

## Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [16. Generic Operating System Services](#) »

## 16.8. curses — Terminal handling for character-cell displays¶

Platforms: Unix

Changed in version 1.6: Added support for the `ncurses` library and converted to a package.

The `curses` module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While `curses` is most widely used in the Unix environment, versions are available for DOS, OS/2, and possibly other systems as well. This extension module is designed to match the API of `ncurses`, an open-source curses library hosted on Linux and the BSD variants of Unix.

### Note

Since version 5.4, the `ncurses` library decides how to interpret non-ASCII data using the `nl_langinfo` function. That means that you have to call `locale.setlocale()` in the application and encode Unicode strings using one of the system's available encodings. This example uses the system's default encoding:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Then use `code` as the encoding for `str.encode()` calls.

See also

Module [curses.ascii](#)

Utilities for working with ASCII characters, regardless of your locale settings.

Module [curses.panel](#)

A panel stack extension that adds depth to curses windows.

Module `curses.textpad`

Editable text widget for curses supporting **Emacs**-like bindings.

Module `curses.wrapper`

Convenience function to ensure proper terminal setup and resetting on application entry and exit.

[Curses Programming with Python](#)

Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The `Demo/curses/` directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

### 16.8.1. Functions¶

The module `curses` defines the following exception:

*exception* `curses.error`¶

Exception raised when a curses library function returns an error.

### Note

Whenever `x` or `y` arguments to a function or a method are optional, they default to the current cursor location. Whenever `attr` is optional, it defaults to `A_NORMAL`.

The module `curses` defines the following functions:

`curses.baudrate()`¶

Returns the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

`curses.beep()`¶

Emit a short attention sound.

`curses.can_change_color()`¶

Returns true or false, depending on whether the programmer can change the colors displayed by the terminal.

`curses.cbreak()`

Enter cbreak mode. In cbreak mode (sometimes called “rare” mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first [raw\(\)](#) then [cbreak\(\)](#) leaves the terminal in cbreak mode.

`curses.color_content(color_number)`

Returns the intensity of the red, green, and blue (RGB) components in the color *color\_number*, which must be between 0 and `COLORS`. A 3-tuple is returned, containing the R,G,B values for the given color, which will be between 0 (no component) and 1000 (maximum amount of component).

`curses.color_pair(color_number)`

Returns the attribute value for displaying text in the specified color. This attribute value can be combined with `A_STANDOUT`, `A_REVERSE`, and the other `A_*` attributes. [pair\\_number\(\)](#) is the counterpart to this function.

`curses.curs_set(visibility)`

Sets the cursor state. *visibility* can be set to 0, 1, or 2, for invisible, normal, or very visible. If the terminal supports the visibility requested, the previous cursor state is returned; otherwise, an exception is raised. On many terminals, the “visible” mode is an underline cursor and the “very visible” mode is a block cursor.

`curses.def_prog_mode()`

Saves the current terminal mode as the “program” mode, the mode when the running program is using curses. (Its counterpart is the “shell” mode, for when the program is not in curses.) Subsequent calls to [reset\\_prog\\_mode\(\)](#) will restore this mode.

`curses.def_shell_mode()`

Saves the current terminal mode as the “shell” mode, the mode when the running program is not using curses. (Its counterpart is the “program” mode, when the program is using curses capabilities.) Subsequent calls to [reset\\_shell\\_mode\(\)](#) will restore this mode.

`curses.delay_output(ms)`

Inserts an *ms* millisecond pause in output.

`curses.doupdate()`

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The [doupdate\(\)](#) ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a [noutrefresh\(\)](#) call after write operations such as [addstr\(\)](#) have been performed on a window. The normal [refresh\(\)](#) call is simply [noutrefresh\(\)](#) followed by [doupdate\(\)](#); if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing [noutrefresh\(\)](#) calls on all windows, followed by a single [doupdate\(\)](#).

`curses.echo()`

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

`curses.endwin()`

De-initialize the library, and return terminal to normal status.

