

---

# 用 Python 进行 Curses 编程

發行 3.8.2rc1

Guido van Rossum  
and the Python development team

2 月 18, 2020

Python Software Foundation  
Email: docs@python.org

## Contents

<b>1</b>	<b>curses 是什么？</b>	<b>2</b>
1.1	Python 的 curses 模块 . . . . .	2
<b>2</b>	<b>开始和结束 curses 应用程序</b>	<b>2</b>
<b>3</b>	<b>Windows 和 Pad</b>	<b>4</b>
<b>4</b>	<b>显示文字</b>	<b>5</b>
4.1	属性和颜色 . . . . .	6
<b>5</b>	<b>用户输入</b>	<b>7</b>
<b>6</b>	<b>更多的信息</b>	<b>8</b>

---

作者 A.M. Kuchling, Eric S. Raymond

发布版本 2.04

### 摘要

本文档介绍了如何使用 curses 扩展模块控制文本模式的显示。

# 1 curses 是什么？

curses 库为基于文本的终端提供了独立于终端的屏幕绘制和键盘处理功能；这些终端包括 VT100，Linux 控制台以及各种程序提供的模拟终端。显示终端支持各种控制代码以执行常见的操作，例如移动光标，滚动屏幕和擦除区域。不同的终端使用相差很大的代码，并且往往有自己的小怪癖。

在普遍使用图形显示的世界中，人们可能会问“为什么自找麻烦”？毕竟字符单元显示终端确实是一种过时的技术，但是在某些领域中，能够用它们做花哨的事情仍然很有价值。一个小众市场是在不运行 X server 的小型或嵌入式 Unix 上。另一个是在提供图形支持之前，可能需要运行的工具，例如操作系统安装程序和内核配置程序。

curses 库提供了相当基础的功能，为程序员提供了包含多个非重叠文本窗口的显示的抽象。窗口的内容可以通过多种方式更改---添加文本，擦除文本，更改其外观---以及 curses 库将确定需要向终端发送哪些控制代码以产生正确的输出。curses 并没有提供诸多用户界面概念，例如按钮，复选框或对话框。如果需要这些功能，请考虑用户界面库，例如 [Urwid](#)。

curses 库最初是为 BSD Unix 编写的。后来 AT&T 的 Unix System V 版本加入了许多增强功能和新功能。如今 BSD curses 已不再维护，被 ncurses 取代，ncurses 是 AT&T 接口的开源实现。如果使用的是 Linux 或 FreeBSD 等开源 Unix 系统，则几乎肯定会使用 ncurses。由于大多数当前的商业 Unix 版本都基于 System V 代码，因此这里描述的所有功能可能都可用。但是，某些专有 Unix 所带来的较早版本的 curses 可能无法支持所有功能。

Windows 版本的 Python 不包含 curses 模块。提供了一个名为 [UniCurses](#) 的移植版本。也可以尝试使用 Fredrik Lundh 编写 [the Console module](#)，它使用了与 curses 不相同的 API，但提供了可光标定位的文本输出，完全支持鼠标和键盘输入。

## 1.1 Python 的 curses 模块

The 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()`, and `mvwaddstr()` into a single `addstr()` method. You'll see this covered in more detail later.

This HOWTO is 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.

## 2 开始和结束 curses 应用程序

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(True)
```

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

```
curses.nocbreak()
stdscr.keypad(False)
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 and using it like this:

```
from curses import wrapper

def main(stdscr):
    # Clear screen
    stdscr.clear()

    # This raises ZeroDivisionError when i == 10.
    for i in range(0, 11):
        v = i-10
        stdscr.addstr(i, 0, '10 divided by {} is {}'.format(v, 10/v))

    stdscr.refresh()
    stdscr.getkey()

wrapper(main)
```

The `wrapper()` function takes a callable object and does the initializations described above, also initializing colors if color support is present. `wrapper()` then runs your provided callable. Once the callable returns, `wrapper()` will restore the original state of the terminal. The callable is called inside a `try...except` that catches exceptions, restores the state of the terminal, and then re-raises the exception. Therefore your terminal won't be left in a funny state on exception and you'll be able to read the exception's message and traceback.

