# Tutorial on Python Curses Programming

Norman Matloff  
University of California, Davis  
©2005-2007, N. Matloff  

April 5, 2007

## Contents

<table>
<thead>
<tr>
<th>1 Overview</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Function</td>
<td>2</td>
</tr>
<tr>
<td>1.2 History</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Relevance Today</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Examples of Python Curses Programs</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Useless Example</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Useful Example</td>
<td>5</td>
</tr>
<tr>
<td>2.3 A Few Other Short Examples</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 What Else Can Curses Do?</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Curses by Itself</td>
<td>7</td>
</tr>
</tbody>
</table>

| 4 Libraries Built on Top of Curses | 7 |

| 5 If Your Terminal Window Gets Messed Up | 7 |

| 6 Debugging | 8 |
1 Overview

1.1 Function

Many widely-used programs need to make use of a terminal’s cursor-movement capabilities. A familiar example is vi; most of its commands make use of such capabilities. For example, hitting the j key while in vi will make the cursor move down line. Typing dd will result in the current line being erased, the lines below it moving up one line each, and the lines above it remaining unchanged. There are similar issues in the programming of emacs, etc.

The curses library gives the programmer functions (APIs, Application Program Interfaces) to call to take such actions.

Since the operations available under curses are rather primitive—cursor movement, text insertion, etc.—libraries have been developed on top of curses to do more advanced operations such as pull-down menus, radio buttons and so on. More on this in the Python context later.

1.2 History

Historically, a problem with all this was that different terminals had different ways in which to specify a given type of cursor motion. For example, if a program needed to make the cursor move up one line on a VT100 terminal, the program would need to send the characters Escape, [, and A:

printf("%c%c%c",27,'[','A');

(the character code for the Escape key is 27). But for a Televideo 920C terminal, the program would have to send the ctrl-K character, which has code 11:

printf("%c",11);

Clearly, the authors of programs like vi would go crazy trying to write different versions for every terminal, and worse yet, anyone else writing a program which needed cursor movement would have to “re-invent the wheel,” i.e. do the same work that the vi-writers did, a big waste of time.

That is why the curses library was developed. The goal was to alleviate authors of cursor-oriented programs like vi of the need to write different code for different terminals. The programs would make calls to the API library, and the library would sort out what to do for the given terminal type.

The library would know which type of terminal you were using, via the environment variable TERM. The library would look up your terminal type in its terminal database (the file /etc/termcap). When you, the programmer, would call the curses API to, say, move the cursor up one line, the API would determine which character sequence was needed to make this happen.

For example, if your program wanted to clear the screen, it would not directly use any character sequences like those above. Instead, it would simply make the call

clear();

and curses would do the work on the program’s behalf.
1.3 Relevance Today

Many dazzling GUI programs are popular today. But although the GUI programs may provide more “eye candy,” they can take a long time to load into memory, and they occupy large amounts of territory on your screen. So, curses programs such as vi and emacs are still in wide usage.

Interestingly, even some of those classical curses programs have also become somewhat GUI-ish. For instance vim, the most popular version of vi (it’s the version which comes with most Linux distributions, for example), can be run in gvim mode. There, in addition to having the standard keyboard-based operations, one can also use the mouse. One can move the cursor to another location by clicking the mouse at that point; one can use the mouse to select blocks of text for deletion or movement; etc. There are icons at the top of the editing window, for operations like Find, Make, etc.

2 Examples of Python Curses Programs

2.1 Useless Example

The program below, crs.py, does not do anything useful. Its sole purpose is to introduce some of the curses APIs.

There are lots of comments in the code. Read them carefully, first by reading the introduction at the top of the file, and then going to the bottom of the file to read main(). After reading the latter, read the other functions.

```python
# crs.py; simple illustration of curses library, consisting of a very
# unexciting "game"; keeps drawing the user’s input characters into a
# box, filling one column at a time, from top to bottom, left to right,
# returning to top left square when reach bottom right square
#
# the bottom row of the box is displayed in another color
#
# usage: python crs.py boxsize
#
import curses, sys, traceback

class gb:
    boxrows = int(sys.argv[1]) # number of rows in the box
    boxcols = boxrows # number of columns in the box
    scrn = None # will point to window object
    row = None # current row position
    col = None # current column position

def draw(chr):
    # paint chr at current position, overwriting what was there; if it’s
    # the last row, also change colors; if instead of color we had
    # wanted, say, reverse video, we would specify curses.A_REVERSE instead of
    # curses.color_pair(1)
    if gb.row == gb.boxrows-1:
        gb.scrn.addch(gb.row,gb.col,chr,curses.color_pair(1))
    else:
        gb.scrn.addch(gb.row,gb.col,chr)
    # implement the change
    gb.scrn.refresh()
```
# move down one row
gb.row += 1
# if at bottom, go to top of next column
if gb.row == gb.boxrows:
    gb.row = 0
    gb.col += 1
    # if in last column, go back to first column
    if gb.col == gb.boxcols:
        gb.col = 0