`curses.erasechar()`

Returns the user’s current erase character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.filter()`

The [filter\(\)](#) routine, if used, must be called before [initscr\(\)](#) is called. The effect is that, during those calls, `LINES` is set to 1; the capabilities `clear`, `cup`, `cud`, `cud1`, `cuu1`, `cuu`, `vpa` are disabled; and the home string is set to the value of `cr`. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

`curses.flash()`

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as ‘visible bell’ to the audible attention signal produced by [beep\(\)](#).

`curses.flushinp()`

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

`curses.getmouse()`

After [getch\(\)](#) returns `KEY_MOUSE` to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (*id*, *x*, *y*, *z*, *bstate*). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event’s coordinates. (*z* is currently unused.). *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 4: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

`curses.getsyx()`

Returns the current coordinates of the virtual screen cursor in *y* and *x*. If `leaveok` is currently true, then -1,-1 is returned.

`curses.getwin(file)`

Reads window related data stored in the file by an earlier [putwin\(\)](#) call. The routine then creates and initializes a new window using that data, returning the new window object.

`curses.has_colors()`

Returns true if the terminal can display colors; otherwise, it returns false.

`curses.has_ic()`

Returns true if the terminal has insert- and delete- character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.has_il()`

Returns true if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

`curses.has_key(ch)`

Takes a key value *ch*, and returns true if the current terminal type recognizes a key with that value.

`curses.halfdelay(tenths)`

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, an exception is raised if nothing has been typed. The value of *tenths* must be a number between 1 and 255. Use [nocbreak\(\)](#) to leave half-delay mode.

`curses.init_color(color_number, r, g, b)`

Changes the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color\_number* must be between 0 and COLORS. Each of *r*, *g*, *b*, must be a value between 0 and 1000. When [init\\_color\(\)](#) is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if [can\\_change\\_color\(\)](#) returns 1.

`curses.init_pair(pair_number, fg, bg)`

Changes the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair\_number* must be between 1 and COLOR\_PAIRS - 1 (the 0 color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between 0 and COLORS. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

`curses.initscr()`

Initialize the library. Returns a `WindowObject` which represents the whole screen.

#### Note

If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

`curses.isendwin()`

Returns true if [endwin\(\)](#) has been called (that is, the curses library has been deinitialized).

`curses.keyname(k)`

Return the name of the key numbered *k*. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-character string consisting of a caret followed by the corresponding printable ASCII character. The name of an alt-key combination (128-255) is a string consisting of the prefix 'M-' followed by the name of the corresponding ASCII character.

`curses.killchar()`

Returns the user's current line kill character. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.longname()`

Returns a string containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to [initscr\(\)](#).

`curses.meta(yes)`

If *yes* is 1, allow 8-bit characters to be input. If *yes* is 0, allow only 7-bit chars.

`curses.mouseinterval(interval)`

Sets the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and returns the previous interval value. The default value is 200 msec, or one fifth of a second.

`curses.mousemask(mousemask)`

Sets the mouse events to be reported, and returns a tuple (*availmask*, *oldmask*). *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns 0. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

`curses.napms(ms)`

Sleep for *ms* milliseconds.

`curses.newpad(nlines, ncols)`

Creates and returns a pointer to a new pad data structure with the given number of lines and columns. A pad is returned as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The [refresh\(\)](#) and [noutrefresh\(\)](#) methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

`curses.newwin([nlines, ncols], begin_y, begin_x)`

Return a new window, whose left-upper corner is at (*begin\_y*, *begin\_x*), and whose height/width is *nlines/ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

`curses.nl()`

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

`curses.nocbreak()`

Leave cbreak mode. Return to normal “cooked” mode with line buffering.

`curses.noecho()`

Leave echo mode. Echoing of input characters is turned off.

`curses.nonl()`

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch( '\n' )`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

`curses.noqiflush()`

When the `noqiflush` routine is used, normal flush of input and output queues associated with the INTR, QUIT and SUSP characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

`curses.noraw()`

Leave raw mode. Return to normal “cooked” mode with line buffering.

`curses.pair_content(pair_number)`

Returns a tuple (fg, bg) containing the colors for the requested color pair. The value of *pair\_number* must be between 1 and `COLOR_PAIRS - 1`.

`curses.pair_number(attr)`

Returns the number of the color-pair set by the attribute value *attr*. `color_pair()` is the counterpart to this function.

