:

```
{p}
{p 4}
{p 0 4}
{p 8 8 8 60}
```

Note concerning paragraph mode: In paragraph mode, you can have either one space or two spaces at the end of sentences, following the characters '.', '?', '!', and ':'. In the output, SMCL puts two spaces after each of those characters if you put two or more spaces after them in your input, or if you put a carriage return; SMCL puts one space if you put one space. Thus

```
{p}
Dr. Smith was near panic.  He could not reproduce the result.
Now he wished he had read about logging output in Stata.
```

will display as

```
Dr. Smith was near panic.  He could not reproduce the result.  Now he wished he
had read about logging output in Stata.
```

Several shortcut directives have also been added for commonly used paragraph mode settings:

SMCL directive ...	is equivalent to typing ...
{pstd}	{p 4 4 2}
{psee}	{p 4 13 2}
{phang}	{p 4 8 2}
{pmore}	{p 8 8 2}
{pin}	{p 8 8 2}
{phang2}	{p 8 12 2}
{pmore2}	{p 12 12 2}
{pin2}	{p 12 12 2}
{phang3}	{p 12 16 2}
{pmore3}	{p 16 16 2}
{pin3}	{p 16 16 2}

{p_end} follows syntax 1.

{p_end} is a way of ending a paragraph without having a blank line between paragraphs. {p_end} may be followed by a carriage return or not; it will make no difference in the output.

Example:

```
{p_end}
```

{p2colset # # # #} follows syntax 3.
{p2col [# # # #] : [*first_column_text*] } [*second_column_text*] follows syntaxes 2 and 4.
{p2line [# #]} follows syntaxes 1 and 3.
{p2colreset} follows syntax 1.

{p2colset} sets column spacing for a two-column table. The first # specifies the beginning position of the first column, the second # specifies the placement of the second column, the third # specifies the placement for subsequent lines of the second column, and the last # specifies the number to indent from the right-hand side for the second column.

{p2col} specifies the rows that make up the two-column table. Specifying the optional numbers redefines the numbers specified in {p2colset} for this row only. If the *first_column_text* or the *second_column_text* is not specified, the respective column is left blank.

{p2line} draws a dashed line for use with a two-column table. The first # specifies the left indentation, and the second # specifies the right indentation. If no numbers are specified, the defaults are based on the numbers provided in {p2colset}.

{p2colreset} restores the {p2col} default values.

Examples:

```
{p2colset 9 26 27 2}{...}
{p2col:{keyword}}rules{p_end}
{p2line}
{p2col:{opt nonm:issing}}all nonmissing values not changed by the
rules{p_end}
{p2col 7 26 27 2:* {opt m:issing}}all missing values not changed by
the rules{p_end}
{p2line}
{p2colreset}{...}
```

{synoptset [#] [tabbed|notes]} follows syntaxes 1 and 3.
{synopthdr: [*first_column_header*]} follows syntaxes 1 and 2.
{syntab:*text*} follows syntax 2.
{synopt: [*first_column_text*]} [*second_column_text*] follows syntax 2.
{p2coldent: [*first_column_text*] } [*second_column_text*] follows syntax 2.
{synoptline} follows syntax 1.

{synoptset} sets standard column spacing for a two-column table used to document options in syntax diagrams. # specifies the width of the first column; the width defaults to 20 if # is not specified. The optional argument tabbed specifies that the table will contain headings or "tabs" for sets of options. The optional argument notes specifies that some of the table entries will have footnotes and results in a larger indentation of the first column than the tabbed argument implies.

{synopthdr} displays a standard header for a syntax-diagram-option table. *first_column_header* is used to title the first column in the header; if *first_column_header* is not specified, the default title "*options*" is displayed. The second column is always titled "Description".

{syntab} displays *text* positioned as a subheading or "tab" in a syntax-diagram-option table.

{synopt} specifies the rows that make up the two-column table; it is equivalent to {p2col} (see above).

