
Curses Programming with Python

Release 2.7.12

**Guido van Rossum
and the Python development team**

June 25, 2016

**Python Software Foundation
Email: docs@python.org**

Contents

1	What is curses?	1
1.1	The Python curses module	2
2	Starting and ending a curses application	2
3	Windows and Pads	3
4	Displaying Text	4
4.1	Attributes and Color	5
5	User Input	6
6	For More Information	7

Author A.M. Kuchling, Eric S. Raymond

Release 2.03

Abstract

This document describes how to write text-mode programs with Python 2.x, using the `curses` extension module to control the display.

What is curses?

The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals; such terminals include VT100s, the Linux console, and the simulated terminal provided by X11 programs such

as `xterm` and `rxvt`. Display terminals support various control codes to perform common operations such as moving the cursor, scrolling the screen, and erasing areas. Different terminals use widely differing codes, and often have their own minor quirks.

In a world of X displays, one might ask “why bother”? It’s true that character-cell display terminals are an obsolete technology, but there are niches in which being able to do fancy things with them are still valuable. One is on small-footprint or embedded Unixes that don’t carry an X server. Another is for tools like OS installers and kernel configurators that may have to run before X is available.

The `curses` library hides all the details of different terminals, and provides the programmer with an abstraction of a display, containing multiple non-overlapping windows. The contents of a window can be changed in various ways—adding text, erasing it, changing its appearance—and the `curses` library will automagically figure out what control codes need to be sent to the terminal to produce the right output.

The `curses` library was originally written for BSD Unix; the later System V versions of Unix from AT&T added many enhancements and new functions. BSD `curses` is no longer maintained, having been replaced by `ncurses`, which is an open-source implementation of the AT&T interface. If you’re using an open-source Unix such as Linux or FreeBSD, your system almost certainly uses `ncurses`. Since most current commercial Unix versions are based on System V code, all the functions described here will probably be available. The older versions of `curses` carried by some proprietary Unixes may not support everything, though.

No one has made a Windows port of the `curses` module. On a Windows platform, try the Console module written by Fredrik Lundh. The Console module provides cursor-addressable text output, plus full support for mouse and keyboard input, and is available from <http://effbot.org/zone/console-index.htm>.

The Python `curses` module

Thy Python module is a fairly simple wrapper over the C functions provided by `curses`; if you’re already familiar with `curses` programming in C, it’s really easy to transfer that knowledge to Python. The biggest difference is that the Python interface makes things simpler, by merging different C functions such as `addstr()`, `mvaddstr()`, `mvwaddstr()`, into a single `addstr()` method. You’ll see this covered in more detail later.

This HOWTO is simply an introduction to writing text-mode programs with `curses` and Python. It doesn’t attempt to be a complete guide to the `curses` API; for that, see the Python library guide’s section on `ncurses`, and the C manual pages for `ncurses`. It will, however, give you the basic ideas.

Starting and ending a `curses` application

Before doing anything, `curses` must be initialized. This is done by calling the `initscr()` function, which will determine the terminal type, send any required setup codes to the terminal, and create various internal data structures. If successful, `initscr()` returns a window object representing the entire screen; this is usually called `stdscr`, after the name of the corresponding C variable.

```
import curses
stdscr = curses.initscr()
```

Usually `curses` applications turn off automatic echoing of keys to the screen, in order to be able to read keys and only display them under certain circumstances. This requires calling the `noecho()` function.

```
curses.noecho()
```

Applications will also commonly need to react to keys instantly, without requiring the Enter key to be pressed; this is called `cbreak` mode, as opposed to the usual buffered input mode.

```
curses.cbreak()
```

Terminals usually return special keys, such as the cursor keys or navigation keys such as Page Up and Home, as a multibyte escape sequence. While you could write your application to expect such sequences and process them accordingly, `curses` can do it for you, returning a special value such as `curses.KEY_LEFT`. To get `curses` to do the job, you'll have to enable keypad mode.

```
stdscr.keypad(1)
```

Terminating a `curses` application is much easier than starting one. You'll need to call

```
curses.nocbreak(); stdscr.keypad(0); curses.echo()
```

to reverse the `curses`-friendly terminal settings. Then call the `endwin()` function to restore the terminal to its original operating mode.

```
curses.endwin()
```

A common problem when debugging a `curses` application is to get your terminal messed up when the application dies without restoring the terminal to its previous state. In Python this commonly happens when your code is buggy and raises an uncaught exception. Keys are no longer echoed to the screen when you type them, for example, which makes using the shell difficult.

