Tutorial on Python Curses Programming

Norman Matloff
University of California, Davis

April 5, 2007

Contents

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.

```
1  # crs.py; simple illustration of curses library, consisting of a very
2  # unexciting "game"; keeps drawing the user's input characters into a
3  # box, filling one column at a time, from top to bottom, left to right,
4  # returning to top left square when reach bottom right square
5
6  # the bottom row of the box is displayed in another color
7
8  # usage:  python crs.py boxsize
9
10 # how to play the "game":  keep typing characters, until wish to stop,
11 # which you do by hitting the q key
12
13 import curses, sys, traceback
14
15 # global variables
16 class gb:
17    boxrows = int(sys.argv[1])  # number of rows in the box
18    boxcols = boxrows  # number of columns in the box
19    scrn = None  # will point to window object
20    row = None  # current row position
21    col = None  # current column position
22
23 def draw(chr):
24    # paint chr at current position, overwriting what was there; if it's
25    # the last row, also change colors; if instead of color we had
26    # wanted, say, reverse video, we would specify curses.A_REVERSE instead of
27    # curses.color_pair(1)
28    if gb.row == gb.boxrows-1:
29       gb.scrn.addch(gb.row,gb.col,chr,curses.color_pair(1))
30    else:
31       gb.scrn.addch(gb.row,gb.col,chr)
32    # implement the change
33    gb.scrn.refresh()
```

```
34      # move down one row
35      gb.row += 1
36      # if at bottom, go to top of next column
37      if gb.row == gb.boxrows:
38         gb.row = 0
39         gb.col += 1
40         # if in last column, go back to first column
41         if gb.col == gb.boxcols: gb.col = 0
42
43   # this code is vital; without this code, your terminal would be unusable
44   # after the program exits
45   def restorescreen():
46      # restore "normal"--i.e. wait until hit Enter--keyboard mode
47      curses.nocbreak()
48      # restore keystroke echoing
49      curses.echo()
50      # required cleanup call
51      curses.endwin()
52
53   def main():
54      # first we must create a window object; it will fill the whole screen
55      gb.scrn = curses.initscr()
56      # turn off keystroke echo
57      curses.noecho()
58      # keystrokes are honored immediately, rather than waiting for the
59      # user to hit Enter
60      curses.cbreak()
61      # start color display (if it exists; could check with has_colors())
62      curses.start_color()
63      # set up a foreground/background color pair (can do many)
64      curses.init_pair(1,curses.COLOR_RED,curses.COLOR_WHITE)
65      # clear screen
66      gb.scrn.clear()
67      # set current position to upper-left corner; note that these are our
68      # own records of position, not Curses'
69      gb.row = 0
70      gb.col = 0
71      # implement the actions done so far (just the clear())
72      gb.scrn.refresh()
73      # now play the "game"
74      while True:
75         # read character from keyboard
76         c = gb.scrn.getch()
77         # was returned as an integer (ASCII); make it a character
78         c = chr(c)
79         # quit?
80         if c == 'q': break
81         # draw the character
82         draw(c)
83      # restore original settings
84      restorescreen()
85
86   if __name__ =='__main__':
87      # in case of execution error, have a smooth recovery and clear
88      # display of error message (nice example of Python exception
89      # handling); it is recommended that you use this format for all of
90      # your Python curses programs; you can automate all this (and more)
91      # by using the built-in function curses.wrapper(), but we've shown
92      # it done "by hand" here to illustrate the issues involved
93      try:
94         main()
95      except:
96         restorescreen()
97         # print error message re exception
98         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.