{p2coldent} is the same as {synopt}, except the *first_column_text* is displayed with the standard indentation (which may be negative). The *second_column_text* is displayed in paragraph mode and ends when a blank line, {p_end}, or a carriage return is encountered. The location of the columns is determined by a prior {synoptset} or {p2colset} directive.

{synoptline} draws a horizontal line that extends to the boundaries of the previous {synoptset} or, less often, {p2colset} directive.

Examples:

```
{synoptset 21 tabbed}{...}
{synopthdr}
{synoptline}
{syntab:Model}
{p2coldent:* {opth a:bsorb(varname)}}categorical variable to be absorbed{p_end}
{synopt:{opt clear}}reminder that untransposed data will be lost if not previously
saved{p_end}
{synoptline}
{p2colreset}{...}
```

{bind:*text*} follows syntax 2.

{bind:...} keeps *text* together on a line, even if that makes one line of the paragraph unusually short. {bind:...} can also be used to insert one or more real spaces into the paragraph if you specify *text* as one or more spaces.

Example:

```
Commonly, bind is used {bind:to keep words together} on a line.
```

{break} follows syntax 1.

{break} forces a line break without ending the paragraph.

Example:

```
{p 4 8 4}
{it:Example:}{break}
Commonly, ...
```

Directive for entering the as-is mode

{asis} follows syntax 1.

{asis} begins as-is mode, which continues until {smcl}(carriage return) is encountered. {asis} may be followed by a carriage return or not; it makes no difference, but {smcl} must be immediately followed by a carriage return. {smcl} returns SMCL to line mode. No other SMCL commands are interpreted in as-is mode.

Inserting values from constant and current-value class

The {ccl} directive outputs the value contained in a constant and current-value class (c()) object. For instance, {ccl pi} provides the value of the constant pi (3.14159...) contained in c(pi). See [P] **creturn** for a list of all the available c() objects.

Displaying characters using ASCII and extended ASCII codes

The {char} directive—synonym {c}—allows you to output any ASCII or extended ASCII character in Latin1 encoding. Extended ASCII characters in Latin1 encoding are converted to the equivalent Unicode characters in the UTF-8 encoding. For instance, {c 232} is equivalent to typing the letter è because extended ASCII code 232 in Latin1 is defined as the letter "e" with a grave accent. You may also type the Unicode character è (code point \u00e8) directly.

You can get to all the ASCII and extended ASCII characters in Latin1 encoding by typing {c #}, where # is between 1 and 255. Or, if you prefer, you can type {c 0x#}, where # is a hexadecimal number between 1 and ff. Thus {c 0x6a} is also j because the hexadecimal number 6a is equal to the decimal number 106.

Also, so that you do not have to remember the numbers, {c} provides special codes for characters that are, for one reason or another, difficult to type. These include

{c S|}	$ (dollar sign)
{c 'g}	` (open single quote)
{c -(}	{ (left curly brace)
{c)-}	} (right curly brace)

{c S|} and {c 'g} are included not because they are difficult to type or cause SMCL any problems but because in Stata display statements, they can be difficult to display, since they are Stata's macro substitution characters and tend to be interpreted by Stata. For instance,

```
. display "shown in $US"
shown in
```

drops the $US part because Stata interpreted $US as a macro, and the global macro was undefined. A way around this problem is to code

```
. display "shown in {c S|}US"
shown in $US
```

{c -(} and {c)-} are included because { and } are used to enclose SMCL directives. Although { and } have special meaning to SMCL, SMCL usually displays the two characters correctly when they do not have a special meaning. SMCL follows the rule that, when it does not understand what it thinks ought to be a directive, it shows what it did not understand in unmodified form. Thus

```
. display "among the alternatives {1, 2, 4, 7}"
among the alternatives {1, 2, 4, 7}
```

works, but

```
. display "in the set {result}"
in the set
```

does not because SMCL interpreted {result} as a SMCL directive to set the output style (color) to that for results. The way to code the above is to type

```
. display "in the set {c -(}result{c )-}"
in the set {result}
```