`curses.putp(string)`

Equivalent to `tputs(str, 1, putchar)`; emits the value of a specified terminfo capability for the current terminal. Note that the output of `putp` always goes to standard output.

`curses.qiflush(flag)`

If *flag* is false, the effect is the same as calling `noqiflush()`. If *flag* is true, or no argument is provided, the queues will be flushed when these control characters are read.

`curses.raw()`

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

`curses.reset_prog_mode()`

Restores the terminal to “program” mode, as previously saved by `def_prog_mode()`.

`curses.reset_shell_mode()`

Restores the terminal to “shell” mode, as previously saved by `def_shell_mode()`.

`curses.setsyx(y, x)`

Sets the virtual screen cursor to *y*, *x*. If *y* and *x* are both -1, then `leaveok` is set.

`curses.setupterm(termstr, fd)`

Initializes the terminal. *termstr* is a string giving the terminal name; if omitted, the value of the TERM environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied, the file descriptor for `sys.stdout` will be used.

`curses.start_color()`

Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`.

`start_color()` initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the `curses` module, `COLORS` and `COLOR_PAIRS`, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

`curses.termattrs()`

Returns a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

`curses.termname()`

Returns the value of the environment variable TERM, truncated to 14 characters.

`curses.tigetflag(capname)`

Returns the value of the Boolean capability corresponding to the terminfo capability name *capname*. The value -1 is returned if *capname* is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

`curses.tigetnum(capname)`

Returns the value of the numeric capability corresponding to the terminfo capability name *capname*. The value -2 is returned if *capname* is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

`curses.tigetstr(capname)`

Returns the value of the string capability corresponding to the terminfo capability name *capname*. `None` is returned if *capname* is not a string capability, or is canceled or absent from the terminal description.

`curses.tparm(str, ...)`

Instantiates the string *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `\033[6;4H`, the exact result depending on terminal type.

`curses.typeahead(fd)`

Specifies that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until `refresh` or `doupdate` is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

`curses.unctrl(ch)`

Returns a string which is a printable representation of the character *ch*. Control characters are displayed as a caret followed by the character, for example as `^C`. Printing characters are left as they are.

`curses.ungetch(ch)`

Push *ch* so the next `getch()` will return it.

Note

Only one *ch* can be pushed before `getch()` is called.

`curses.ungetmouse(id, x, y, z, bstate)`

Push a `KEY_MOUSE` event onto the input queue, associating the given state data with it.

`curses.use_env(flag)`

If used, this function should be called before `initscr()` or `newterm` are called. When *flag* is false, the values of lines and columns specified in the terminfo database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if `LINES` and `COLUMNS` are not set).

`curses.use_default_colors()`

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number `-1`. After calling this function, `init_pair(x, curses.COLOR_RED, -1)` initializes, for instance, color pair *x* to a red foreground color on the default background.

## 16.8.2. Window Objects

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods:

`window.addch(y, x, ch, attr)`

Note

A *character* means a C character (an ASCII code), rather than a Python character (a string of length 1). (This note is true whenever the documentation mentions a character.) The built-in `ord()` is handy for conveying strings to codes.

Paint character *ch* at (*y*, *x*) with attributes *attr*, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

`window.addnstr(y, x, str, n, attr)`

Paint at most *n* characters of the string *str* at (*y*, *x*) with attributes *attr*, overwriting anything previously on the display.

`window.addstr(y, x, str, attr)`

Paint the string *str* at (*y*, *x*) with attributes *attr*, overwriting anything previously on the display.

`window.attroff(attr)`

Remove attribute *attr* from the "background" set applied to all writes to the current window.

`window.attron(attr)`

Add attribute *attr* from the "background" set applied to all writes to the current window.

`window.attrset(attr)`

Set the "background" set of attributes to *attr*. This set is initially 0 (no attributes).

`window.bkgd(ch, attr)`

Sets the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

`window.bkgdset(ch, attr)`

Sets the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

`window.border([ls[, rs[, ts[, bs[, tl[, tr[, bl[, br]]]]]]])`

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details. The characters can be specified as integers or as one-character strings.