In Python you can avoid these complications and make debugging much easier by importing the `curses.wrapper()` function. It takes a callable and does the initializations described above, also initializing colors if color support is present. It then runs your provided callable and finally deinitializes appropriately. The callable is called inside a try-catch clause which catches exceptions, performs `curses` deinitialization, and then passes the exception upwards. Thus, your terminal won't be left in a funny state on exception.

Windows and Pads

Windows are the basic abstraction in `curses`. A window object represents a rectangular area of the screen, and supports various methods to display text, erase it, allow the user to input strings, and so forth.

The `stdscr` object returned by the `initscr()` function is a window object that covers the entire screen. Many programs may need only this single window, but you might wish to divide the screen into smaller windows, in order to redraw or clear them separately. The `newwin()` function creates a new window of a given size, returning the new window object.

```
begin_x = 20; begin_y = 7
height = 5; width = 40
win = curses.newwin(height, width, begin_y, begin_x)
```

A word about the coordinate system used in `curses`: coordinates are always passed in the order `y,x`, and the top-left corner of a window is coordinate `(0,0)`. This breaks a common convention for handling coordinates, where the `x` coordinate usually comes first. This is an unfortunate difference from most other computer applications, but it's been part of `curses` since it was first written, and it's too late to change things now.

When you call a method to display or erase text, the effect doesn't immediately show up on the display. This is because `curses` was originally written with slow 300-baud terminal connections in mind; with these terminals, minimizing the time required to redraw the screen is very important. This lets `curses` accumulate changes to the screen, and display them in the most efficient manner. For example, if your program displays some characters in a window, and then clears the window, there's no need to send the original characters because they'd never be visible.

Accordingly, `curses` requires that you explicitly tell it to redraw windows, using the `refresh()` method of window objects. In practice, this doesn't really complicate programming with `curses` much. Most programs go into a flurry of activity, and then pause waiting for a keypress or some other action on the part of the user. All you have to do is to be sure that the screen has been redrawn before pausing to wait for user input, by simply calling `stdscr.refresh()` or the `refresh()` method of some other relevant window.

A pad is a special case of a window; it can be larger than the actual display screen, and only a portion of it displayed at a time. Creating a pad simply requires the pad's height and width, while refreshing a pad requires giving the coordinates of the on-screen area where a subsection of the pad will be displayed.

```
pad = curses.newpad(100, 100)
# These loops fill the pad with letters; this is
# explained in the next section
for y in range(0, 100):
    for x in range(0, 100):
        try:
            pad.addch(y,x, ord('a') + (x*x+y*y) % 26)
        except curses.error:
            pass

# Displays a section of the pad in the middle of the screen
pad.refresh(0,0, 5,5, 20,75)
```

The `refresh()` call displays a section of the pad in the rectangle extending from coordinate (5,5) to coordinate (20,75) on the screen; the upper left corner of the displayed section is coordinate (0,0) on the pad. Beyond that difference, pads are exactly like ordinary windows and support the same methods.

If you have multiple windows and pads on screen there is a more efficient way to go, which will prevent annoying screen flicker at refresh time. Use the `noutrefresh()` method of each window to update the data structure representing the desired state of the screen; then change the physical screen to match the desired state in one go with the function `doupdate()`. The normal `refresh()` method calls `doupdate()` as its last act.

Displaying Text

From a C programmer's point of view, curses may sometimes look like a twisty maze of functions, all subtly different. For example, `addstr()` displays a string at the current cursor location in the `stdscr` window, while `mvaddstr()` moves to a given `y,x` coordinate first before displaying the string. `waddstr()` is just like `addstr()`, but allows specifying a window to use, instead of using `stdscr` by default. `mvwaddstr()` follows similarly.

