Python Curses Programming HOWTO

Original text by:

Pradeep Pradala

Python Translation by:

Alan J Gauld

Table of Contents

Revision History.....v
Introduction.....vii
The Translation to Python.....vii
Purpose/Scope of the document.....vii
About the Original Document and Programs.....viii
Copyright.....ix
Programming in a Terminal.....x
What is NCURSES?.....x
What We Can Do with curses?.....xi
Where to get it.....xii
1 Hello World !!!.....1
1.1 Using the curses module.....1
1.2 Dissection.....1
1.3 About initscr().....1
1.4 The mysterious refresh().....2
1.5 Reading character input.....3
1.6 About endwin().....3
2 Initialization.....5
2.1 raw() and cbreak().....5
2.2 echo() and noecho().....5
2.3 keypad().....6
2.4 halfdelay().....6
2.5 Miscellaneous Initialization Functions.....6
2.6 An Example.....6
2.7 A Word about Windows.....7
2.8 The curses.wrapper() Function.....8
3 Output Functions.....9
3.1 addch() Method.....9
3.2 addstr() Method.....10
3.3 insch() Method.....11
3.4 inssttr() Method.....11
3.5 A Word of Caution.....11

4 Input functions....13
4.1 getch() category of methods....13
4.2 getstr() category of methods....14
4.3 An Example....14
4.4 Reading from the Screen....15
5 Attributes....17
5.1 The Details....18
5.2 attron() vs attrset()....19
5.3 attr_get()....19
5.4 chgat() functions....20
6 Windows....23
6.1 The Basics....23
6.2 Let there be a Window!!!....23
6.3 Explanation....25
6.4 Some Other Stuff in the Example....26
6.5 Other Border Functions....26
6.6 Sub-Windows....28
7 Colors....31
7.1 The Basics....31
7.2 Changing Color Definitions....32
7.3 Color Content....33
8 Interfacing with the Keyboard....35
8.1 The Basics....35
8.2 Simple Key Usage....36
9 Interfacing with the Mouse....39
9.1 The Basics....39
9.2 Getting Events....40
9.3 Putting it all Together....41
9.4 Miscellaneous Functions....44
10 Screen Manipulation....45
10.1 getyx() functions....45
10.2 Screen Dumping....45
10.3 Window Dumping....45
11 Miscellaneous features....49

11.1 curs_set()....49
11.2 Temporarily Leaving Curses Mode....49
11.3 ACS_ Variables....50
11.4 And finally....50
12 curses.panel module....51
12.1 The Basics....51
12.2 Panel Window Browsing....53
12.3 Using User Pointers....55
12.4 Moving and Resizing Panels....55
12.5 Hiding and Showing Panels....60
12.6 panel_above() & panel_below() Methods....62
13 Tools and Widget Libraries....63
13.1 curses.textpad....63
13.2 dialog....64
14 A Case Study – The Totalizer....67
14.1 Totalizer Design Summary....67
14.2 The Cell Class....68
14.3 The Grid Class....69
14.4 The SummingGrid Class....73
14.5 The Totalizer Class....73
14.6 The Driver Code....76
14.7 Things to Consider....77
15 Just For Fun !!!....79
15.1 The Game of Life....79
15.2 Magic Square....79
15.3 Towers of Hanoi....79
15.4 Queens Puzzle....80
15.5 Shuffle....80
15.6 Typing Tutor....80
16 References....81
16.1 Online....81
16.2 Books....81

Revision History

Python Curses Programming HOWTO

by Alan J Gauld

alan.gauld@yahoo.co.uk

Issue 1.0 - 2020-10-27

Revised by A J Gauld

Fixed typos and minor tweaks following final review.

Revision 0.4 – 2020-08-18

Revised by AJ Gauld

Tidied up and issued for final review.

Revision 0.3 – 2020-08-12

Revised by AJ Gauld

Editing tweaks and corrections following reviews from Python mailing list. Reformatted pages to book style with headers/footers/chapter heads etc. Significant editing of text descriptions to better fit the Python code.

Extended subwindow example to better demonstrate relationships of different window types.

Added a grid based case-study chapter to pull everything together within a larger context using OOP application design.

Revision 0.2 - 2020-06-12

Revised by: AJ Gauld

Still retaining as much as possible of the original material by Pradeep Padala but removing irrelevant topics and references for the Python curses module.

New sections added for subwindows, the insertion and screen reading operations and the Python wrapper function and the textpad widget.

Submitted to Python mailing list for review and feedback.

Revision 0.1 – 2020-05-15

Revised by: AJ Gauld

Translation to the Python curses library by Alan Gauld

Most text as-is, code translated to Python. Irrelevant C stuff deleted.

Based on Revision 1.9 of the original document. See the original for full change history prior to R1.9.

Send comments on the Python translation to this address
alan.gauld@yahoo.co.uk

Introduction

The Translation to Python

This document is a conversion of the Linux HOWTO document on programming with ncurses by Pradeep Pradala.

https://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/

Ncurses is a library used to control cursor movement as well as enabling color and mouse control and even multiple windows, all within a traditional terminal environment.

Ncurses is written in C but a wrapper has been provided for the Python language and is included in the Python Standard Library for Unix-like distributions as the `curses` module. While the module covers the majority of the ncurses library functions there are a few areas where it does not follow the C version, since Python provides superior facilities within the language. This document takes account of those changes and explains the Pythonic alternatives. There are also a few lesser used C functions that do not appear in the Python module, although they can usually be worked around. One example, the missing `get_attr()` function, is covered in the text.

The `curses` module documentation within the Python Standard Library is primarily a reference and not helpful to those with no experience in using curses. This is largely because the module is just a thin wrapper over the C library and the documentation merely indicates which of the library functions are available within Python and their function signature. The user is assumed to already be familiar with the C version of curses and its usage.

Although several online tutorials already exist, including a fairly short Python How-To guide, none are as complete as Pradeep’s Linux HOWTO. I therefore took it upon myself to create a translation of his document into Python. Initially I tried to make minimal changes to Pradeep’s text but it became obvious that the differences in the libraries are more than superficial and a significant amount of rewriting would be necessary. As a result the final document retains Pradeep’s structure and example programs (while adding a few extra examples) but most of the text has been rewritten to suit the Python implementation of curses.

Purpose/Scope of the document

This document is intended to be an "All in One" guide for programming with Python curses and its sister modules.

We graduate from a simple "Hello World" program to more complex multi-window manipulation. No prior experience in curses is assumed. The writing is informal, but a lot of detail is provided for each of the examples. It is assumed that the reader is already experienced in regular Python and no discussion is made of standard Python language features or idioms.

About the Original Document and Programs

All the programs in the original document are available in zipped form at the Linux Documentation Project web site. Instructions on how to build and run them are in the orignal document along with a file structure map.

Various formats of the original document exist

- Acrobat PDF Format
- PostScript Format
- Multi-page HTML
- Single-page HTML

Much of the credit must go to Pradeep and his original collaborators. I have simply translated the code and updated the text consistent with the changes to the code and programming environment and added a couple of new sections more relevant to Python.

Copyright

Python Translation Copyright © 2020 by Alan J Gauld

Original Copyright © 2001 by Pradeep Padala.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, distribute with modifications, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE ABOVE COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name(s) of the above copyright holders shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization.

Programming in a Terminal

This section covers some of the historical background to the curses system. If you are impatient to get started you can safely jump ahead to Section 1 and dive right in. If you like to understand the background to things then carry on reading.

