Curses Programming in Python

Originalartikel

Backup

<html> The

```
curses
```

and

```
ncurses
```

(new curses) libraries go back to 1980's and 90's and provide an API to create textual user interfaces (TUI). If you write a command-line application, you should consider using curses to implement functionality you could not otherwise do with standard console output. The text editor

```
nano
```

is a good example of a

```
ncurses
```

application. We will look at how to use this library in Python. Read more about curses programming from one of the ncurses authors, Thomas E. Dickey, who also worked on

```
xterm
```

and

```
lynx
```

among other things. https://invisible-island.net/. Another author of ncurses was Eric S. Raymond, who has a bunch of awesome writings at http://www.catb.org/~esr/. The official Python curses tutorial is really good, make sure to check it out as well at https://docs.python.org/3/howto/curses.html. The full API documentation is also available at https://docs.python.org/3/library/curses.html. There are lots of useful functions in the full API that are not covered here. I strongly encourage you to browse the full documentation. This tutorial will serve as an introduction to common tasks. If you want to check out a simple finished project that uses Python curses, check out the issh DevDungeon project which creates a menu for choosing SSH connections. The

```
curses
```

package comes with the Python standard library. In Linux and Mac, the curses dependencies should

already be installed so there is no extra steps needed. On Windows, you need to install one special Python package,

```
windows-curses
```

available on PyPI to add support. <pre class=„bash“># Needed in Windows only python -m pip install windows-curses </pre> You can verify everything works by running a Python interpreter and attempting to

```
import curses
```

. If you do not get any errors, you are in good shape. <pre class=„python“>>>> import curses >>> </pre> There are a few important concepts to understand before digging in to the advanced concepts. Some primary things you will need to understand are: The concept of „windows“ How to initialize and shut down curses to get a main window How to add text, clear, and refresh windows These primary topics will be covered first before we look in to some more common tasks like modifying terminal properties to turn off the cursor, listening for key presses, centering text, and more. Now that we are confident the curses import worked, we can try to initialize it. When you initialize curses, it creates a new window and clears the screen, displaying your new window. This example will show how to initialize curses and obtain a window to work with. We will call the main window

```
screen
```

. <pre class=„python“>import curses print(„Preparing to initialize screen…“) screen = curses.initscr() print(„Screen initialized.“) screen.refresh() curses.napms(2000) curses.endwin() print(„Window ended.“) </pre> After running this example, you might be surprised by the behavior. It will display a blank screen for one second, and then you will see all of your print statements at the end when you return to the terminal. The print statements continue going to standard output and will remain there, even though it is not visible. It's creating a special buffer that is being displayed in the terminal, independent of STDOUT. If print statements don't go to the screen, then how do you get text on to this fancy new screen we initialized? Now that we know how to initialize a blank screen and clean up at the end, let's try adding text to the screen. This example shows how to initialize the screen like before, but taking it further and adding a string to the screen. Note that you need to refresh the screen after making changes. <pre class=„python“>import curses screen = curses.initscr() # Update the buffer, adding text at different locations screen.addstr(0, 0, „This string gets printed at position (0, 0)“) screen.addstr(3, 1, „Try Russian text: Привет“) # Python 3 required for unicode screen.addstr(4, 4, „X“) screen.addch(5, 5, „Y“) # Changes go in to the screen buffer and only get # displayed after calling `refresh()` to update screen.refresh() curses.napms(3000) curses.endwin() </pre> With this knowledge, you can draw text anywhere you want, all over the screen! You can do all kinds of stuff with just this knowledge alone. You may be wondering how you know your limits, like what is the maximum row and maximum column? If you want to fill up the screen or draw a border, what rows and columns should you use? We'll cover that in a later section. You could go through cell by cell and fill it with a black background space character to reset the terminal, but there is a convenient function to clear the screen, with the

```
clear()
```

function of a window. <pre class=„python“>import curses screen = curses.initscr()

screen.addstr(„Hello, I will be cleared in 2 seconds.“) screen.refresh() curses.napms(2000) # Wipe the screen buffer and set the cursor to 0,0 screen.clear() screen.refresh() curses.napms(2000) curses.endwin() </pre> Curses provides two important concepts: windows and pads. So far we have been working with one window, the main screen. You can create multiple windows of different sizes and place them around the screen. You can do all the same things we showed with the „screen“ window like

```
addstr()
```