# this code is vital; without this code, your terminal would be unusable
# after the program exits

def restorescreen():
    # restore "normal"--i.e. wait until hit Enter--keyboard mode
    curses.nocbreak()
    # restore keystroke echoing
    curses.echo()
    # required cleanup call
    curses.endwin()

def main():
    # first we must create a window object; it will fill the whole screen
    gb.scrn = curses.initscr()
    # turn off keystroke echo
    curses.noecho()
    # keystrokes are honored immediately, rather than waiting for the
    # user to hit Enter
    curses.cbreak()
    # start color display (if it exists; could check with has_colors())
    curses.start_color()
    # set up a foreground/background color pair (can do many)
    curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
    # clear screen
    gb.scrn.clear()
    # set current position to upper-left corner; note that these are our
    # own records of position, not Curses'
    gb.row = 0
    gb.col = 0
    # implement the actions done so far (just the clear())
    gb.scrn.refresh()
    # now play the "game"
    while True:
        # read character from keyboard
        c = gb.scrn.getch()
        # was returned as an integer (ASCII); make it a character
        c = chr(c)
        # quit?
        if c == 'q': break
        # draw the character
        draw(c)
        # restore original settings
        restorescreen()

if __name__ == '__main__':
    # in case of execution error, have a smooth recovery and clear
    # display of error message (nice example of Python exception
    # handling); it is recommended that you use this format for all of
    # your Python curses programs; you can automate all this (and more)
    # by using the built-in function curses.wrapper(), but we've shown
    # it done "by hand" here to illustrate the issues involved
    try:
        main()
    except:
        # print error message re exception
        traceback.print_exc()
2.2 Useful Example

The following program allows the user to continuously monitor processes on a Unix system. Although some more features could be added to make it more useful, it is a real working utility.

```python
import curses, os, sys, traceback

# global variables

class gb:
    scrn = None  # will point to Curses window object
    cmdoutlines = []  # output of 'ps ax' (including the lines we don’t use, for possible future extension)
    winrow = None  # current row position in screen
    startrow = None  # index of first row in cmdoutlines to be displayed

def runpsax():
    p = os.popen('ps ax','r')
    gb.cmdoutlines = []
    row = 0
    for ln in p:
        # don’t allow line wraparound, so truncate long lines
        ln = ln[:curses.COLS]
        # remove EOLN if it is still there
        if ln[-1] == '
': ln = ln[:-1]
        gb.cmdoutlines.append(ln)
p.close()

def showlastpart():
    # display last part of command output (as much as fits in screen)
    gb.scrn.clear()
    gb.winrow = 0
    ncmdlines = len(gb.cmdoutlines)
    if ncmdlines <= curses.LINES:
        gb.startrow = 0
        nwinlines = ncmdlines
    else:
        gb.startrow = ncmdlines - curses.LINES - 1
        nwinlines = curses.LINES
    lastrow = gb.startrow + nwinlines - 1
    for ln in gb.cmdoutlines[gb.startrow:lastrow]:
        gb.scrn.addstr(gb.winrow,0,ln)
```

```bash
# psax.py; illustration of curses library
# runs the shell command 'ps ax' and saves the last lines of its output,
# as many as the window will fit; allows the user to move up and down
# within the window, killing those processes
# run line: python psax.py

# user commands:
# 'u': move highlight up a line
# 'd': move highlight down a line
# 'k': kill process in currently highlighted line
# 'r': re-run 'ps ax' for update
# 'q': quit

# possible extensions: allowing scrolling, so that the user could go
# through all the 'ps ax' output; allow wrap-around for long lines; ask
# user to confirm before killing a process
import curses, os, sys, traceback

class gb:
    scrn = None  # will point to Curses window object
    cmdoutlines = []  # output of 'ps ax' (including the lines we don’t
    winrow = None  # use, for possible future extension)
    startrow = None  # current row position in screen
    startrow = None  # index of first row in cmdoutlines to be displayed

def runpsax():
    p = os.popen('ps ax','r')
    gb.cmdoutlines = []
    row = 0
    for ln in p:
        # don’t allow line wraparound, so truncate long lines
        ln = ln[:curses.COLS]
        # remove EOLN if it is still there
        if ln[-1] == '
': ln = ln[:-1]
        gb.cmdoutlines.append(ln)
p.close()

def showlastpart():
    # display last part of command output (as much as fits in screen)
    gb.scrn.clear()
    gb.winrow = 0
    ncmdlines = len(gb.cmdoutlines)
    if ncmdlines <= curses.LINES:
        gb.startrow = 0
        nwinlines = ncmdlines
    else:
        gb.startrow = ncmdlines - curses.LINES - 1
        nwinlines = curses.LINES
    lastrow = gb.startrow + nwinlines - 1
    for ln in gb.cmdoutlines[gb.startrow:lastrow]:
        gb.scrn.addstr(gb.winrow,0,ln)
```
def main():
    # window setup
    gb.scrn = curses.initscr()
    curses.noecho()
    curses.cbreak()
    # rpdb.set_trace() (I used RPDB for debugging)
    # run 'ps ax' and process the output
    gb.psax = runpsax()
    # display in the window
    showlastpart()
    # user command loop
    while True:
        # get user command
        c = gb.scrn.getch()
        c = chr(c)
        if c == 'u': updown(-1)
        elif c == 'd': updown(1)
        elif c == 'r': rerun()
        elif c == 'k': kill()
        else: break
    restorescreen()