In the early days of teletype (TTY) and video display terminals (VDT), terminals were located away from computers and were connected to them through serial cables. The terminals could be controlled by sending particular sequences of bytes. All the terminal's capabilities (such as moving the cursor to a new location, erasing part of the screen, scrolling the screen, changing modes etc.) could be controlled by such a series of bytes. These control sequences are usually called "escape sequences", because they start with an escape (0x1B) character. Even today, with proper emulation, we can send escape sequences to the emulator and achieve the same effect on a terminal window.

Suppose you wanted to print a line in color. Try typing this on your console:

```
echo "^[[0;31;40mIn Color"
```

The first character is an escape character, which looks like two characters ^ and [. To be able to type it, you have to press CTRL+V and then the ESC key. All the others are normal printable characters.

You should be able to see the string "In Color" in red. If it stays that way then revert back to the original mode by typing this:

```
echo "^[[0;37;40m"
```

Now, what do these magic characters mean? Difficult to comprehend? They might even be different for different terminals. So the designers of UNIX invented a mechanism named *termcap*. It is a file that lists all the capabilities of a particular terminal, along with the escape sequences needed to achieve a particular effect. Later this was replaced by *terminfo*. Without delving too much into details, this mechanism allows application programs to query the terminfo database and obtain the control characters to be sent to a particular terminal or terminal emulator for a given effect.

What is NCURSES?

You might be wondering what the import of all this technical gibberish is. In the above scenario, every application program is supposed to query the terminfo and perform the necessary stuff (sending control characters etc.). It soon became difficult to manage this complexity and this gave birth to *curses*. The name curses is a pun on the phrase "cursor optimization". The `curses` library forms a wrapper around working with raw terminal codes, and provides a highly flexible and efficient API (Application Programming Interface). It provides functions to move the cursor, create windows, produce

colors, play with the mouse etc. The application programs need not worry about the underlying terminal capabilities. The original library was developed at Berkely as part of BSD Unix. AT&T later developed their own version for SystemVR2 Unix. In 1993 a free-software clone was developed and released as ncurses under the auspices of the GNU project.

https://www.gnu.org/philosophy/free-sw.html

In short, ncurses is a library of functions that manages an application's display on character based terminals. The terms curses and ncurses are often used interchangeably.

A detailed history of ncurses can be found in the NEWS file from the source distribution. The current package is maintained by Thomas Dickey. You can contact the maintainers at *bug-ncurses@gnu.org*.

The Python wrapper around the ncurses library is part of the Python Standard Library on Unix-like distributions of Python. It translates Python function and method calls into calls to the underlying C library. Ncurses must also be installed on the host machine alongside Python.

What We Can Do with curses?

Curses not only creates a wrapper over terminal capabilities, but also gives a robust framework to create a nice looking UI (User Interface) in text mode. It provides functions to create windows etc. Its sister modules, `curses.panel`, `curses.ascii`, `curses.textpad` and `dialog` (a third party module) provide extensions to the basic curses library. One can create applications that contain multiple windows, panels and dialogs. Windows can be managed independently, can provide 'scrollability' and can even be hidden.

Panels extend the capabilities of ncurses to deal with overlapping and stacked windows. The textpad widget provides a minimal multi-line text editing control with emacs-like key bindings. The dialog module provides a set of standard dialogs which can be accessed from curses programs (as well as from regular Python CLI applications).

The `ascii` module is not covered here but provides capabilities to translate curses character codes into the more commonly used ASCII character set as well as functions to check for membership of various sets (such as control characters etc).

These are just some of the basic things we can do with curses. As we move along, we will see most of the capabilities of these modules.

Where to get it

All right, now that you know what you can do with curses, you must be raring to get started. The ncurses C library is usually shipped with your installation. In case you don't have the library or want to compile it on your own, you can fetch it from:

ftp://ftp.gnu.org/pub/gnu/ncurses/ncurses.tar.gz

The Python curses package should be included in the standard library installed with Python (except on Windows). You can check whether it is there by running, at the Python interactive prompt:

```
>>> import curses
```

If there are no error messages then it is there and all is well. If not then you will need to find a copy and download it into your Python library.

Note: For Windows users there are at least two ports of Python curses available but my personal preference is to use the Cygwin package which includes curses in its Python bundle along with a suitable terminal emulator and bash shell.

1 Hello World !!!

Welcome to the world of curses. Before we plunge into the module and look into its various features, let's write a simple program and say hello to the world.

1.1 Using the curses module

To use curses module functions, you have to import curses into your programs.

```
import curses
```

You may want to use an alias to reduce typing:

```
import curses as cur
```

Example 1. The Hello World !!! Program

```
import curses as cur

scr = cur.initscr()                # Start curses mode
scr.addstr("Hello World !!!")      # Print Hello World
scr.refresh()                      # Print it on to the real screen
scr.getch()                        # Wait for user input
cur.endwin()                       # End curses mode
```

1.2 Dissection

The above program prints "Hello World !!!" to the screen and exits. This program shows how to initialize curses and do screen manipulation and end curses mode. Let's dissect it line by line.

1.3 About initscr()

The function `initscr()` initializes the terminal in curses mode. In some implementations, it clears the screen and presents a blank screen. To do any screen manipulation using curses package this has to be called first. This function initializes the curses system and allocates memory for our present

window, which is the entire terninal screen (often called *stdscr* in curses documentation) and some other data-structures. (Under extreme cases this function might fail due to insufficient memory to allocate memory for curses library's data structures.) It returns a new Window object representing `stdscr`. You should note that in C curses there is an actual variable called `stdscr` that you can pass to functions that require a window argument. In Python things are slightly different and `initscr()` returns a window object which we can call anything we like. In this document I use the name `scr`, but you can use any name you like. However, the concept of `stdscr` as a window encompassing the whole available terminal space is important in curses so you will see references to `stdscr` throughout the text.

After this is done, we can do a variety of initializations to customize our curses settings. These details will be explained later . For now we will restrain ourselves to the simplest case and only use `initscr()`.

1.4 The mysterious refresh()

The next line `scr.addstr()` prints the string "Hello World !!!" on to the screen. This function can take various forms as we will see later. In its basic form as seen here it prints the data on a window at the current (y,x) coordinates. Curses is unusual in that it always places the `y` (row number) coordinate before the `x` (column number). The coordinates count from zero in both cases.

Notice that `addstr()` is a method of the window object returned by `initscr().` In Python all the curses functions are either module level functions or methods of the Window class.

Since we haven't yet moved the cursor anywhere our present co-ordinates are at (0,0), the string is therefore printed at the top-left corner of the window.

This brings us to that mysterious `scr.refresh()`. When we called `scr.addstr()` the data is actually written to a virtual window, which is not automatically written to the screen. The job of `scr.addstr()` is to update a few flags and data structures and write the data to a buffer. In order to show it on the screen, we need to call `scr.refresh()` and tell the curses system to dump the contents on to the physical display. It's a good idea to call refresh just before asking for user input, to ensure the user can see the screen.

The philosophy behind all this is to allow the programmer to do multiple updates on the imaginary screen or windows and do a refresh once all his screen updates are done. `scr.refresh()` checks the window and updates only the portion which has been changed. (You can force a full window refresh by calling the `scr.touchwin()` function before `scr.refresh().`) This improves performance and offers greater flexibility too. However, it is sometimes frustrating to beginners. A common mistake committed by beginners is to forget to call `scr.refresh()` after doing some updates through the `scr.addstr()` or `scr.addch()`class of functions. Another common error is to call the wrong window's refresh method.

1.5 Reading character input

The penultimate fuction is `scr.getch()` which as the name suggests reads a character. Note that it does not by default wait for the enter key, it reads each keypress as it is performed. It does however block until a keypress is present. In this case we ignore the character because we are only using it as a pause before ending the program.