Fortunately the Python interface hides all these details; `stdscr` is a window object like any other, and methods like `addstr()` accept multiple argument forms. Usually there are four different forms.

Form	Description
<code>str</code> or <code>ch</code>	Display the string <code>str</code> or character <code>ch</code> at the current position
<code>str</code> or <code>ch</code> , <code>attr</code>	Display the string <code>str</code> or character <code>ch</code> , using attribute <code>attr</code> at the current position
<code>y, x</code> , <code>str</code> or <code>ch</code>	Move to position <code>y,x</code> within the window, and display <code>str</code> or <code>ch</code>
<code>y, x</code> , <code>str</code> or <code>ch</code> , <code>attr</code>	Move to position <code>y,x</code> within the window, and display <code>str</code> or <code>ch</code> , using attribute <code>attr</code>

Attributes allow displaying text in highlighted forms, such as in boldface, underline, reverse code, or in color. They'll be explained in more detail in the next subsection.

The `addstr()` function takes a Python string as the value to be displayed, while the `addch()` functions take a character, which can be either a Python string of length 1 or an integer. If it's a string, you're limited to displaying characters between 0 and 255. SVr4 curses provides constants for extension characters; these constants are integers greater than 255. For example, `ACS_PLMINUS` is a +/- symbol, and `ACS_ULCORNER` is the upper left corner of a box (handy for drawing borders).

Windows remember where the cursor was left after the last operation, so if you leave out the `y,x` coordinates, the string or character will be displayed wherever the last operation left off. You can also move the cursor with the `move(y, x)` method. Because some terminals always display a flashing cursor, you may want to ensure that the cursor is positioned in some location where it won't be distracting; it can be confusing to have the cursor blinking at some apparently random location.

If your application doesn't need a blinking cursor at all, you can call `curs_set(0)` to make it invisible. Equivalently, and for compatibility with older curses versions, there's a `leaveok(bool)` function. When *bool* is true, the curses library will attempt to suppress the flashing cursor, and you won't need to worry about leaving it in odd locations.

Attributes and Color

Characters can be displayed in different ways. Status lines in a text-based application are commonly shown in reverse video; a text viewer may need to highlight certain words. curses supports this by allowing you to specify an attribute for each cell on the screen.

An attribute is an integer, each bit representing a different attribute. You can try to display text with multiple attribute bits set, but curses doesn't guarantee that all the possible combinations are available, or that they're all visually distinct. That depends on the ability of the terminal being used, so it's safest to stick to the most commonly available attributes, listed here.

Attribute	Description
A_BLINK	Blinking text
A_BOLD	Extra bright or bold text
A_DIM	Half bright text
A_REVERSE	Reverse-video text
A_STANDOUT	The best highlighting mode available
A_UNDERLINE	Underlined text

So, to display a reverse-video status line on the top line of the screen, you could code:

```
stdscr.addstr(0, 0, "Current mode: Typing mode",
               curses.A_REVERSE)
stdscr.refresh()
```

The curses library also supports color on those terminals that provide it. The most common such terminal is probably the Linux console, followed by color xterms.

To use color, you must call the `start_color()` function soon after calling `initscr()`, to initialize the default color set (the `curses.wrapper.wrapper()` function does this automatically). Once that's done, the `has_colors()` function returns TRUE if the terminal in use can actually display color. (Note: curses uses the American spelling 'color', instead of the Canadian/British spelling 'colour'. If you're used to the British spelling, you'll have to resign yourself to misspelling it for the sake of these functions.)

The curses library maintains a finite number of color pairs, containing a foreground (or text) color and a background color. You can get the attribute value corresponding to a color pair with the `color_pair()` function; this can be bitwise-OR'ed with other attributes such as `A_REVERSE`, but again, such combinations are not guaranteed to work on all terminals.

An example, which displays a line of text using color pair 1:

```
stdscr.addstr("Pretty text", curses.color_pair(1))
stdscr.refresh()
```

As I said before, a color pair consists of a foreground and background color. `start_color()` initializes 8 basic colors when it activates color mode. They are: 0:black, 1:red, 2:green, 3:yellow, 4:blue, 5:magenta, 6:cyan, and 7:white. The curses module defines named constants for each of these colors: `curses.COLOR_BLACK`, `curses.COLOR_RED`, and so forth.

