Charming Python: Curses programming
Tips for beginners

David Mertz, Ph.D.
President, Gnosis Software, Inc.
September 2000

A certain class of Python applications works best with an interactive user interface without the overhead or complexity of a graphical environment. For interactive text-mode programs (under Linux/UNIX), for example, the `ncurses` library, wrapped in Python's standard curses module, is just what you need. In this article, David Mertz discusses the use of curses in Python. He illustrates the curses environment using sample source code from a front-end to the Txt2Html program.

Contents:

Introducing curses
The application
Wrapping a curses application
Our main() event loop
Getting user input
Conclusion
Resources
About the author

The curses library (`ncurses`) provides a terminal-independent method of controlling character screens. curses is a standard part of most UNIX-like systems, including Linux, but has also been ported to Windows and other systems. Curses programs will run on text-only systems and within xterm's and other windowed console sessions, which helps make these applications very portable.

Introducing curses
Python's standard curses provides a basic interface to the common features of the "glass teletype," as the CRT was known in the 1970s when the original curses library was created. There are a number of ways to bring greater sophistication to interactive text-mode programs written in Python. These fall into two categories.

On one hand, there are Python modules that support the full feature set of `ncurses` (a superset of curses) or `slang` (a similar but independent console library). Most notably, one of these enhanced libraries (wrapped by the appropriate Python module) will let you add color to your interface.

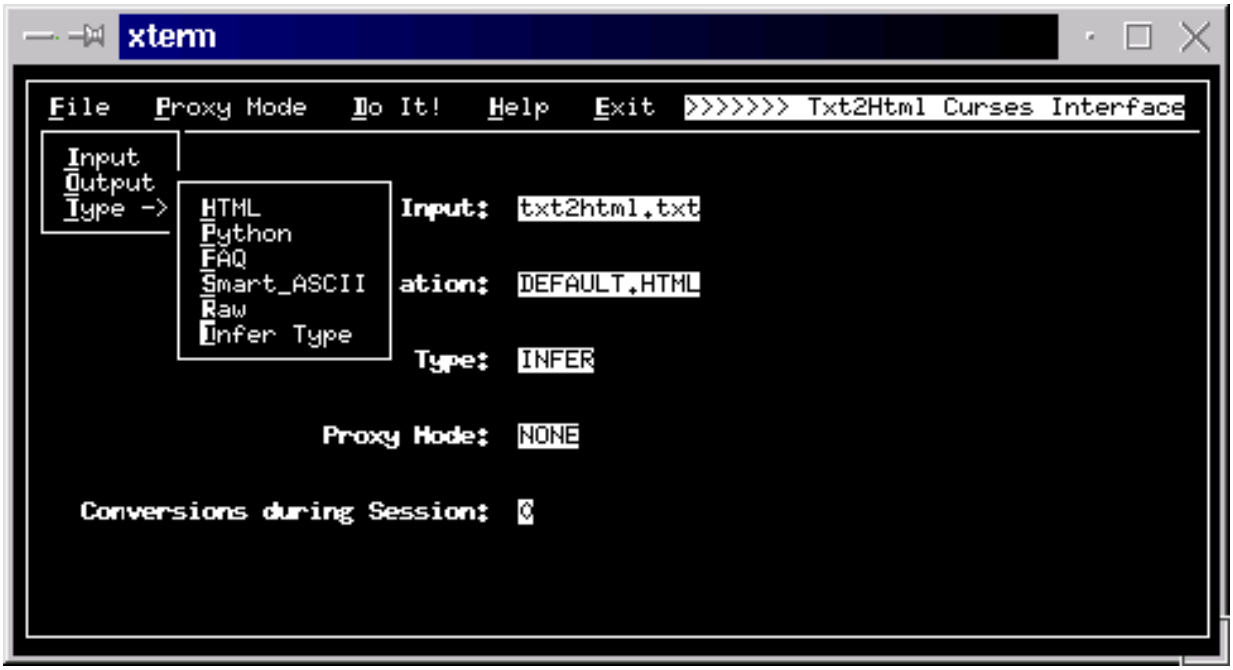
On the other hand, a number of high-level widget libraries, built on top of curses (or `ncurses` / `slang`), add features like buttons, menus, scroll bars, and various common interface devices. If you've seen applications developed with libraries such as Borland's TurboWindows (for DOS), you know how attractive these features can be in a text-mode console. There is nothing in the widget libraries that you **could not** do yourself with just curses, but you might as well take advantage of the work that other programmers have done on high-level interfaces. See the Resources section for links to the modules mentioned.