#### Note

A 0 value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

Parameter	Description	Default value
<code>ls</code>	Left side	<code>ACS_VLINE</code>
<code>rs</code>	Right side	<code>ACS_VLINE</code>
<code>ts</code>	Top	<code>ACS_HLINE</code>
<code>bs</code>	Bottom	<code>ACS_HLINE</code>
<code>tl</code>	Upper-left corner	<code>ACS_ULCORNER</code>
<code>tr</code>	Upper-right corner	<code>ACS_URCORNER</code>
<code>bl</code>	Bottom-left corner	<code>ACS_LLCORNER</code>
<code>br</code>	Bottom-right corner	<code>ACS_LRCORNER</code>

`window.box([vertch, horch])`

Similar to [border\(\)](#), but both `ls` and `rs` are `vertch` and both `ts` and `bs` are `horch`. The default corner characters are always used by this function.

`window.chgat([y, x][, num], attr)`

Sets the attributes of `num` characters at the current cursor position, or at position (`y`, `x`) if supplied. If no value of `num` is given or `num = -1`, the attribute will be set on all the characters to the end of the line. This function does not move the cursor. The changed line will be touched using the [touchline\(\)](#) method so that the contents will be redisplayed by the next window refresh.

`window.clear()`

Like [erase\(\)](#), but also causes the whole window to be repainted upon next call to [refresh\(\)](#).

`window.clearok(yes)`

If `yes` is 1, the next call to [refresh\(\)](#) will clear the window completely.

`window.clrtoeol()`

Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of [clrtoeol\(\)](#) is performed.

`window.clrtoeol()`

Erase from cursor to the end of the line.

`window.cursyncup()`

Updates the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

`window.delch([y, x])`

Delete any character at (`y`, `x`).

`window.deleteln()`

Delete the line under the cursor. All following lines are moved up by 1 line.

`window.derwin([nlines, ncols], begin_y, begin_x)`

An abbreviation for "derive window", [derwin\(\)](#) is the same as calling [subwin\(\)](#), except that `begin_y` and `begin_x` are relative to the origin of the window, rather than relative to the entire screen. Returns a window object for the derived window.

`window.echochar(ch[, attr])`

Add character `ch` with attribute `attr`, and immediately call [refresh\(\)](#) on the window.

`window.enclose(y, x)`

Tests whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning true or false. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

`window.erase()`

Clear the window.

`window.getbegyx()`

Return a tuple (`y`, `x`) of co-ordinates of upper-left corner.

`window.getch([y, x])`

Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on return numbers higher than 256. In no-delay mode, -1 is returned if there is no input, else [getch\(\)](#) waits until a key is pressed.

`window.getkey([y, x])`

Get a character, returning a string instead of an integer, as [getch\(\)](#) does. Function keys, keypad keys and so on return a multibyte string containing the key name. In no-delay mode, an exception is raised if there is no input.

`window.getmaxyx()`

Return a tuple (`y`, `x`) of the height and width of the window.

`window.getparyx()`

Returns the beginning coordinates of this window relative to its parent window into two integer variables `y` and `x`. Returns -1, -1 if this window has no parent.

`window.getstr([y, x])`

Read a string from the user, with primitive line editing capacity.

`window.getyx()`

Return a tuple (*y*, *x*) of current cursor position relative to the window's upper-left corner.

`window.hline([y, x], ch, n)`

Display a horizontal line starting at (*y*, *x*) with length *n* consisting of the character *ch*.

`window.idcok(flag)`

If *flag* is false, `curses` no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is true, use of character insertion and deletion is enabled. When `curses` is first initialized, use of character insert/delete is enabled by default.

`window.idlok(yes)`

If called with *yes* equal to 1, `curses` will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

`window.immedok(flag)`

If *flag* is true, any change in the window image automatically causes the window to be refreshed; you no longer have to call `refresh()` yourself. However, it may degrade performance considerably, due to repeated calls to `wrefresh`. This option is disabled by default.

`window.inch([y, x])`

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

`window.insch([y, x], ch[, attr])`

Paint character *ch* at (*y*, *x*) with attributes *attr*, moving the line from position *x* right by one character.

`window.insdelln(nlines)`

Inserts *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

`window.insertln()`

Insert a blank line under the cursor. All following lines are moved down by 1 line.