def restorescreen():
    curses.nocbreak()
    curses.echo()
    curses.endwin()

if __name__ == '__main__':
    try:
        main()
    except:
        # print error message re exception
        traceback.print_exc()
2.3 A Few Other Short Examples

See the directory Demo/curses in the Python source code distribution

3 What Else Can Curses Do?

3.1 Curses by Itself

The examples above just barely scratch the surface. We won’t show further examples here, but to illustrate other operations, think about what vi, a curses-based program, must do in response to various user commands, such as the following (suppose our window object is scrn):

- k command, to move the cursor up one line: might call scrn.mov(r,c), which moves the curses cursor to the specified row and column\(^1\)
- dd command, to delete a line: might call scrn.deleteln(), which causes the current row to be deleted and makes the rows below move up\(^2\)
- ~ command, to change case of the character currently under the cursor: might call scrn.inch(), which returns the character currently under the cursor, and then call scrn.addch() to put in the character of opposite case
- :sp command (vim), to split the current vi window into two subwindows: might call curses.newwin()
- mouse operations in gvim: call functions such as curses.mousemask(), curses.getmouse(), etc.

You can imagine similar calls in the source code for emacs, etc.

4 Libraries Built on Top of Curses

The operations provided by curses are rather primitive. Say for example you wish to have a menu sub-window in your application. You could do this directly with curses, using its primitive operations, but it would be nice to have high-level libraries for this.

A number of such libraries have been developed. One you may wish to consider is urwid, http://excess.org/urwid/.

5 If Your Terminal Window Gets Messed Up

Curses programs by nature disable the “normal” behavior you expect of a terminal window. If your program has a bug that makes it exit prematurely, that behavior will not automatically be re-enabled.

\(^1\)But if the movement causes a scrolling operation, other curses functions will need to be called too.
\(^2\)But again, things would be more complicated if that caused scrolling.
In our first example above, you saw how we could include to do the re-enabling even if the program crashes. This of course is what is recommended. But if you don’t do it, you can re-enable your window capabilities by hitting ctrl-j then typing “reset”, then hitting ctrl-j again.

### 6 Debugging

The open source debugging tools I usually use for Python—PDB, DDD—but neither can be used for debugging Python `curses` application. For the PDB, the problem is that one’s PDB commands and their outputs are on the same screen as the application program’s display, a hopeless mess. This ought not be a problem in using DDD as an interface to PDB, since DDD does allow one to have a separate execution window. That works fine for `curses` programming in C/C++, but for some reason this can’t be invoked for Python. Even the Eclipse IDE seems to have a problem in this regard.

However, a very usable tool is RPDB, [http://RPDBdb.digitalpeers.com/](http://RPDBdb.digitalpeers.com/) a very usable program which was adapted from PDB. RPDB does indeed set up a separate execution window, which solves the problem. I’ve got a quick introduction to RPDB at [http://heather.cs.ucdavis.edu/~matloff/rpdb.html](http://heather.cs.ucdavis.edu/~matloff/rpdb.html)