and

```
addch()
```

. You can save the contents of a window to a file, fill the contents of the window from a file, add borders, add background characters, create sub-windows, and more. Check out the full API documentation at https://docs.python.org/3/library/curses.html#window-objects. This example shows how to create a window, add some text, and then move the window to a different location. It demonstrates how text is automatically wrapped when the window width is reached. <pre class=„python“>import curses # The `screen` is a window that acts as the master window # that takes up the whole screen. Other windows created # later will get painted on to the `screen` window. screen = curses.initscr() # lines, columns, start line, start column my_window = curses.newwin(15, 20, 0, 0) # Long strings will wrap to the next line automatically # to stay within the window my_window.addstr(4, 4, „Hello from 4,4“) my_window.addstr(5, 15, „Hello from 5,15 with a long string“) # Print the window to the screen my_window.refresh() curses.napms(2000) # Clear the screen, clearing my_window contents that were printed to screen # my_window will retain its contents until my_window.clear() is called. screen.clear() screen.refresh() # Move the window and put it back on screen # If we didn't clear the screen before doing this, # the original window contents would remain on the screen # and we would see the window text twice. my_window.mvwin(10, 10) my_window.refresh() curses.napms(1000) # Clear the window and redraw over the current window space # This does not require clearing the whole screen, because the window # has not moved position. my_window.clear() my_window.refresh() curses.napms(1000) curses.endwin() </pre> Pads are basically windows that can have content that is larger than its display area. Pads are essentially scrollable windows. Read more about pads at https://docs.python.org/3/howto/curses.html#windows-and-pads. To create a pad, you do it very similarly, using

```
curses.newpad()
```

instead of

```
curses.newwin()
```

. When calling

```
refresh()
```

on the pad, you have to provide a few extra arguments though. To refresh the pad you have to tell it See the documentation for

```
refresh()
```

at https://docs.python.org/3/library/curses.html#curses.window.refresh. <pre class=„python“>import curses screen = curses.initscr() # Make a pad 100 lines tall 20 chars wide # Make the pad large enough to fit the contents you want # You cannot add text larger than the pad # We are only going to add one line and barely use any of the space pad = curses.newpad(100, 100) pad.addstr(„This text is thirty characters“) # Start printing text from (0,2) of the pad (first line, 3rd char) # on the screen at position (5,5) # with the maximum portion of the pad displayed being 20 chars x 15 lines # Since we only have one line, the 15 lines is overkill, but the 20 chars # will only show 20 characters before cutting off pad.refresh(0, 2, 5, 5, 15, 20) curses.napms(3000) curses.endwin() </pre> Now let's look at some other common tasks. Note that when you call

```
curses.endwin()
```

it returns your original terminal, but if you call

```
screen.refresh()
```

again after that, you can get your screen back, and you'd have to call

```
curses.endwin()
```

again to bring back STDOUT once again. This can be useful if you want to temporarily hide your screen and do something in the original terminal before jumping back in to your screen. This example shows you how to „shell out“. This allows you to hide your screen and enter a command prompt to do some tasks and then when you exit the shell, go back to your custom screen. <pre class=„python“>import curses import subprocess import os # Create a screen and print hello screen = curses.initscr() screen.addstr(„Hello! Dropping you in to a command prompt...\n“) print(„Program initialized...“) screen.refresh() curses.napms(2000) # Hide the screen, show original terminal, restore cursor position curses.endwin() # Update screen in background screen.addstr(„I'll be waiting when you get back.\n“) # Drop the user in a command prompt print(„About to open command prompt...“) curses.napms(2000) if os.name == 'nt':

```
  shell = 'cmd.exe'
```

else:

```
  shell = 'sh'
```

subprocess.call(shell) # When the subprocess ends, return to our screen. # also restoring cursor position screen.refresh() curses.napms(2000) # Finally go back to the terminal for real curses.endwin() </pre> When using curses it's usually very important to make sure you are working within the boundaries of the current terminal. If your application does not adapt to the terminal size, you can at least check the size and ensure it meets your minimum required size. A safe default size to assume is generally 80×24. Let's see how we figure out exactly what size the user's terminal is with curses. <pre class=„python“>import curses screen = curses.initscr() num_rows,

num_cols = screen.getmaxyx() curses.endwin() print(„Rows: %d“ % num_rows) print(„Columns: %d“ % num_cols) </pre> By knowing the width and height of the terminal, you can calculate the center of the screen and position text accordingly. This shows how to use the screen width and height to find the center point of the screen and print text that is properly aligned. This example shows how to calculate the middle row and the proper offset to make the text look centered. <pre class=„python“># Draw text to center of screen import curses screen = curses.initscr() num_rows, num_cols = screen.getmaxyx() # Make a function to print a line in the center of screen def print_center(message):

```
  # Calculate center row
  middle_row = int(num_rows / 2)
  # Calculate center column, and then adjust starting position based
  # on the length of the message
  half_length_of_message = int(len(message) / 2)
  middle_column = int(num_cols / 2)
  x_position = middle_column - half_length_of_message
  # Draw the text
  screen.addstr(middle_row, x_position, message)
  screen.refresh()
```