`window.insnstr([y, x], str, n[, attr])`

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

`window.insstr([y, x], str[, attr])`

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

`window.instr([y, x], n)`

Returns a string of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, `instr()` returns return a string at most *n* characters long (exclusive of the trailing NUL).

`window.is_linetouched(line)`

Returns true if the specified line was modified since the last call to `refresh()`; otherwise returns false. Raises a `curses.error` exception if *line* is not valid for the given window.

`window.is_wintouched()`

Returns true if the specified window was modified since the last call to `refresh()`; otherwise returns false.

`window.keypad(yes)`

If *yes* is 1, escape sequences generated by some keys (keypad, function keys) will be interpreted by `curses`. If *yes* is 0, escape sequences will be left as is in the input stream.

`window.leaveok(yes)`

If *yes* is 1, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *yes* is 0, cursor will always be at "cursor position" after an update.

`window.move(new_y, new_x)`

Move cursor to (*new\_y*, *new\_x*).

`window.mvderwin(y, x)`

Moves the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

`window.mvwin(new_y, new_x)`

Move the window so its upper-left corner is at (*new\_y*, *new\_x*).

`window.nodelay(yes)`

If *yes* is 1, `getch()` will be non-blocking.

`window.notimeout(yes)`

If *yes* is 1, escape sequences will not be timed out.

If *yes* is 0, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

`window.noutrefresh()`

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call [doupdate\(\)](#).

```
window.overlay(destwin, smirow, smicol, dmirow, dmincol, dmaxrow, dmaxcol)
```

Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of [overlay\(\)](#) can be used. *smirow* and *smicol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

```
window.overwrite(destwin, smirow, smicol, dmirow, dmincol, dmaxrow, dmaxcol)
```

Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of [overwrite\(\)](#) can be used. *smirow* and *smicol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

```
window.putwin(file)
```

Writes all data associated with the window into the provided file object. This information can be later retrieved using the [getwin\(\)](#) function.

```
window.redrawln(beg, num)
```

Indicates that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next [refresh\(\)](#) call.

```
window.redrawwin()
```

Touches the entire window, causing it to be completely redrawn on the next [refresh\(\)](#) call.

```
window.refresh(pminrow, pmincol, smirow, smicol, smaxrow, smaxcol)
```

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with [newpad\(\)](#). The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *smirow*, *smicol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *smirow*, or *smicol* are treated as if they were zero.

```
window.scroll([lines=1])
```

Scroll the screen or scrolling region upward by *lines* lines.

```
window.scrollok(flag)
```

Controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is false, the cursor is left on the bottom line. If *flag* is true, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call [idlok\(\)](#).

```
window.setscrreg(top, bottom)
```

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

```
window.standend()
```

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

```
window.standout()
```

Turn on attribute `A_STANDOUT`.

```
window.subpad(nlines, ncols, begin_y, begin_x)
```

Return a sub-window, whose upper-left corner is at (*begin\_y*, *begin\_x*), and whose width/height is *ncols/nlines*.

```
window.subwin(nlines, ncols, begin_y, begin_x)
```

Return a sub-window, whose upper-left corner is at (*begin\_y*, *begin\_x*), and whose width/height is *ncols/nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

```
window.syncdown()
```

Touches each location in the window that has been touched in any of its ancestor windows. This routine is called by [refresh\(\)](#), so it should almost never be necessary to call it manually.

```
window.syncok(flag)
```

If called with *flag* set to true, then [syncup\(\)](#) is called automatically whenever there is a change in the window.

```
window.syncup()
```

Touches all locations in ancestors of the window that have been changed in the window.

```
window.timeout(delay)
```

Sets blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and -1 will be returned by [getch\(\)](#) if no input is waiting. If *delay* is positive, then [getch\(\)](#) will block for *delay* milliseconds, and return -1 if there is still no input at the end of that time.

```
window.touchline(start, count, changed)
```

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed=1*) or unchanged (*changed=0*).

`window.touchwin()`

Pretend the whole window has been changed, for purposes of drawing optimizations.

`window.untouchwin()`

Marks all lines in the window as unchanged since the last call to [refresh\(\)](#).

`window.vline([y, x], ch, n)`

Display a vertical line starting at (*y*, *x*) with length *n* consisting of the character *ch*.