This article covers only the features of curses itself. Since the curses module is part of the standard distribution, you can expect it to be available and functional without requiring you to download support libraries or other Python modules (at least on Linux or UNIX systems). It's useful to have an understanding of the base support provided by curses even if only as a foundation for understanding higher-level modules. If you don't use those other modules, it's quite easy to build attractive and useful text-mode applications in Python using curses alone. Pre-release notes suggest that Python 2.0 will include an enhanced version of curses, but this should be backward-compatible with the version explained here in any case.

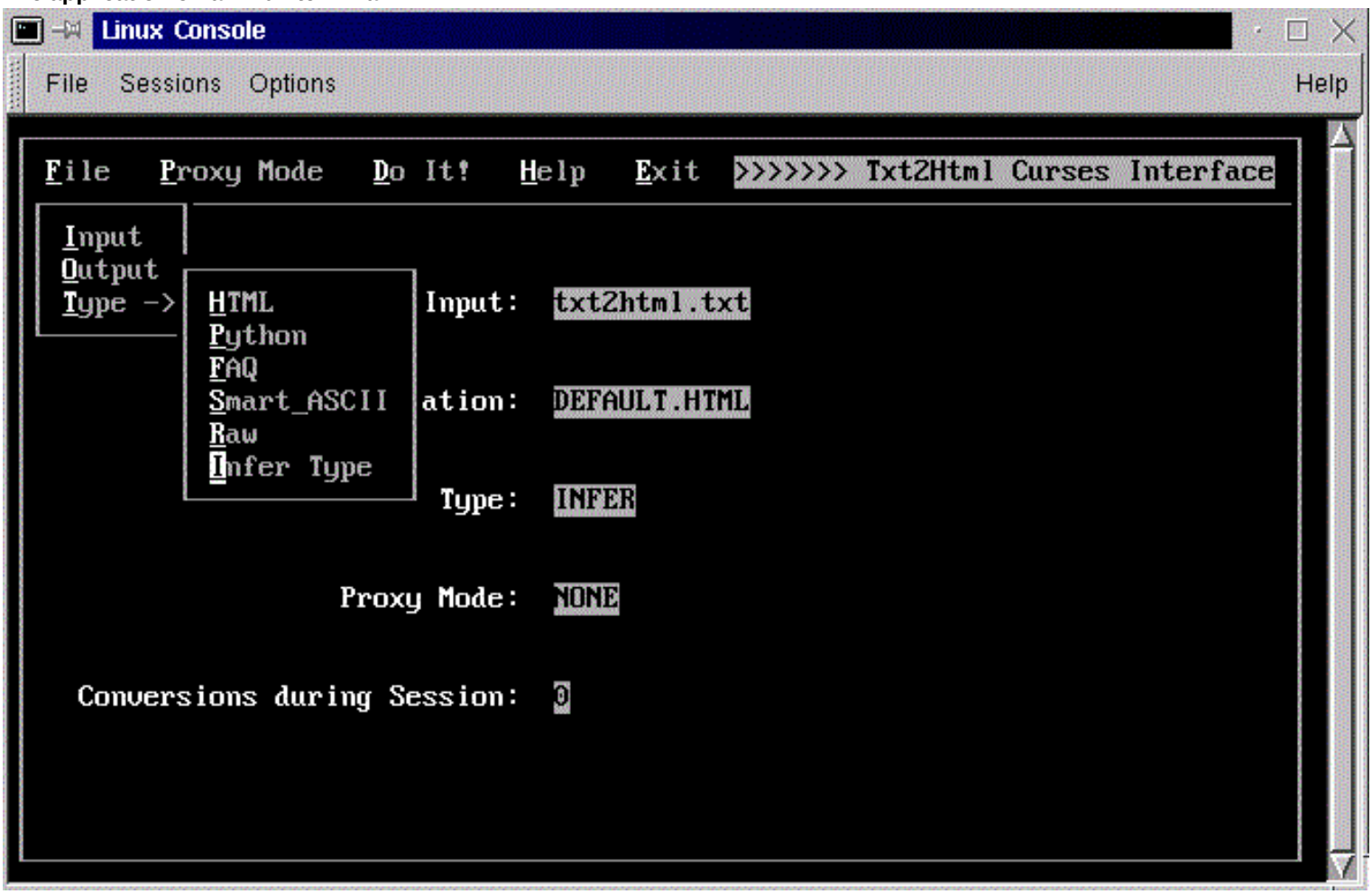
The application
As a test application for this article, I will discuss a wrapper I wrote for Txt2Html (a text-to-HTML conversion program introduced in "Charming Python: My first Web-based filtering proxy". Txt2Html works in several ways. But for the purposes of this article, we are interested in Txt2Html as it is run from the command-line. One way to operate Txt2Html is to feed it a bunch of command-line arguments indicating various aspects of the conversion to be performed, and then to let the application run as a batch process. For occasional usage, a more user-friendly interface might present an interactive selection screen that leads the user through conversion options (providing visual feedback of options selected) before performing the actual conversion.