print_center(„Hello from the center!“) # Wait and cleanup curses.napms(3000) curses.endwin() </pre> The terminal is always keeping track of the current cursor position. After you write text to the screen with

```
addstr()
```

the cursor ends up in the cell just after your text. You can hide the cursor so it does not blink and is not visible to the user. You can just as easily turn it back on. You can control the behavior using the

```
curs_set()
```

function as demonstrated in this example. <pre class=„python“>import curses screen = curses.initscr() curses.curs_set(0) screen.addstr(2, 2, „Hello, I disabled the cursor!“) screen.refresh() curses.napms(3000) curses.curs_set(1) screen.addstr(3, 2, „And now the cursor is back on.“) screen.refresh() curses.napms(3000) curses.endwin() </pre> If we were trying to make an awesome menu for our program, it could use some colors. Curses allows you to change the color and the style of the text. You can make text bold, highlighted, or underline as well as change the color of the foreground and background. Notice in this example how you can call

```
addstr()
```

without passing it an x,y coordinate pair, it outputs to the current cursor location. The

```
\n
```

newline character will control the cursor as normal and move it to the beginning of the next line. <pre class=„python“>import curses screen = curses.initscr() # Initialize color in a separate step curses.start_color() # Change style: bold, highlighted, and underlined text screen.addstr(„Regular text\n“) screen.addstr(„Bold\n“, curses.A_BOLD) screen.addstr(„Highlighted\n“, curses.A_STANDOUT) screen.addstr(„Underline\n“, curses.A_UNDERLINE) screen.addstr(„Regular text again\n“) # Create a

custom color set that you might re-use frequently # Assign it a number (1-255), a foreground, and background color. curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE) screen.addstr(„RED ALERT!\n“, curses.color_pair(1)) # Combine multiple attributes with bitwise OR screen.addstr(„SUPER RED ALERT!\n“, curses.color_pair(1) | curses.A_BOLD | curses.A_UNDERLINE | curses.A_BLINK) screen.refresh() curses.napms(3000) </pre> In all of these examples so far though, we've been using

```
curses.napms()
```

function to hold the curses screen open for a few seconds before it closes itself. How do we give the user some control? Instead of sleeping to hold the screen open for a few seconds, let's wait until the user presses the 'q' key to quit. This example will show how to get a keypress from the user. <pre class=„python“>import curses screen = curses.initscr() screen.addstr(„Press any key...“) screen.refresh() c = screen.getch() curses.endwin() # Convert the key to ASCII and print ordinal value print(„You pressed %s which is keycode %d.“ % (chr©, c)) </pre> Depending on what settings you modify in the terminal, you might want to do some cleanup to restore the state of the terminal For example, if you turned off the cursor, disabled echo, turned on the keypad keys, or turned on cbreak mode, you will likely want to restore the original state of the terminal. Here are a few examples to keep in mind: <pre class=„python“>curses.nocbreak() # Turn off cbreak mode curses.echo() # Turn echo back on curses.curs_set(1) # Turn cursor back on # If initialized like `my_screen = curses.initscr()` my_screen.keypad(0) # Turn off keypad keys </pre> In the last section we looked at manually resetting the terminal manually. There is a better way provided in the standard library with

```
curses.wrapper()
```

. Sometimes you might encounter an unexpected exception in your application that can cause it to crash. This can potentially leave your terminal in a bad state that is unusable after the crash. To accomodate this, the

```
curses
```

package provides a function that can wrap your whole application that way it can handle restoring things to a sane state. To use the wrapper, create a function that takes one argument: the screen. Then, call

```
wrapper()
```

and pass it your function that will operate with the screen. The

```
wrapper()
```

function takes care of initializing the curses screen that is normally done with

```
curses.initscr()
```

and also takes care of calling

```
curses.endwin()
```

when your function is over or an exception is caught. https://docs.python.org/3/library/curses.html#curses.wrapper This example creates a

```
main()
```

function that will be passed to

```
wrapper()
```

. The main function will intentionally throw an exception to show how the wrapper function will exit somewhat gracefully and restore the terminal to a decent state. <pre class=„python“>from curses import wrapper def main(main_screen):

```
  raise Exception
```

wrapper(main) </pre> Here are some other libraries that might be useful when trying to create curses-based applications. npyscreen curtsies urwid blessings </html>