### 16.8.3. Constants

The `curses` module defines the following data members:

`curses.ERR`

Some `curses` routines that return an integer, such as `getch()`, return `ERR` upon failure.

`curses.OK`

Some `curses` routines that return an integer, such as `napms()`, return `OK` upon success.

`curses.version`

A string representing the current version of the module. Also available as `__version__`.

Several constants are available to specify character cell attributes:

Attribute	Meaning
<code>A_ALTCHARSET</code>	Alternate character set mode.
<code>A_BLINK</code>	Blink mode.
<code>A_BOLD</code>	Bold mode.
<code>A_DIM</code>	Dim mode.
<code>A_NORMAL</code>	Normal attribute.
<code>A_STANDOUT</code>	Standout mode.
<code>A_UNDERLINE</code>	Underline mode.

Keys are referred to by integer constants with names starting with `KEY_`. The exact keycaps available are system dependent.

Key constant	Key
<code>KEY_MIN</code>	Minimum key value
<code>KEY_BREAK</code>	Break key (unreliable)
<code>KEY_DOWN</code>	Down-arrow
<code>KEY_UP</code>	Up-arrow
<code>KEY_LEFT</code>	Left-arrow
<code>KEY_RIGHT</code>	Right-arrow
<code>KEY_HOME</code>	Home key (upward+left arrow)
<code>KEY_BACKSPACE</code>	Backspace (unreliable)
<code>KEY_F0</code>	Function keys. Up to 64 function keys are supported.
<code>KEY_Fn</code>	Value of function key <i>n</i>
<code>KEY_DL</code>	Delete line
<code>KEY_IL</code>	Insert line
<code>KEY_DC</code>	Delete character
<code>KEY_IC</code>	insert char or enter insert mode
<code>KEY_ETC</code>	Exit insert char mode
<code>KEY_CLEAR</code>	Clear screen
<code>KEY_EOS</code>	Clear to end of screen
<code>KEY_EOL</code>	Clear to end of line
<code>KEY_SF</code>	Scroll 1 line forward
<code>KEY_SR</code>	Scroll 1 line backward (reverse)
<code>KEY_NPAGE</code>	Next page
<code>KEY_PPAGE</code>	Previous page
<code>KEY_STAB</code>	Set tab
<code>KEY_CTAB</code>	Clear tab
<code>KEY_CATAB</code>	Clear all tabs
<code>KEY_ENTER</code>	Enter or send (unreliable)
<code>KEY_SRESET</code>	Soft (partial) reset (unreliable)
<code>KEY_RESET</code>	Reset or hard reset (unreliable)
<code>KEY_PRINT</code>	Print
<code>KEY_LL</code>	Home down or bottom (lower left)
<code>KEY_A1</code>	Upper left of keypad
<code>KEY_A3</code>	Upper right of keypad
<code>KEY_B2</code>	Center of keypad
<code>KEY_C1</code>	Lower left of keypad

KEY_C3	Lower right of keypad
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	Cancel
KEY_CLOSE	Close
KEY_COMMAND	Cmd (command)
KEY_COPY	Copy
KEY_CREATE	Create
KEY_END	End
KEY_EXIT	Exit
KEY_FIND	Find
KEY_HELP	Help
KEY_MARK	Mark
KEY_MESSAGE	Message
KEY_MOVE	Move
KEY_NEXT	Next
KEY_OPEN	Open
KEY_OPTIONS	Options
KEY_PREVIOUS	Prev (previous)
KEY_REDO	Redo
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	Refresh
KEY_REPLACE	Replace
KEY_RESTART	Restart
KEY_RESUME	Resume
KEY_SAVE	Save
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Dxit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRESUME	Shifted Resume
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend
KEY_UNDO	Undo
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY\_F1, KEY\_F2, KEY\_F3, KEY\_F4) available, and the arrow keys mapped to KEY\_UP, KEY\_DOWN, KEY\_LEFT and KEY\_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keypcap	Constant
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END

Page Up	KEY_NPAGE
Page Down	KEY_PPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

#### Note

These are available only after `initscr()` has been called.