1.6 About endwin()

And finally, don't forget to end the curses mode. Otherwise your terminal might behave strangely after the program quits. `cur.endwin()` frees the memory taken by the curses sub-system and its data structures and puts the terminal in normal mode. This function must be called after you are done with the curses mode. If your program does exit prematurely so that `endwin()` is not called, you may need to type one or more of the following commands:

```
$ stty sane
$ stty echo -nl
$ stty reset
```

to reset your terminal so that it displays properly. If that fails then you will probably need to close the terminal and open a new one!

Now that we have seen how to write a simple curses program let's get into the details. There are many functions that help customize what you see on screen and many features which can be put to full use.

Here we go...

2 Initialization

We now know that to initialize curses system the function `initscr()` has to be called. There are functions which can be called after this initialization to customize our curses session. We may ask the curses system to set the terminal to raw mode or initialize color or initialize the mouse etc. Let's discuss some of the functions that are normally called immediately after `initscr()`

2.1 raw() and cbreak()

Normally the terminal driver buffers the characters a user types until a new line or carriage return is encountered. However, most GUI style programs require that the characters be available as soon as the user types them. The above two functions are used to disable line buffering. The difference between these two functions is in the way control sequences like suspend (CTRL-Z) and interrupt (CTRL-C) are passed to the program. In the `raw()` mode these characters are directly passed to the program without generating a signal. It is then up to the programmer to catch them and do whatever seems appropriate. In the `cbreak()` mode these control characters are interpreted by the terminal driver, so that you can, for example hit CTRL-C to interrupt the program (but if you do, you will likely need to use `stty` to reset your terminal since `endwin()` will not have been called).

2.2 echo() and noecho()

When you use a terminal to interact with the computer the terminal sends the characters you type to the computer and the computer then sends them back to the terminal which displays them on screen. This is how you see what you have typed and is known as echoing. Without an echo you would be typing blind into the terminal. Sometimes echoing is deliberately turned off - for example when typing a password.

`echo()` and `noecho()` control the echoing of characters typed by the user to the terminal. `noecho()` switches off echoing. The reason you might want to do this is to gain more control over echoing or to suppress unnecessary echoing while taking input from the user through the `getch()` etc. functions. Many interactive programs call `noecho()` at initialization and do the echoing of characters in a controlled manner. It gives the programmer the flexibility of echoing characters at any place in the window without updating current (y,x) coordinates.

2.3 keypad()

This method enables the reading of function keys like F1, F2, arrow keys, mouse clicks etc.

Almost every interactive program enables this, as arrow keys are a major part of any User Interface. Do `win.keypad(True)` to enable this feature for any given window. You will learn more about key management in later sections of this document. Note that it is window specific so if you find that you are not able to recognize the special keys (or mouse clicks) check that you have set this for the window in question.

2.4 halfdelay()

This function is not often used but can be useful sometimes. Half-delay mode is similar to the cbreak mode in that characters typed are immediately available to the program, however half-delay adds a time delay of 'X' tenths of a second and then returns `ERR`, if no input is available. 'X' is the timeout value passed to the function `halfdelay()`.

This is useful when you want to ask the user for input, and if they don't respond within a certain time, we can do some thing else. One possible example is a timeout at the password prompt. Curiously you have to call `nocbreak()` to get out of half-delay mode (which will also exit cbreak mode of course, so you may need to immediately re-initialize `cbreak()` if you need its features).

2.5 Miscellaneous Initialization Functions

There are a few more functions which are sometimes used during initialization to customize curses behavior. They are not used as extensively as those mentioned above. Some of them are explained later where appropriate.

2.6 An Example

Let's write a program which will clarify the usage of these functions.

Example 2. Initialization Function Usage example

```
import curses as cur
scr = cur.initscr()       # Start curses mode
cur.raw()                 # Line buffering disabled
cur.noecho()              # Don't echo() while we do getch
scr.keypad(True)          # We get F1, F2 Arrow keys etc.

scr.addstr("Type any character to see it in bold\n")
ch = scr.getch()          # If raw() or cbreak() hadn't been called
                          # we have to press enter before it
                          # gets to the program
```

```
if ch == cur.KEY_F1:    # Without keypad enabled this will
    scr.addstr("F1 Key pressed")   # not get to us either
                        # Without noecho() some ugly escape
                        # characters might have been printed
                        # on screen
else:
    scr.addstr("The pressed key is ")
    scr.attron(cur.A_BOLD)
    scr.addstr(chr(ch))
    scr.attroff(cur.A_BOLD)

scr.refresh()           # Print it on to the real screen
scr.getch()             # Wait for user input
cur.endwin()            # End curses mode
```

This program is self-explanatory but I used functions which haven't been explained yet. The functions `attron()` and `attroff()` are used to switch text attributes on and off respectively. In the example I used them to print the characters in bold. These functions are explained in detail later.

2.7 A Word about Windows

Before we plunge into the myriad curses functions, let me clarify a few things about windows. A window is an area of the screen defined by the curses system. A window does not mean a bordered window with controls for sizing or moving etc, such as you usually see on GUI systems.

When `curses.initscr()` is called, it creates a memory structure representing the default window, conventionally called `stdscr,` which represents your full terminal screen. If you are doing simple tasks like printing a few strings, reading input etc, you can safely use this single window for all of your purposes. You can also however create new windows and call functions which explicitly work on the specified window. We will see all of these techniques later.

For example, if you call:

```
scr.addstr("Hi There !!!")
scr.refresh()
```

Curses prints the string on the `stdscr` buffer at the current cursor position. The call to `scr.refresh()` causes any changes in the stdscr buffer to be written to the physical display.

Say you have created other windows then these will have their own memory buffers. In the C library you have to call a function with a 'w' added to the usual function and explicitly pass the window buffer to the function:

```
waddstr(win, "Hi There !!!");
wrefresh(win);
```

However, in Python these functions have all been translated to methods of a Window class so you use the normal function name and simply attach it to the relevant window. In this case the C code above would become:

```
win.addstr(“Hi There!!!”)
win.refresh()
```

In addition, many of these text methods take a coordinate pair of lines and columns to indicate where within the window the text should appear. The coordinate system starts at `0` in both dimensions.

You can also add a text attribute at the end of the argument list to control colors and bold or underline etc. You will see more on that shortly.

2.8 The curses.wrapper() Function

A unique feature to the Python version of curses is the wrapper functionality. So far, every program we have written has included the standard `initscr()` and `endwin()` top and tail code as well as other initialization type things. To save us having to type this everytime Python provides a convenience function that takes care of the standard initialization and then calls a function that takes a window as its only argument. The wrapper also catches any errors and ensures that `endwin()` gets called to clean up the screen for us. In the examples from here on I’ll normally use the wrapper for convenience unless there is some specific initialization that needs doing or we are omitting some of the usual initialization steps.

Here is a minimal example using the wrapper:

Example 13b – Python wrapper example

```
import curses as cur

def main(win):
    win.addstr(4,4,"Hello world")
    win.refresh()
    win.getch()

cur.wrapper(main)
```

3 Output Functions

I guess you can't wait any more to see some action. Back to our odyssey through the world of curses functions. Now that curses is initialized, let's interact with the world.

There are four categories of method by which you can output to the screen.

1. `Window.addch([y,x],ch, [attr])` Print single character with attributes
2. `Window.addstr([y,x],string,[attr])` Print strings
3. `Window.insch([y,x],ch,[attr])` Insert a single character
4. `Window.insstr([y,x,str,,[attr])` Insert a string

Note: The C library contains yet another class `print()` which takes a format-string like the C `printf()` function. However, in Python we do string formatting on the the string object itself, so there is no need for the print family of functions in Python.