The interface to curses_txt2html is based on a familiar topbar menu with drop-downs and nested submenus. All of the menu-related functions were done "from scratch" on top of curses. While these menus lack some of the features of more sophisticated curses wrappers, their basic functionality is implemented in a moderate number of lines using only curses. The interface also features a simple scrolling help box and several user-input fields. Below are screenshots of the application showing the general layout and style.

The application on an X terminal



The application on a Linux terminal



Wrapping a curses application

The basic element of curses programming is the window object. A window is a region of the actual physical screen with an addressable cursor whose coordinates are relative to the window. Windows can be moved around, and can be created and deleted independently of other windows. Within a window object, input or output actions take place at the cursor, which is usually set explicitly by input and output methods but can also be modified independently.

After initializing curses, stream-oriented console input and output is modified in various ways or disabled entirely. This is basically the whole point of using curses. But once streaming console interaction is changed, Python traceback events are not displayed in a normal

manner in the case of program errors. Andrew Kuchling solves this problem with a good top-level framework for curses programs (see his tutorial in Resources).

The following template (basically the same as Kuchling's) preserves the error-reporting capabilities of normal command-line Python:

Top level setup code for Python [curses] program

```
import curses, traceback
if __name__=='__main__':
  try:
      # Initialize curses
      stdscr=curses.initscr()
      # Turn off echoing of keys, and enter cbreak mode,
      # where no buffering is performed on keyboard input
      curses.noecho()
      curses.cbreak()

      # In keypad mode, escape sequences for special keys
      # (like the cursor keys) will be interpreted and
      # a special value like curses.KEY_LEFT will be returned
      stdscr.keypad(1)
      main(stdscr)                    # Enter the main loop
      # Set everything back to normal
      stdscr.keypad(0)
      curses.echo()
      curses.nocbreak()
      curses.endwin()                 # Terminate curses
  except:
      # In event of error, restore terminal to sane state.
      stdscr.keypad(0)
      curses.echo()
      curses.nocbreak()
      curses.endwin()
      traceback.print_exc()           # Print the exception
```

The `try` block does some initialization, calls the `main()` function to do the actual work and then performs a bit of final cleanup. If something goes wrong, the `except` block restores the console to its default state and reports the exceptions encountered.