```
1  # psax.py; illustration of curses library
2
3  # runs the shell command 'ps ax' and saves the last lines of its output,
4  # as many as the window will fit; allows the user to move up and down
5  # within the window, killing those processes
6
7  # run line:  python psax.py
8
9  # user commands:
10
11 #    'u':  move highlight up a line
12 #    'd':  move highlight down a line
13 #    'k':  kill process in currently highlighted line
14 #    'r':  re-run 'ps ax' for update
15 #    'q':  quit
16
17
18 # possible extensions:  allowing scrolling, so that the user could go
19 # through all the 'ps ax' output; allow wraparound for long lines; ask
20 # user to confirm before killing a process
21
22 import curses, os, sys, traceback
23
24 # global variables
25 class gb:
26    scrn = None  # will point to Curses window object
27    cmdoutlines = []  # output of 'ps ax' (including the lines we don't
28                      # use, for possible future extension)
29    winrow = None  # current row position in screen
30    startrow = None  # index of first row in cmdoutlines to be displayed
31
32 def runpsax():
33    p = os.popen('ps ax','r')
34    gb.cmdoutlines = []
35    row = 0
36    for ln in p:
37       # don't allow line wraparound, so truncate long lines
38       ln = ln[:curses.COLS]
39       # remove EOLN if it is still there
40       if ln[-1] == '\n': ln = ln[:-1]
41       gb.cmdoutlines.append(ln)
42    p.close()
43
44 # display last part of command output (as much as fits in screen)
45 def showlastpart():
46    # clear screen
47    gb.scrn.clear()
48    # prepare to paint the (last part of the) 'ps ax' output on the screen
49    gb.winrow = 0
50    ncmdlines = len(gb.cmdoutlines)
51    # two cases, depending on whether there is more output than screen rows
52    if ncmdlines <= curses.LINES:
53       gb.startrow = 0
54       nwinlines = ncmdlines
55    else:
56       gb.startrow = ncmdlines - curses.LINES - 1
57       nwinlines = curses.LINES
58    lastrow = gb.startrow + nwinlines - 1
59    # now paint the rows
60    for ln in gb.cmdoutlines[gb.startrow:lastrow]:
61       gb.scrn.addstr(gb.winrow,0,ln)
```

```
62          gb.winrow += 1
63       # last line highlighted
64       gb.scrn.addstr(gb.winrow,0,gb.cmdoutlines[lastrow],curses.A_BOLD)
65       gb.scrn.refresh()
66
67    # move highlight up/down one line
68    def updown(inc):
69       tmp = gb.winrow + inc
70       # ignore attempts to go off the edge of the screen
71       if tmp >= 0 and tmp < curses.LINES:
72          # unhighlight the current line by rewriting it in default attributes
73          gb.scrn.addstr(gb.winrow,0,gb.cmdoutlines[gb.startrow+gb.winrow])
74          # highlight the previous/next line
75          gb.winrow = tmp
76          ln = gb.cmdoutlines[gb.startrow+gb.winrow]
77          gb.scrn.addstr(gb.winrow,0,ln,curses.A_BOLD)
78          gb.scrn.refresh()
79
80    # kill the highlighted process
81    def kill():
82       ln = gb.cmdoutlines[gb.startrow+gb.winrow]
83       pid = int(ln.split()[0])
84       os.kill(pid,9)
85
86    # run/re-run 'ps ax'
87    def rerun():
88       runpsax()
89       showlastpart()
90
91    def main():
92       # window setup
93       gb.scrn = curses.initscr()
94       curses.noecho()
95       curses.cbreak()
96       # rpdb.set_trace() (I used RPDB for debugging)
97       # run 'ps ax' and process the output
98       gb.psax = runpsax()
99       # display in the window
100      showlastpart()
101      # user command loop
102      while True:
103         # get user command
104         c = gb.scrn.getch()
105         c = chr(c)
106         if c == 'u': updown(-1)
107         elif c == 'd': updown(1)
108         elif c == 'r': rerun()
109         elif c == 'k': kill()
110         else: break
111      restorescreen()
112
113   def restorescreen():
114      curses.nocbreak()
115      curses.echo()
116      curses.endwin()
117
118   if __name__ =='__main__':
119      try:
120         main()
121      except:
122         restorescreen()
123         # print error message re exception
124         traceback.print_exc()
```

Try running it yourself!

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]
- $\sim$ 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 subwindow 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. Butif 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 tools is RPDB, `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`