The `init_pair(n, f, b)` function changes the definition of color pair *n*, to foreground color *f* and background color *b*. Color pair 0 is hard-wired to white on black, and cannot be changed.

Let's put all this together. To change color 1 to red text on a white background, you would call:

```
curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
```

When you change a color pair, any text already displayed using that color pair will change to the new colors. You can also display new text in this color with:

```
stdscr.addstr(0,0, "RED ALERT!", curses.color_pair(1))
```

Very fancy terminals can change the definitions of the actual colors to a given RGB value. This lets you change color 1, which is usually red, to purple or blue or any other color you like. Unfortunately, the Linux console doesn't support this, so I'm unable to try it out, and can't provide any examples. You can check if your terminal can do this by calling `can_change_color()`, which returns `TRUE` if the capability is there. If you're lucky enough to have such a talented terminal, consult your system's man pages for more information.

User Input

The `curses` library itself offers only very simple input mechanisms. Python's support adds a text-input widget that makes up some of the lack.

The most common way to get input to a window is to use its `getch()` method. `getch()` pauses and waits for the user to hit a key, displaying it if `echo()` has been called earlier. You can optionally specify a coordinate to which the cursor should be moved before pausing.

It's possible to change this behavior with the method `nodelay()`. After `nodelay(1)`, `getch()` for the window becomes non-blocking and returns `curses.ERR` (a value of -1) when no input is ready. There's also a `halfdelay()` function, which can be used to (in effect) set a timer on each `getch()`; if no input becomes available within a specified delay (measured in tenths of a second), `curses` raises an exception.

The `getch()` method returns an integer; if it's between 0 and 255, it represents the ASCII code of the key pressed. Values greater than 255 are special keys such as Page Up, Home, or the cursor keys. You can compare the value returned to constants such as `curses.KEY_PPAGE`, `curses.KEY_HOME`, or `curses.KEY_LEFT`. Usually the main loop of your program will look something like this:

```
while 1:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break # Exit the while()
    elif c == curses.KEY_HOME:
        x = y = 0
```

The `curses.ascii` module supplies ASCII class membership functions that take either integer or 1-character-string arguments; these may be useful in writing more readable tests for your command interpreters. It also supplies conversion functions that take either integer or 1-character-string arguments and return the same type. For example, `curses.ascii.ctrl()` returns the control character corresponding to its argument.

There's also a method to retrieve an entire string, `getstr()`. It isn't used very often, because its functionality is quite limited; the only editing keys available are the backspace key and the Enter key, which terminates the string. It can optionally be limited to a fixed number of characters.

```
curses.echo() # Enable echoing of characters

# Get a 15-character string, with the cursor on the top line
s = stdscr.getstr(0,0, 15)
```

The Python `curses.textpad` module supplies something better. With it, you can turn a window into a text box that supports an Emacs-like set of keybindings. Various methods of `Textbox` class support editing with input validation and gathering the edit results either with or without trailing spaces. See the library documentation on `curses.textpad` for the details.

For More Information

This HOWTO didn't cover some advanced topics, such as screen-scraping or capturing mouse events from an xterm instance. But the Python library page for the curses modules is now pretty complete. You should browse it next.

If you're in doubt about the detailed behavior of any of the ncurses entry points, consult the manual pages for your curses implementation, whether it's ncurses or a proprietary Unix vendor's. The manual pages will document any quirks, and provide complete lists of all the functions, attributes, and ACS_* characters available to you.

Because the curses API is so large, some functions aren't supported in the Python interface, not because they're difficult to implement, but because no one has needed them yet. Feel free to add them and then submit a patch. Also, we don't yet have support for the menu library associated with ncurses; feel free to add that.

If you write an interesting little program, feel free to contribute it as another demo. We can always use more of them!

The ncurses FAQ: <http://invisible-island.net/ncurses/ncurses.faq.html>