Our main() event loop
Now let's take a look at the main() function to see what `curses_txt2html` does:

curses_txt2html.py main() function and event loop

```
def main(stdscr):
    # Frame the interface area at fixed VT100 size
    global screen
    screen = stdscr.subwin(23, 79, 0, 0)
    screen.box()
    screen.hline(2, 1, curses.ACS_HLINE, 77)
    screen.refresh()

    # Define the topbar menus
    file_menu = ("File", "file_func()")
    proxy_menu = ("Proxy Mode", "proxy_func()")
    doit_menu = ("Do It!", "doit_func()")
    help_menu = ("Help", "help_func()")
    exit_menu = ("Exit", "EXIT")
    # Add the topbar menus to screen object
    topbar_menu((file_menu, proxy_menu, doit_menu,
                 help_menu, exit_menu))

    # Enter the topbar menu loop
    while topbar_key_handler():
        draw_dict()
```

The `main()` function can be most easily understood in terms of three sections separated by blank lines.

The first section performs some general setup of our application's appearance. To establish some predictable spacing between application elements, the interactive area is limited to an 80 x 25 VT100/PC screen size (even if an actual terminal window is larger). The program draws a box around this sub-window and uses a horizontal line for visual offset of the topbar menus.

The second section establishes the menus used by our applications. The function `topbar_menu()` performs a little bit of magic in binding hotkeys to application actions and in displaying menus with the desired visual attributes. Check out the source archive (see Resources) to see all of the code. `topbar_menu()` should be pretty generic. (You are welcome to incorporate it into your own applications.) The important thing is that once the hotkeys are bound, they `eval()` whatever string is contained in the second element

of the tuple associated with a menu. Activating the "File" menu in the above setup, for example, will call "eval("file_func()")". So the application is required to define a function called `file_func()`, which is required to return a Boolean value indicating whether an application end-state has been reached.

The third section consists of just two lines, but this where the whole application actually runs. The function `topbar_key_handler()` does pretty much what its name suggests: it waits for keystrokes and then handles them. The key handler might return a Boolean false value. (If it does, the application ends.) In this application, the key handler consists of a check for the keys that were bound by the second section. But even if your curses application does not bind keys like this, you'll still want to use a similar event loop. The key point is that your handler will probably use a line like this:

```
c = screen.getch()   # read a keypress
```

The call to `draw_dict()` is the only code directly within the event loop. This function draws some values in a few locations in the `screen` window. But in your application you will probably want to include a line like:

```
screen.refresh()   # redraw the screen w/ any new output
```

inside your drawing/refresh function (or just inside the event loop itself).

Getting user input

A curses application gets all its user input in the form of keypress events. We have already seen the `.getch()` method, so let's look at an example that combines `.getch()` with the other input method, `.getstr()`. Below is an abbreviated version of the `file_func()` function we mentioned earlier (it's activated by the "File" menu).

curses_txt2html.py file_func() function

```
def file_func():
  s = curses.newwin(5,10,2,1)
  s.box()
  s.addstr(1,2, "I", hotkey_attr)
  s.addstr(1,3, "nput", menu_attr)
  s.addstr(2,2, "O", hotkey_attr)
  s.addstr(2,3, "utput", menu_attr)
  s.addstr(3,2, "T", hotkey_attr)
  s.addstr(3,3, "ype", menu_attr)
  s.addstr(1,2, "", hotkey_attr)
  s.refresh()
  c = s.getch()
  if c in (ord('I'), ord('i'), curses.KEY_ENTER, 10):
      curses.echo()
      s.erase()
      screen.addstr(5,33, " "*43, curses.A_UNDERLINE)
      cfg_dict['source'] = screen.getstr(5,33)
      curses.noecho()
  else:
      curses.beep()
      s.erase()
  return CONTINUE
```

This function combines several curses features. The first thing it does is create another window object. Since this new window object is the actual drop-down menu for the "File" selection, the program draws a frame around it with the `.box()` method. Within the window `s`, the program draws several drop-down menu options. A slightly laborious method is used so that the hotkey for each option will be highlighted to contrast with the rest of the option description. (Take a look at `topbar_menu()` in the full source (see Resources) for a somewhat more automated handling of the highlights.) The final `.addstr()` call moves the cursor to the default menu option. As with the main screen, `s.refresh()` actually displays the elements that were drawn to the window object.