The `addXXX` family put the characters directly to the screen overwiting any existing text. The `insXXX` methods insert the characters, pushing existing text to the right. Any text pushed off the screen is lost (it does not wrap to the next line) and will not be restored if the inserted text is later deleted.

Also note once more that the coordinates are passed as y,x rather than the more usual x,y. This is just a foible of the curses library, you will get used to it eventually!

Let's see each one in detail.

3.1 `addch()` Method

This method puts a single character into the current cursor location and advances the position of the cursor. (Remember that in Python a character is simply a string of length 1, it is not a separate type as in C.) You provide the character to be printed. In practice `addch()` is most commonly used to print a character with some specific attributes.

Note: you cannot pass an empty string as the character.

Attributes are explained in detail later in the document. If a character is associated with an attribute (bold, reverse video etc.), then curses prints the character with that attribute.

In order to combine a character with some attributes, you have two options:

- Pass a bitwise OR of all the required attributes as a final argument to `win.addch()` (or `addstr()`) These attribute values can be found as defined constants in the `curses` module. For example, if you want to

print a character `ch` in bold and underlined, you would call `win.addch()` as below.

```
win.addch(ch, curses.A_BOLD | curses.A_UNDERLINE)
```

- Use a Window method like `attrset(), attron(), attroff().`

These methods are explained later in the Attributes section. Briefly, they manipulate the current attributes of the target window. Once set, the characters printed in the window are associated with the attributes until it is turned off.

Additionally, curses provides some special characters for character-based graphics. You can draw tables, horizontal or vertical lines, etc. You can find definitions for all the available characters in the curses module documentation. Try looking for names beginning with ACS. They are not available in code until after `initscr()` has been called. The ACS characters are discussed in more detail later in the document.

Note that `win.addch()` can also take an optional pair of y,x coordinates as its first two attributes, in which case the attribute is printed at the specified location within the window.

We are now familiar with the most basic output method `win.addch()`. However, if we want to print a string, it would be very annoying have to print it character by character. Fortunately, curses provides methods for outputting complete strings too.

3.2 `addstr()` Method

`addstr()` is used to put a character string into a given window. This method is similar to calling `addch()` once for each character in a given string. Another method in this family is `addnstr()`, which takes an integer `n` as its second argument. This method puts, at most, `n` characters into the screen. If `n` is negative, then the entire string will be added. Both variants can also accept an attribute value, which is applied to the whole string in the same way as discussed for `addch()` above.

Note: If the string cannot be fitted into the window the method will fail and nothing will be written.

Example 3: A simple addstr() example

```
import curses as cur
msg = "Just a string"

scr = cur.initscr()
rows,cols = scr.getmaxyx()    # get number of rows & columns

# print message at center of screen
scr.addstr(rows//2,(cols-len(msg))//2, msg)
```

```
# print message at bottom of screen
scr.addstr(rows-2, 0,
           "This screen has %d rows and %d columns\n" % (rows,cols)
           )
scr.addstr("Try resizing your window(if possible) and " +
           "then run this program again"
           )

scr.refresh()
scr.getch()
cur.endwin()
```

The above program demonstrates how easy it is to use `addstr()`. You just provide the coordinates and the message to be printed on the screen, then it does what you want. Notice that it takes account of any newline characters (`'\n'`) embedded in the string and responds accordingly.

3.3 insch() Method

This method is very similar to `addch()` above but in this case it pushes any existing text one position right. Any characters at the end of the line will be pushed off the window and lost.

3.4 insstr() Method

This is similar to `addstr()` above but again it pushes any existing text to the right. Any characters at the end of the line will be pushed off the window and lost.

There are other members of the `insXXX` group of methods that can be used for inserting lines. In this case the lines below are moved down and those at the bottom of the window will be lost. Consult the curses documentation for more details.

3.5 A Word of Caution

All these methods take y coordinate first and then x in their arguments. A common mistake by beginners is to pass x,y in that order. If you are doing too many manipulations of (y,x) coordinates, think of dividing the screen into windows and manipulate each one separately. Windows are explained later in the Windows section.

Another gotcha with curses is that if you move the cursor outside the window it will raise a `curses.error` exception. This means that any attempt to write a string in the last character position will result in an error as the cursor has nowhere to go after writing the characer! This warning also applies to creating new windows and subwindows, if they are too big or start in the

wrong position you will get an error. If you use the `curses.wrapper()` you won't see the error, you will simply find your program exiting prematurely!

4 Input functions

Printing without taking any input is pretty boring. Let's take a look at methods which allow us to get input from the user or sceen. These methods also can be divided into four categories.

1. `Window.getch([y,x])` Get a character, optionally from a location
2. `Window.getstr([y,x],[n])` Get a string, optionally from a location
3. `Windows.inch([y,x])` Read an existing character, from a location
4. `Windows.instr(y,x,[n])` Read an existing string from a location

Note that the return value from `getch()/inch()` is an integer and values above 255 represent special characters such as cursor movement keys or function keys etc. `getstr()/instr()` return a bytestring object and so may need to be decoded to a string in the usual Python manner. The `curses.ascii` module can be helpful in interpreting the integers as characters, but is not covered in this document.

4.1 getch() category of methods

These functions read a single character from the terminal but there are several subtle facts to consider. For example if you don't use the initialization function `cbreak()` (or `raw()` or `halfdelay()`), curses will not read your input characters as they are typed but will begin reading them only after a newline or an `EOF/EOT` is encountered. In order to avoid this, the `cbreak()` function must be used so that characters are immediately available to your program.

Another widely used function is `noecho()`. As the name suggests, when this function is set (used), the characters that are keyed in by the user will not show up on the screen. The two functions `cbreak()` and `noecho()` are typical examples of key management. Functions of this genre are explained in the key management section.

There is a closely related method called `getkey()` which reads a string of characters rather than a single character. This is useful for special keys such as function keys that return more than a single character per keystroke.

Note that there is a w`in.nodelay()` method which, when activated, causes the `getch()`/`getkey()` methods to return immediately and, if no key is pressed, give -1 as a return value for `getch()` or raise an exception for `getkey()`. (For those raised on BASIC programming this makes `getch()` act like the BASIC function `INKEYS$`) We will take a closer look at `getch()` in the section on interfacing with the keyboard.

Finally there is a method for reading so-called wide characters. `win.get_wch()` is the wide equivalent to `win.getch()`. We don't discuss the use of wide characters in detail in this document.

4.2 getstr() category of methods

These functions are used to get strings from the terminal. In essence, this function performs the same task as would be achieved by a series of calls to `win.getch()` until a newline, carriage return, or end-of-file is received. They return a bytes object which can be converted to a Python string by decoding in the usual way. `win.getstr()` does allow for some very basic editing of input prior to hitting return.

4.3 An Example

Example 4. A Simple getstr() example

```
import curses as cur

prompt ="Enter a string: "

scr = cur.initscr()
rows,cols = scr.getmaxyx()

scr.addstr(rows//2,(cols-len(prompt)) // 2, prompt)
bs = scr.getstr()  # read the user input as a bytestring
scr.addstr(cur.LINES-2, 0, "You entered: %s" % bs.decode('utf-8'))

scr.getch()
cur.endwin()
```

Note that `cur.LINES` is a constant defined in the curses module that holds the number of lines (or rows) in the current terminal. It is effectively the maximum height of a window and specifically the size of `stdscr.` There is another like it: `cur.COLS` that defines the current number of columns on the screen. In the code above we could have used `cur.LINES` and `cur.ROWS` instead of the call to `getmaxyx()` however, if dealing with windows other than `stdscr`, you need to use the function as demonstrated, since they may not be the same size as the terminal.