SMCL also provides the following line-drawing characters:

{c -}	−	a wide dash character
{c |}	|	a tall | character
{c +}	┼	a wide dash on top of a tall |
{c TT}	┬	a top T
{c BT}	┴	a bottom T
{c LT}	├	a left T
{c RT}	┤	a right T
{c TLC}	┌	a top-left corner
{c TRC}	┐	a top-right corner
{c BRC}	┘	a bottom-right corner
{c BLC}	└	a bottom-left corner

{hline} constructs the line by using the {c -} character. The above are not really characters; they are instructions to SMCL to draw lines. The "characters" are, however, one character wide and one character tall, so you can use them as characters in your output. The result is that Stata output that appears on your screen can look like

```
. summarize mpg weight
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         74     21.2973    5.785503         12         41
      weight |         74    3019.459    777.1936       1760       4840
```

but, if the result is translated into plain text, it will look like

```
. summarize mpg weight
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         74     21.2973    5.785503         12         41
      weight |         74    3019.459    777.1936       1760       4840
```

because SMCL will be forced to restrict itself to the characters.

Finally, SMCL provides the following Western European characters:

{c a'}	á	{c e'}	é	{c i'}	í	{c o'}	ó	{c u'}	ú
{c A'}	Á	{c E'}	É	{c I'}	Í	{c O'}	Ó	{c U'}	Ú
{c a'g}	à	{c e'g}	è	{c i'g}	ì	{c o'g}	ò	{c u'g}	ù
{c A'g}	À	{c E'g}	È	{c I'g}	Ì	{c O'g}	Ò	{c U'g}	Ù
{c a^}	â	{c e^}	ê	{c i^}	î	{c o^}	ô	{c u^}	û
{c A^}	Â	{c E^}	Ê	{c I^}	Î	{c O^}	Ô	{c U^}	Û
{c a~}	ã					{c o~}	õ		
{c A~}	Ã					{c O~}	Õ		
{c a:}	ä	{c e:}	ë	{c i:}	ï	{c o:}	ö	{c u:}	ü
{c A:}	Ä	{c E:}	Ë	{c I:}	Ï	{c O:}	Ö	{c U:}	Ü
{c ae}	æ	{c c,}	ç	{c n~}	ñ	{c o/}	ø	{c y'}	ý
{c AE}	Æ	{c C,}	ç	{c N~}	Ñ	{c O/}	Ø	{c Y'}	Ý
{c y:}	ÿ	{c ss}	ß	{c r?}	¿	{c r!}	¡		
{c L-}	£	{c Y=}	(yen)						

SMCL uses UTF-8 to render the above characters. For instance, {c e'} is equivalent to {c 0xe9}, if you care to look it up. {c 0xe9} will display as é if you are using a Latin1 encoding.

Advice on using display

Do not think twice; you can just use SMCL directives in your display statements, and they will work. What we are really talking about, however, is programming, and there are two things to know.

First, remember how display lets you display results as text, as result, as input, and as error, with the abbreviations as txt, as res, as inp, and as err. For instance, a program might contain the lines

```
program ...
        ...
        quietly summarize ‘varname’
        display as txt "the mean of ‘varname’ is " as res r(mean)
        ...
end
```

Results would be the same if you coded the display statement

```
        display "{txt}the mean of ‘varname’ is {res}" r(mean)
```

That is, the display directive as txt just sends {txt} to SMCL, the display directive as res just sends {res} to SMCL, and so on.

However, as err does not just send {err}. as err also tells Stata that what is about to be displayed is an error message so that, if output is being suppressed, Stata knows to display this message anyway. For example,

```
        display as err "varname undefined"
```

is the right way to issue the error message "varname undefined".

```
display "{err}varname undefined"
```

would not work as well; if the program's output were suppressed, the error message would not be displayed because Stata would not know to stop suppressing output. You could code

```
display as err "{err}varname undefined"
```

but that is redundant. `display`'s `as error` directive both tells Stata that this is an error message and sends the `{err}` directive to SMCL. The last part makes output appear in the form of error messages, probably in red. The first part is what guarantees that the error message appears, even if output is being suppressed.