### 3 Windows 和 Pad

Windows are the basic abstraction in curses. A window object represents a rectangular area of the screen, and supports 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)
```

Note that the coordinate system used in curses is unusual. Coordinates are always passed in the order *y,x*, and the top-left corner of a window is coordinate (0,0). This breaks the normal convention for handling coordinates where the *x* coordinate 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.

Your application can determine the size of the screen by using the `curses.LINES` and `curses.COLS` variables to obtain the *y* and *x* sizes. Legal coordinates will then extend from (0,0) to (`curses.LINES - 1`, `curses.COLS - 1`).

When you call a method to display or erase text, the effect doesn't immediately show up on the display. Instead you must call the `refresh()` method of window objects to update the screen.

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 was very important. Instead curses accumulates changes to the screen and displays them in the most efficient manner when you call `refresh()`. For example, if your program displays some text in a window and then clears the window, there's no need to send the original text because they're never visible.

In practice, explicitly telling curses to redraw a window 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 first 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 the pad displayed at a time. Creating a pad 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; addch() is
# explained in the next section
for y in range(0, 99):
    for x in range(0, 99):
        pad.addch(y,x, ord('a') + (x*x+y*y) % 26)

# Displays a section of the pad in the middle of the screen.
# (0,0) : coordinate of upper-left corner of pad area to display.
# (5,5) : coordinate of upper-left corner of window area to be filled
#         with pad content.
# (20, 75) : coordinate of lower-right corner of window area to be
#           : filled with pad content.
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 update the screen and prevent annoying screen flicker as each part of the screen gets updated. `refresh()` actually does two things:

- 1) Calls the `noutrefresh()` method of each window to update an underlying data structure representing the desired state of the screen.
- 2) Calls the function `doupdate()` function to change the physical screen to match the desired state recorded in the data structure.

Instead you can call `noutrefresh()` on a number of windows to update the data structure, and then call `doupdate()` to update the screen.

## 4 显示文字

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()` allows specifying both a window and a coordinate.

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

形式	描述
<code>str</code> 或 <code>ch</code>	在当前位置显示字符串 <code>str</code> 或字符 <code>ch</code>
<code>str</code> 或 <code>ch</code> , <code>attr</code>	在当前位置使用 <code>attr</code> 属性显示字符串 <code>str</code> 或字符 <code>ch</code>
<code>y, x</code> , <code>str</code> 或 <code>ch</code>	移动到窗口内的 <code>y,x</code> 位置, 并显示 <code>str</code> 或 <code>ch</code>
<code>y, x</code> , <code>str</code> 或 <code>ch</code> , <code>attr</code>	移至窗口内的 <code>y,x</code> 位置, 并使用 <code>attr</code> 属性显示 <code>str</code> 或 <code>ch</code>

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

The `addstr()` method takes a Python string or bytestring as the value to be displayed. The contents of bytestrings are sent to the terminal as-is. Strings are encoded to bytes using the value of the window's `encoding` attribute; this defaults to the default system encoding as returned by `locale.getpreferredencoding()`.

The `addch()` methods take a character, which can be either a string of length 1, a bytestring of length 1, or an integer.

Constants are provided 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). You can also use the appropriate Unicode character.

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(False)` to make it invisible. For compatibility with older curses versions, there's a `leaveok(bool)` function that's a synonym for `curs_set()`. 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.

## 4.1 属性和颜色

Characters can be displayed in different ways. Status lines in a text-based application are commonly shown in reverse video, or 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.

属性	描述
A_BLINK	闪烁文字
A_BOLD	超亮或粗体文字
A_DIM	半明亮的文字
A_REVERSE	反向视频文本
A_STANDOUT	可用的最佳突出显示模式
A_UNDERLINE	带下划线的文字

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()` 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. 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.

Colors are numbered, and `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.

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.

## 5 用户输入

The C `curses` library offers only very simple input mechanisms. Python's `curses` module adds a basic text-input widget. (Other libraries such as `Urwid` have more extensive collections of widgets.)

There are two methods for getting input from a window:

- `getch()` refreshes the screen and then waits for the user to hit a key, displaying the key if `echo()` has been called earlier. You can optionally specify a coordinate to which the cursor should be moved before pausing.
- `getkey()` does the same thing but converts the integer to a string. Individual characters are returned as 1-character strings, and special keys such as function keys return longer strings containing a key name such as `KEY_UP` or `^G`.

It's possible to not wait for the user using the `nodelay()` window method. After `nodelay(True)`, `getch()` and `getkey()` for the window become non-blocking. To signal that no input is ready, `getch()` returns `curses.ERR` (a value of -1) and `getkey()` raises an exception. 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`. The main loop of your program may look something like this:

```
while True:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break # Exit the while loop
    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 such loops. 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 `curses.textpad` module supplies a text box that supports an Emacs-like set of keybindings. Various methods of the `Textbox` class support editing with input validation and gathering the edit results either with or without trailing spaces. Here's an example:

```
import curses
from curses.textpad import Textbox, rectangle

def main(stdscr):
    stdscr.addstr(0, 0, "Enter IM message: (hit Ctrl-G to send)")

    editwin = curses.newwin(5, 30, 2, 1)
    rectangle(stdscr, 1, 0, 1+5+1, 1+30+1)
    stdscr.refresh()

    box = Textbox(editwin)

    # Let the user edit until Ctrl-G is struck.
    box.edit()

    # Get resulting contents
    message = box.gather()
```

See the library documentation on `curses.textpad` for more details.

## 6 更多的信息

This HOWTO doesn't cover some advanced topics, such as reading the contents of the screen or capturing mouse events from an xterm instance, but the Python library page for the `curses` module is now reasonably complete. You should browse it next.

If you're in doubt about the detailed behavior of the `curses` functions, 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. Often this isn't because they're difficult to implement, but because no one has needed them yet. Also, Python doesn't yet support the menu library associated with `ncurses`. Patches adding support for these would be welcome; see [the Python Developer's Guide](#) to learn more about submitting patches to Python.

- [Writing Programs with NCURSES](#): a lengthy tutorial for C programmers.
- `ncurses` 手册主页 <<https://linux.die.net/man/3/ncurses>>\_
- `ncurses` 常见问题 <<http://invisible-island.net/ncurses/ncurses.faq.html>>\_
- "Use `curses`... don't swear": video of a PyCon 2013 talk on controlling terminals using `curses` or `Urwid`.
- "Console Applications with `Urwid`": video of a PyCon CA 2012 talk demonstrating some applications written using `Urwid`.