4.4 Reading from the Screen

In addition to gathering input from users curses also provides a couple of functions for reading characters from a window

1. `Window.inch(y,x)` read character (as an integer) from location
2. `Window.instr(y,x, [n])` read string (or n chars) from location

Note that the returned value from `inch()` is an integer in which the bottom 8 bits represent the character and the upper bits the attributes. The return value from `instr()` is a bytestring just as it was for `getstr()`.

Note also that `instr()` can take a length value, `n`, as an argument in which case it will return the next `n` characters. If `n` is longer than the space to the end of the line the method will fail.

You shouldn't need to use these methods very often but occasionally it is useful. We will see an example in the next section when we discuss how to get the current set of text attributes.

5 Attributes

We have already seen an example of how attributes can be used to print characters with some special effects.

Attributes, when set prudently, can present information in an easy, understandable manner. The following program takes a Python script as input and prints the file with comments in bold. If the file is too long to fit on a screen we pause until the user hits a key before starting a new page.

Example 5. A Simple Attributes example

```
import curses as cur
import sys

def main(fname):
    in_comment = False
    scr = cur.initscr()
    rows,cols = scr.getmaxyx()

    with open(fname) as fp:
        body = fp.read()   # read whole file

    for ch in body:    # read a char at a time
        y,x = scr.getyx()

        if (ch == '#') and not in_comment:
           in_comment = True
           scr.attron(cur.A_BOLD)  # make comments bold
        if ch == '\n' and in_comment:
            in_comment = False
            scr.attroff(cur.A_BOLD)  # end bold
        if y == rows - 1:   # reached last row
           scr.addstr("<-Press Any Key->")
           scr.getch()
           scr.clear()
           scr.move(0, 0)   # back to top of screen
           scr.addch(ch)
           scr.refresh()
           continue
        scr.addch(ch)
        scr.refresh()
    scr.addstr(cur.LINES-1,0, "Hit any key to exit")
    scr.getch()     # pause for exit
```

```
    cur.endwin()

if __name__ == "__main__":
   if len(sys.argv) != 2:
      print("Usage: %s <a script file name>\n" % sys.argv[0])
      sys.exit(1)
   name = sys.argv[1]
   main(name)
```

Don't worry about all the initialization and file handling stuff we have already seen. Concentrate on the `for` loop. It reads each character in the file and searches for the pattern '#'. Once it spots the pattern, it switches the BOLD attribute on with `attron()` . When we reach the end of a line after a comment it is switched off by `attroff()` .

The above program also introduces us to two useful functions `getyx()` and `move()`. The first function gets the coordinates of the present cursor into the variables y, x. The function `move()` moves the cursor to the coordinates given to it.

The above program is really a simple one which doesn't do much. However, by adding to these lines one could write a more useful program which reads a file, parses it and prints it in different colors just like a syntax-aware editor. One could even extend it to other languages as well.

5.1 The Details

Let's get into more details of attributes. The functions `attron(), attroff(), attrset()` can be used to switch attributes on/off , get attributes and produce a colorful display.

The functions `attron()` and `attroff()` take a bit-mask of attributes and switch them on or off, respectively. The following video attributes, which are defined in the curses module can be passed to these functions.

`A_NORMAL` Normal display (no highlight)

`A_STANDOUT` Best highlighting mode of the terminal.

`A_UNDERLINE` Underlining

`A_REVERSE` Reverse video

`A_BLINK` Blinking

`A_DIM` Half bright

`A_BOLD` Extra bright or bold

`A_PROTECT` Protected mode

`A_INVIS` Invisible or blank mode

`A_ALTCHARSET` Alternate character set

`A_CHARTEXT` Bitmask to extract a character

`color_pair(n)` Color-pair number n

The last one is the most colorful one :−) Colors are explained in the next section.

We can bitwise OR (`|`) any number of the above attributes to get a combined effect. If you wanted reverse video with blinking characters you can use

```
scr.attron(curses.A_REVERSE | curses.A_BLINK)
```

5.2 attron() vs attrset()

Then what is the difference between `attron()` and `attrset()`? `attrset()` sets all the background attributes of a window whereas `attron()` just switches on the attribute given to it. So `attrset()` fully overrides whatever attributes the window previously had and sets it to the new attribute(s). Similarly `attroff()` just switches off the attribute(s) given to it as an argument. This gives us the flexibility of managing attributes easily but if you use them carelessly you may lose track of what attributes the window has and garble the display. This is especially true while managing menus with colors and highlighting. So decide on a consistent policy and stick to it.

5.3 attr_get()

In C curses the function `wattr_get(win,...)` gets the current attributes and color pair of the window. Python does not provide this function, however the attributes of a given character can be read using `inch()`. (We cover the input fuctions later.) So we can create our own version of `attr_get()` like this:

```
def attr_get(win):
    y,x = win.getmaxyx() # how many lines in the window?
    win.insch(y-1,0,' ') # write a space at bottom left
    ch = win.inch(y-1,0) # read the char (including attributes)
    win.delch(y-1,0)     # remove the space again
    return ch
```

And use it like this:

```
atts = attr_get(scr)
if atts & cur.A_UNDERLINE:  # test for underline on with a bitwise AND
    scr.addstr("Underline is on")
else:
    scr.addstr("Underline is off")
```

There is an unfortunate bug in this code in that if a characer exists in the bottom right position of the screeen then it will be pushed off the screen by the call to `insch()` but when the character is later deleted with `delch()` the lost characer is not replaced. I leave it as an exercise for the reader to add code to check for that issue and take remedial action!

5.4 chgat() functions

The function `chgat()` can be used to set attributes for a group of characters without moving the cursor. It changes the attributes of a given number of characters starting at the current cursor location or from a given position.

We can give -1 as the character count to update to the end of the line. If you wanted to change the characters from the current position to the end of the line to reverse video, just use this.

```
scr.chgat(-1, A_REVERSE)
```

This function is useful when changing attributes for characters that are already on the screen. Move to the character that you want to change and change the attribute.

Example 6. `chgat()` Usage

```
import curses as cur

scr = cur.initscr()
cur.start_color()   # color handling is described later
cur.init_pair(1, cur.COLOR_BLACK,cur.COLOR_YELLOW)

# This uses the default attributes
scr.addstr("A big string which I didn't want to fully type.\n")
scr.refresh()           # make it visible
cur.napms(1000)         # pause 1 second
# Now call chgat to change the colors
# params 1,2 are the (optional) y,x position
# param 3 is (optional) length, -1 = EOL
# 4 is text attributes(color_pair converts colors to attributes)
scr.chgat(0,0,-1,cur.A_BOLD|cur.color_pair(1))
scr.refresh()           # see the change
cur.napms(1000)         # pause again
scr.chgat(cur.A_NORMAL) # use default values to restore
```

```
scr.refresh()
scr.getch()
cur.endwin()
```

This example also introduces us to the color world of curses. Colors will be explained in detail later. It also introduces the useful `curses.napms()` function which pauses (naps) for the given number of milliseconds. So `curses.napms(1000)` pauses for 1 second.

6 Windows

Windows form the most important concept in curses. You have seen the standard window `stdscr` above, where all the text handling functions were called as methods of this window. Now to design even the simplest GUI style program, you need to resort to windows. One reason you may want to use windows is to manipulate parts of the screen separately, for better efficiency, by updating only the areas that need to be changed. Another is for a better user experience, by making it easier to focus on a particular part of the screen.