If you think about this, you will now realize that you could code

```
display as err "{txt}varname undefined"
```

to produce an error message that would appear as ordinary text (meaning that it would probably be in black) and yet still display in all cases. Please do not do this. By convention, all error messages should be displayed in SMCL's `{err}` (default red) rendition.

The second thing to know is how Stata sets the state of SMCL the instant before `display` displays its output. When you use `display` interactively—when you use it at the keyboard or in a do-file—Stata sets SMCL in line mode, font face `{sf}`, and style `{res}`. For instance, if you type

```
. display 2+2
4
```

the 4 will appear in `{sf}{res}`, meaning in standard font face and in result style, which probably means in black and bold. On the other hand, consider the following:

```
. program demonstrate_display
  1. display 2+2
  2. end
. demonstrate_display
4
```

Here the 4 will appear in {sf}{inp}, meaning that the result is probably also shown in black and bold. However, if your preferences are set to display input differently than results, the output from the program will be different from the interactive output.

When display is executed from inside a program, no changes are made to SMCL. SMCL is just left in the mode it happens to be in, and here it happened to be in line mode {sf}{inp} because that was the mode it was in after the user typed the command demonstrate_display.

This is an important feature of display because it means that, in your programs, one display can pick up where the last left off. Perhaps you have four or five displays in a row that produce the text to appear in a paragraph. The first display might begin paragraph mode, and the rest of the displays finish it off, with the last display displaying a blank line to end paragraph mode. Here it is of great importance that SMCL stay in the mode you left it in between displays.

That leaves only the question of what mode SMCL is in when your program begins. You should assume that SMCL is in line mode but make no assumptions about the style (color) {txt}, {res}, {err}, or {inp}. Within a program, all display commands should be coded as

```
display as ... ...
```

or

```
display "one of {txt}, {res}, {err}, or {inp} ..." ...
```

although you may violate this rule if you really intend one display to pick up where another left off. For example,

```
display as text "{p}"
display "This display violates the rule, but that is all right"
display "because it is setting a paragraph, and we want all"
display "these displays to be treated as a whole."
display "We did follow the rule with the first display in the"
display "sequence."
display
display "Now we are back in line mode because of the blank line"
```

You could even code

```
program example2
        display as text "{p}"
        display "Below we will call a subroutine to contribute a sentence"
        display "to this paragraph being constructed by example2:"
        example2_subroutine
        display "The text that example2_subroutine contributed became"
        display "part of this single paragraph.  Now we will end the paragraph."
        display
end
program example2_subroutine
        display "This sentence is being displayed by"
        display "example2_subroutine."
end
```

The result of running this would be

```
. example2
Below we will call a subroutine to contribute a sentence to this paragraph
being constructed by example2: This sentence is being displayed by
example2_subroutine.  The text that example2_subroutine contributed became
part of this single paragraph.  Now we will end the paragraph.
```

Advice on formatting help files

Help files are just files named *filename*`.sthlp` that Stata displays when the user types "`help` *filename*". The first line of a help file should read

```
{smcl}
```

After that, it is a matter of style. To see examples of our style, type

`. viewsource assert.sthlp`	(simple example with a couple of options)
`. viewsource centile.sthlp`	(example with an options table)
`. viewsource regress.sthlp`	(example of an estimation command)
`. viewsource regress_postestimation.sthlp`	(example of a postestimation entry)

We recommend opening a second Viewer window (one way is to right-click within an existing Viewer and select "Open New Viewer") to look at the help file and the raw source file side by side.

Also see

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

[RPT] **dyndoc** — Convert dynamic Markdown document to HTML or Word (.docx) document

[RPT] **markdown** — Convert Markdown document to HTML file or Word (.docx) document

[R] **log** — Echo copy of session to file