After drawing the drop-down menu, the program gets the user's selection with the simple `s.getch()` call. In the demonstration application, menus respond only to hotkeys and not to arrow-key selection or movable highlight bars. These more sophisticated menuing functions could be built by capturing additional key actions and setting up event loops within drop-down menus. But the example suffices to illustrate the concept.

Next, the program compares the keystroke just read against various hotkey values. In our case, a drop-down menu option can be activated by an upper or lower case version of its hotkey and the default option can be activated with the ENTER key. (The curses special key constants do not seem to be entirely reliable, and I found that I had to add the actual ASCII value "10" in order to trap the ENTER key.) Notice that if you want to perform a comparison to a character value, you'll want to wrap the character's string in the `ord()` built-in Python function.

When the "Input" option is selected the program uses the `.getstr()` method, which provides field entry with crude editing capability (you can use the backspace key). Entry is terminated by the ENTER key, and the method returns whatever value was entered. This value will generally be assigned to a variable, as in the above example.

To help visually distinguish the entry field, I used a little trick to pre-underline the area where data entry would occur. Doing this is by

any means necessary, but it adds a bit of visual flair. The underline is performed by the line:

```
screen.addstr(5,33, " "*43, curses.A_UNDERLINE)
```

Of course the program also has to remove the field entry emphasis, which it does within the `draw_dict()` refresh function with the line:

```
screen.addstr(5,33, " "*43, curses.A_NORMAL)
```

Conclusion

The techniques outlined here, along with those used in the full application source code (see Resources), should get you started with curses programming. Play with it a bit. It's not hard to work with. One nice thing is that the curses library can be accessed by many languages other than Python, so what you learn using Python's curses module is mostly transferable elsewhere.

If the base curses module proves to be more limited than you would like, the Resources section provides links to a number of modules that add to the capabilities of curses and provide a nice gentle path for growth.

Resources

- Andrew Kuchling has written an introductory tutorial on curses programming, titled *Curses Programming With Python*. Parts of this article are inspired by Kuchling's examples, although it covers somewhat different (mostly higher level) elements of curses programming.
- Visit the best general starting place for information on text-based user interface tools in Python.
- Python `ncurses` is an enhanced module to support a larger range of `ncurses` functionality than Python 1.5.2 curses does. There are preliminary plans to have `ncurses` replace curses in Python 2.0. `ncurses`.
- `Tinter` is a module of high-level widgets built on top of curses. `Tinter` supports buttons, text boxes, dialog boxes, and progress bars.
- An under-publicized (and somewhat hard to track down) alternative to ncurses and other various wrappers is the combination of `slang` and `newt` with the python wrapper module `snack`. `slang` does roughly what curses does, and `newt` does roughly what `Tinter` does.
- Look at some examples of `snack`.
- `pcrt` is a module for direct ANSI escape-code screen access. This writes to specific locations on screen with specific colors and attributes. It is a low-level interface (even more so than curses) and will only work on consoles that support ANSI escape-codes (which is most of them), but it is a nice way to add some splash to your text-mode applications.
- `dialog` is a Python wrapper around the Linux `dialog` utility. The utility (with its Python wrapper) lets you create yes/no, menu, input, message, text, info checklist, and radiolist dialogs. You can do a lot very quickly using this utility and module if the platform restriction is not a problem (the target Linux distribution will need to have `dialog`, of course).
- Download files used and mentioned in this article.

About the author

David Mertz believes that God gave us the keyboard and the TTY while all other interface devices are mere human artifice. David may be reached at mertz@gnosis.cx. His life may be poured over at http://gnosis.cx/publish/. Suggestions and recommendations on this, past, or future columns are welcomed.