I would say the last reason is the most important in going for windows. You should always strive for a clear and easy to manage design in your programs. If you are writing big, complex GUIs this is of pivotal importance before you start doing anything.

It is important to emphasize that curses windows are very different from OS GUI windows. They have no frame or title bar or any other adornments. They certainly don't respond to the mouse or any keyboard shortcuts unless you program them to do so. They are simply a defined area of screen which can have text inserted, be moved, resized and so on.

6.1 The Basics

A window can be created by calling the function `curses.newwin()`. It doesn't create anything on the screen. It creates an object in memory which you manipulate and so update its attributes like size, `beginy`, `beginx` etc. Hence, in curses, a window is just an abstraction of an area of screen, which can be manipulated independent of other parts of the screen.

Finally the window can be destroyed with `del(win)` in the usual Python style. It will delete the window object and free the memory.

6.2 Let there be a Window!!!

What fun is it, if a window is created and we can't see it? So the fun part begins by displaying the window. A call to our old friend the `refresh()` method achieves this but first we need to put some visible data, such as text or a border, into the window or we won't see any indication of its presence.

The method `window.box()` can be used to draw a border around the window. There are many other methods for manipulating window objects in curses. Let's explore these methods in more detail in this example.

Example 7. Window Border example

```
import curses as cur

def create_newwin(height, width, starty, startx):
    local_win = cur.newwin(height, width, starty, startx)
    local_win.box(0 , 0)    # 0, 0 gives default characters
    return local_win

def destroy_win(local_win):
    # we must remove the contents before deleting the object.
    # The parameters are
    # 1. ls: character to be used for the left side of the window
    # 2. rs: character to be used for the right side of the window
    # 3. ts: character to be used for the top side of the window
    # 4. bs: character to be used for the bottom side of the window
    # 5. tl: character to be used for the top left corner of the window
    # 6. tr: character to be used for the top right corner of the window
    # 7. bl: character to be used for the bottom left corner of the window
    # 8. br: character to be used for the bottom right corner of the window
    local_win.border(' ', ' ', ' ',' ',' ',' ',' ',' ')
    local_win.refresh() # remove visible contents before deleting object
    del(local_win)      # delete the object


def main(scr):
     cur.curs_set(0)        # turn off the cursor
     height = 3
     width = 10
     starty = (cur.LINES - height) // 2   # Calculate center
     startx = (cur.COLS - width) // 2     # of the window

     scr.addstr("Press Q to exit (F1 for help)")
     scr.refresh();

     my_win = create_newwin(height, width, starty, startx)
     my_win.refresh()

     while True:     # create an event loop
         ch = scr.getch()
         if ch in (ord('Q'), ord('q')):
             break   # exit the event loop
         elif ch == cur.KEY_F1:
             scr.addstr(0,0, "Use arrow keys to move box, Q to exit")
             scr.refresh()
         elif ch == cur.KEY_LEFT:
             startx -= 1
             destroy_win(my_win)
```

```
            my_win = create_newwin(height, width, starty, startx)
            my_win.refresh()
        elif ch == cur.KEY_RIGHT:
            startx += 1
            destroy_win(my_win)
            my_win = create_newwin(height, width, starty, startx)
            my_win.refresh()
        elif ch == cur.KEY_UP:
            starty -= 1
            destroy_win(my_win)
            my_win = create_newwin(height, width, starty, startx)
            my_win.refresh()
        elif ch == cur.KEY_DOWN:
            starty += 1
            destroy_win(my_win)
            my_win = create_newwin(height, width, starty, startx)
            my_win.refresh()
        elif ch == ord('h'):
            my_win.addstr(1,2,"Hello!")  # 1,2, to centre text
            my_win.refresh()

    destroy_win(my_win)

cur.wrapper(main)
```

6.3 Explanation

Don't scream. I know it's a big example but I have to explain some important things here :-). This program creates a rectangular window that can be moved with left, right, up, down arrow keys. It repeatedly creates and destroys windows as the user presses keys. Let's dissect it line by line.

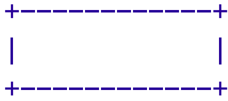
The `create_newwin()` function creates a window with `newwin()` and displays a border around it with `box()`.

The function `destroy_win()` first erases the window from screen by painting a border with ' ' characters and then calling `del(win)` to delete the object.

As you can see, I used `border()` instead of `box()`. This is because `box()` only takes the top and side characters, you cannot specify the corners which are always the defaults. Thus the window would not be wholly deleted. `border()` draws a border around the window with the characters given to it as the 4 lines and 4 corner points so we can delete the entire border. To give an example, if you have called `border()` as below:

```
local_win.border(, '|', '|', '-', '-', '+', '+', '+', '+')
```

it produces some thing like:



Finally we need to refresh the window to make the border dissappear and then, and only then, can we delete the window object.

Note that the curses.wrapper() function calls various initialization functions to ensure that a) all the keys are passed to the program (`cbreak`) and b) that they do not display (`noecho`) on screen and c) the cursor is hidden (`curs_set`). We then create the new window `local_win` , display it with its own `refresh()` (it's not part of `stdscr`, it is an independant window).

Next an event loop (`while`) is started using `getch()` to read key-presses. Depending on the key the user presses the appropriate code is executed. We either move `local_win` by modifying `starty` or `startx` and deleting the existing window before creating a new window at the new location. Or we can display some text (hitting 'h') using `local_win`'s version of `addstr()`, using `local_win`'s coordinates rather than those of `stdscr`. Or we can display some help text (by hitting F1)

6.4 Some Other Stuff in the Example

You can also see in the above examples, that I have used the variables `curses.COLS` and `curses.LINES` which are initialized to the terminal size by `initscr()`. They can be useful in finding screen dimensions or finding the center coordinate of the screen as used above. The function win.`getch()` ,as usual, reads a key from the keyboard (although in this case we actually store it in the `ch` variable whereas previously we just ignored it!) and depending on the key pressed, the program does the corresponding work. This type of `while` loop combined with an `if/elif` ladder is very common in any GUI style programs although usually hidden in the depths of a GUI framework like Tkinter.

6.5 Other Border Functions

The above example is grossly inefficient in that, with each press of a key, a window is destroyed and another is created. So let's write a more efficient program which uses other window-related functions.

The following program uses the `clear()` and `mvwin()` window methods to achieve a similar effect. These two functions are simple. `clear()` deletes everything in the window (including the border) and `mvwin()` moves the window to a new location (but does not draw it). The only problem is that `mvwin()` fails if the window moves off the visible screen. So we need to check and, if necessary, modify the coordinates before calling it.

The following example is very similar to the previous one except we don't delete/create windows but create a new function to clear and move the existing window using the curses window method `mvwin()`.