ACS code	Meaning
ACS_BBSS	alternate name for upper right corner
ACS_BLOCK	solid square block
ACS_BOARD	board of squares
ACS_BSBS	alternate name for horizontal line
ACS_BSSB	alternate name for upper left corner
ACS_BSSS	alternate name for top tee
ACS_BTEE	bottom tee
ACS_BULLET	bullet
ACS_CKBOARD	checker board (stipple)
ACS_DARROW	arrow pointing down
ACS_DEGREE	degree symbol
ACS_DIAMOND	diamond
ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	left arrow
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal sign
ACS_PI	letter pi
ACS_PLMINUS	plus-or-minus sign
ACS_PLUS	big plus sign
ACS_RARROW	right arrow
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	alternate name for lower right corner
ACS_SBSB	alternate name for vertical line
ACS_SBSS	alternate name for right tee
ACS_SSBB	alternate name for lower left corner
ACS_SSBS	alternate name for bottom tee
ACS_SSSB	alternate name for left tee
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	pound sterling
ACS_TTEE	top tee
ACS_UARROW	up arrow
ACS_ULCORNER	upper left corner
ACS_URCORNER	upper right corner
ACS_VLINE	vertical line

The following table lists the predefined colors:

Constant	Color
COLOR_BLACK	Black
COLOR_BLUE	Blue
COLOR_CYAN	Cyan (light greenish blue)
COLOR_GREEN	Green
COLOR_MAGENTA	Magenta (purplish red)
COLOR_RED	Red
COLOR_WHITE	White
COLOR_YELLOW	Yellow

## 16.9. `curses.textpad` — Text input widget for curses programs

New in version 1.6.

The `curses.textpad` module provides a [Textbox](#) class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BEdit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function:

```
curses.textpad.rectangle(win, uly, ulx, lry, lrx)
```

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

### 16.9.1. Textbox objects

You can instantiate a [Textbox](#) object as follows:

```
class curses.textpad.Textbox(win)
```

Return a textbox widget object. The `win` argument should be a curses `WindowObject` in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates (0, 0). The instance's [stripspaces](#) flag is initially on.

[Textbox](#) objects have the following methods:

```
edit([validator])
```

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If `validator` is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the [stripspaces](#) member.

```
do_command(ch)
```

Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
Control-A	Go to left edge of window.
Control-B	Cursor left, wrapping to previous line if appropriate.
Control-D	Delete character under cursor.
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	Cursor right, wrapping to next line when appropriate.
Control-G	Terminate, returning the window contents.
Control-H	Delete character backward.
Control-J	Terminate if the window is 1 line, otherwise insert newline.
Control-K	If line is blank, delete it, otherwise clear to end of line.
Control-L	Refresh screen.
Control-N	Cursor down; move down one line.
Control-O	Insert a blank line at cursor location.
Control-P	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constant	Keystroke
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

```
gather()
```

This method returns the window contents as a string; whether blanks in the window are included is affected by the [stripspaces](#) member.

```
stripspaces
```

This data member is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

## 16.10. curses.wrapper — Terminal handler for curses programs

New in version 1.6.

This module supplies one function, [wrapper\(\)](#), which runs another function which should be the rest of your curses-using application. If the application raises an exception, [wrapper\(\)](#) will restore the terminal to a sane state before re-raising the exception and generating a traceback.

`curses.wrapper.wrapper(func, ...)`

Wrapper function that initializes curses and calls another function, *func*, restoring normal keyboard/screen behavior on error. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to `wrapper()`.

Before calling the hook function, `wrapper()` turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

## [Table Of Contents](#)

[16.8. curses — Terminal handling for character-cell displays](#)

- [16.8.1. Functions](#)
- [16.8.2. Window Objects](#)
- [16.8.3. Constants](#)

[16.9. curses.textpad — Text input widget for curses programs](#)

- [16.9.1. Textbox objects](#)
- [16.10. curses.wrapper — Terminal handler for curses programs](#)

### Previous topic

[16.7. getpass — Portable password input](#)

### Next topic

[16.11. curses.ascii — Utilities for ASCII characters](#)

### This Page

- [Show Source](#)

### Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [16. Generic Operating System Services](#) »

© [Copyright](#) 1990-2010, Python Software Foundation.

The Python Software Foundation is a non-profit corporation. [Please donate.](#)

Last updated on Feb 26, 2010. Created using [Sphinx](#) 0.6.3.