Example 8. More border functions

```
import curses as cur

def move_window(win, y, x):
    win.clear()
    win.refresh()
    win.mvwin(y,x)
    win.box(0,0)

def main(scr):
    cur.curs_set(0)        # turn off the cursor
    scr.refresh()          # show initial display

    # create new window
    height, width = 3,10
    starty = (cur.LINES - height) // 2   # Calculate center
    startx = (cur.COLS - width) // 2     # of the window
    my_win = cur.newwin(height, width, starty, startx)
    my_win.box(0 , 0)    # 0, 0 gives default characters
    my_win.refresh()     # make it visible

    # Run event loop
    ch = scr.getch()
    while True:     # create an event loop
        if ch in (ord('Q'), ord('q')):
            break   # exit the event loop
        elif ch == cur.KEY_F1:
            scr.addstr(0,0, "Use arrow keys to move box, Q to exit")
            scr.refresh()
        elif ch == cur.KEY_LEFT:
            startx -= 1
            if startx < 0:                 # reached left edge
               startx = cur.COLS-width     # go to opposite side
            move_window(my_win,starty,startx)
            my_win.refresh()
        elif ch == cur.KEY_RIGHT:
            startx += 1
            if startx > cur.COLS-width-1:   # reached right edge
               startx = 0                   # go to opposite side
            move_window(my_win,starty,startx)
            my_win.refresh()
        elif ch == cur.KEY_UP:
```

```
            starty -= 1
            if starty < 0:                   # reached top
               starty = cur.LINES-height-1   # go to opposite side
            move_window(my_win,starty,startx)
            my_win.refresh()
        elif ch == cur.KEY_DOWN:
            starty += 1
            if starty > cur.LINES-height-1:   # reached bottom
               starty = 0                     # go to opposite side
            move_window(my_win,starty,startx)
            my_win.refresh()
        elif ch == ord('h'):
            my_win.addstr(1,2,"Hello!")       # 1,2, to centre text
            my_win.refresh()
        ch = scr.getch()

cur.wrapper(main)
```

Note that there are various other things you can do with windows, including set the background color or pattern, using `window.bkgd()` and `window.bkgdset()`. We will see these later when we discuss colors.

6.6 Sub-Windows

Curses allows us to create sub-windows inside other windows. These can best be thought of as panes within a larger window into which you can insert text etc. When you move a window all of its sub-windows move with it. When you clear a window you clear its sub-windows too.

It is crucially important to realize that a sub-window is just a view onto the parent window. When you write into a sub window you are also writing onto the parent since they share a common model of the display. Equally if you write to a part of the parent covered by a sub-window the text will show inside the sub-window. This is what distinguishes sub-windows from new windows. When you call `newwin()` you get a completely new, independent window that owns its own display area. If a window underneath it writes text it does not show on the new window. It will be invisible until you delete or move the new window. With a sub window the text is shared and visible immediately.

Closely related to sub-windows are derived windows. The difference relates to the coordinates used: screen based for sub windows, parent-window based for derived windows. Personally, I prefer derived windows since they simplify the calculation of coordinates.

The following example illustrates all of the various window types in action.

Example 8a Manipulating sub-windows

```
import curses as cur

def show_status(win,txt):
    # display text in bottom line of screen
    ht,wd = win.getmaxyx()
    win.addstr(ht-1,1, txt)
    win.clrtoeol()
    win.refresh()

def main(scr):
    # create the main window
    win = cur.newwin(16,50,4,4)
    win.box(0,0)
    win.addstr(4,7,"Main window")
    show_status(scr, “Created main window”)
    win.refresh()
    scr.getch()

    # add sub window of main
    sw = win.subwin(4,18, 6,9)    # use screen coords
    sw.box(0,0)
    sw.addstr(1,2, "Sub window")  # use sub win coords
    sw.refresh()
    show_status(scr, "Sub window creates a view onto main window")
    scr.getch()

    # add derived window of main
    dw = win.derwin(4,18, 6,9)    # use mainwin coords
    dw.box(0,0)
    dw.addstr(1,2,"Derived window")  # use derived win coords
    dw.refresh()
    show_status(scr,"Derived window is the same but uses relative coords")
    scr.getch()

    # add new window on top of main
    nw = cur.newwin(4,18,15,6)
    nw.box(0,0)
    nw.addstr(1,2,"New window")
    nw.refresh()
    show_status(scr,"New window on top of main window")
    scr.getch()

    # move main window, including sub windows
    scr.touchwin()   # remove old window from screen
    scr.refresh()
```

```
    win.mvwin(2,10)
    win.refresh()    # draw at new position, including subwins
    nw.touchwin()
    nw.refresh()     # but new window stays where it was
    show_status(scr,"Move main window with sub windows, newwin is not affected")
    scr.getch()

    # clear main window - includes subwindow and border
    win.clear()
    win.refresh()
    nw.touchwin()
    nw.refresh()     # new window stays as it was
    show_status(scr,"Clear main window text and all borders, newwin untouched")
    scr.getch()

    # redraw just the borders
    win.box(0,0)
    sw.box(0,0)
    dw.box(0,0)
    win.refresh()
    nw.touchwin()
    nw.refresh()     # bring new window to top

    show_status(scr, "Redrawing borders shows window objects still exist")
    scr.getch()

cur.wrapper(main)
```

7 Colors

7.1 The Basics

Life seems dull with no colors. Curses has a nice mechanism to handle colors. Let's get into the thick of the things with a small program.

Example 9. A Simple Color example

```
import curses as cur

def print_in_middle(win, aString):
    y,x = win.getmaxyx()
    y //= 2
    x = (x - len(aString)) // 2
    win.addstr(y,x, aString)
    win.refresh()

scr = cur.initscr()
if not cur.has_colors():   # Can we do color?
   cur.endwin()
   print("Your terminal does not support color")
   raise SystemExit(1)
else:
   cur.start_color()      # initialize color data structures
   scr.addstr("Your terminal supports color.\n")
   scr.addstr("You can define up to %d color pairs" % (cur.COLOR_PAIRS-1))
   scr.refresh()
   cur.napms(1000)

cur.init_pair(1, cur.COLOR_RED, cur.COLOR_BLACK)
scr.attron(cur.A_BOLD | cur.color_pair(1))   # apply color scheme
print_in_middle(scr, "Viola !!! In color ...\n")
cur.napms(1000)                  # pause 1s

scr.attroff(cur.color_pair(1))   # turn color off
print_in_middle(scr, "Booo!!! In color no more!")
scr.getch()
cur.endwin()
```

As you can see, to start using color, you should first check whether the terminal supports color using `has_color()`. If it does you can call the function

start_color() to initialize the color handling data structures. After that, you can use color capabilities of your terminals using various functions.

Curses initializes all the colors supported by the terminal when start_color() is called. These can be accessed by the defined constants like COLOR_BLACK etc. Now, to actually start using colors, you have to define pairs.

Colors are always used in pairs. That means you have to use the function init_pair() to define the foreground and background for the pair number you give. The first pair color_pair(0) is always white on black, the other pairs are free for you to define as you wish. You can define up to COLOR_PAIRS-1 combinations but that value will not exist until after you call start_color()! COLOR_PAIRS is terminal dependant. For example on my Gnome Terminal application it is 256 but on a vanilla xterm it is only 64.

After initialization the pair number can be used as a normal attribute when combined with the color_pair() function. In our example we combined it with the A_BOLD text attribute using a bitwise or (|). This may seem to be cumbersome at first but this elegant solution allows us to manage color pairs very easily. To appreciate it, you should look into the the source code of "dialog"; a utility for displaying dialog boxes from shell scripts (we cover the Python version of dialog in a later section). The developers have defined foreground and background combinations for all the colors they might need and initialized them at the beginning. This makes it very easy to set attributes just by accessing a pair which we have already defined as a constant.

The following colors are defined in the curses module. You can use these as parameters for various color functions.

```
COLOR_BLACK 0
COLOR_RED 1
COLOR_GREEN 2
COLOR_YELLOW 3
COLOR_BLUE 4
COLOR_MAGENTA 5
COLOR_CYAN 6
COLOR_WHITE 7
```

Note that the Python wrapper() function calls start_color() for you.

7.2 Changing Color Definitions

The function curses.init_color() can be used to change the RGB values for the colors defined by curses initially.

Say you wanted to lighten the intensity of the COLOR_RED color by a smidge (or even change it to something completely different like brown!). Then you can use this function as

```
cur.init_color(COLOR_RED, 700, 0, 0)
# param 1 : color name
```

```
# param 2, 3, 4 : RGB content min = 0, max = 1000
```

Note that the RGB (Red,Green,Blue) values are specified from 1-1000 not the more common 0-255 used in HTML/CSS etc.

Note too that if you call this mid-way through a session any existing text in the specified color is automatically changed to the new color and the screen refreshed to display it.

If your terminal cannot change the color definitions, the function returns ERR. The function `curses.can_change_color()` can be used to find out whether the terminal has the capability of changing color content or not.

7.3 Color Content

The functions `curses.color_content()` and `curses.pair_content()` can be used to find the color content and foreground-background combination for a pair. `color_content()` returns the RGB values for a given color number (eg. `COLOR_RED`) while `pair_content()` returns the foreground,background pair of color numbers (eg. `COLOR_GREEN,COLOR_BLACK`) for a given color pair.

8 Interfacing with the Keyboard

No GUI is complete without a strong user interface and to interact with the user, a curses program should be sensitive to key presses or the mouse actions performed by the user. Let's deal with the keys first.

8.1 The Basics

As you have seen in almost all of the above examples, it's very easy to get key input from the user. A simple way of getting key presses is to use window object's `getch()` method. The `cbreak()` mode should be enabled to read keys when you are interested in reading individual key hits rather than complete lines of text (which usually end with a carriage return). `keypad()` should be enabled to get the Function keys, Arrow keys etc. See the initialization section for details.

`getch()` returns an integer corresponding to the key pressed. If it is a text character, the integer value will be equivalent to the ordinal of the character. Otherwise it returns a number which can be matched with the constants defined in the curses module. For example, if the user presses F1, the integer returned is 265. This can be checked using the constant `KEY_F1` defined in `curses`. This makes reading keys portable and easy to manage.

When you call `getch()`, `it` will wait for the user to press a key, (unless you specified a timeout using `curses.halfdelay()`) then, when the user presses a key, the corresponding integer is returned. You can then check the value returned against the constants defined in `curses` to match against the keys you want. If you want to compare to a text character then you will need to get the integer equivalent using the regular Python function `ord()`. (You can also use the `curses.ascii` module features but, in most cases, `ord()` is simpler)

The following code snippet demonstrates the principle.

```
ch = scr.getch()
if ch == cur.KEY_LEFT:
    scr.addstr("Left arrow is pressed\n")
```

You saw this earlier, in Examples 7 and 8, which read the arrow keys and moved a window on the screen.

Let's write a small program which creates a menu which can be navigated by up and down arrows.

8.2 Simple Key Usage

Example 10. A Simple Key Usage example

```
import curses as cur

WIDTH=30
HEIGHT=10

choices = [
"Choice 1",
"Choice 2",
"Choice 3",
"Choice 4",
"Exit",
]
n_choices = len(choices)
highlight = 0

def print_menu(m_win, hlight):
    global highlight
    x = 2
    y = 2
    m_win.box(0, 0)
    for n,choice in enumerate(choices,1):
       if highlight == n:       # Highlight the present choice
           m_win.attron(cur.A_REVERSE)
           m_win.addstr(y, x, choice)
           m_win.attroff(cur.A_REVERSE)
       else:
           m_win.addstr(y, x, choice)
       y += 1
    m_win.refresh()


def main(scr):
    scr.clear()

    #initialize application data
    startx = (80 - WIDTH) // 2
    starty = (24 - HEIGHT) // 2
    highlight = 1
    choice = 0
    choice_fmt = "You chose choice %d with choice string %s\n"
```

```
    # initialize menu window
    menu_win = cur.newwin(HEIGHT, WIDTH, starty, startx)
    menu_win.keypad(True)
    scr.addstr(0, 0, "Use arrow keys to go up and down. ”)
    scr.addstr(“Press Enter to select a choice")
    scr.refresh()
    print_menu(menu_win, highlight)

    # event loop
    while True:
        c = menu_win.getch()
        if c == cur.KEY_UP:
           if highlight == 1:
              highlight = n_choices
           else: highlight -= 1
        elif c == cur.KEY_DOWN:
           if highlight == n_choices:
              highlight = 1
           else: highlight += 1
        elif c in (KEY_RETURN, cur.KEY_ENTER):
           choice = highlight
           scr.addstr(cur.LINES-2, 0,
                      choice_fmt % (choice, choices[choice - 1]))
        else:
           scr.addstr(cur.LINES-1, 0,
                      "Character pressed: %3d" % c)

        print_menu(menu_win, highlight);
        if choice == n_choices:  # User chose to come out of the menu
           scr.addstr(cur.LINES-1, 0, "Sorry to see you go")
           scr.clrtoeol()
           break
        scr.refresh()
    scr.getch()   # pause a moment

cur.wrapper(main)
```

Notice the menu wrapping code that ensures the highlighting wraps around at the ends of the menu movement. Notice also that we test for key value `10` (defined locally as `KEY_RETURN)` as well as the symbolic value `KEY_ENTER`. This is because in most modern terminals the ENTER key is actually mapped to the enter key on the numeric keypad. Key code `10` (the ‘`\n`’ new line) is for the more commonly used “Carriage Return” key. The final point to note is that after printing the final farewell we call `clrtoeol()` just to remove any characters lingering from previous messages on that line.

9 Interfacing with the Mouse

Now that you have seen how to get keys, let's do the same thing from the mouse. Usually each UI allows the user to interact with both keyboard and mouse.

9.1 The Basics

Before you can do anything with the mouse you must enable the keypad feature using `window.keypad(True)` (previously used to detect Function keys) and the events that you want to receive have to be enabled with

```
curses.mousemask(mask) -> (available, oldmask)
```

The parameter is a bit-mask of the mouse events you would like to detect. By default, all the events are turned off. The bit mask `ALL_MOUSE_EVENTS` can be used to get all the events and is the most common use-case. The return value is a tuple of the currently available mouse mask and the previous mask. (This allows you to restore previous behaviour in the case where you only want a temporary change)

The following are all the event masks defined in `curses` (notice that there are no mouse movement events, only button operations):

Name	**Description**
`BUTTON1_PRESSED`	mouse button 1 down
`BUTTON1_RELEASED`	mouse button 1 up
`BUTTON1_CLICKED`	mouse button 1 clicked
`BUTTON1_DOUBLE_CLICKED`	mouse button 1 double clicked
`BUTTON1_TRIPLE_CLICKED`	mouse button 1 triple clicked
`BUTTON2_PRESSED`	mouse button 2 down
`BUTTON2_RELEASED`	mouse button 2 up
`BUTTON2_CLICKED`	mouse button 2 clicked
`BUTTON2_DOUBLE_CLICKED`	mouse button 2 double clicked
`BUTTON2_TRIPLE_CLICKED`	mouse button 2 triple clicked
`BUTTON3_PRESSED`	mouse button 3 down

Name	Description
BUTTON3_CLICKED	mouse button 3 clicked
BUTTON3_DOUBLE_CLICKED	mouse button 3 double clicked
BUTTON3_TRIPLE_CLICKED	mouse button 3 triple clicked
BUTTON4_PRESSED	mouse button 4 down
BUTTON4_RELEASED	mouse button 4 up
BUTTON4_CLICKED	mouse button 4 clicked
BUTTON4_DOUBLE_CLICKED	mouse button 4 double clicked
BUTTON4_TRIPLE_CLICKED	mouse button 4 triple clicked
BUTTON_SHIFT	shift was down during button state change
BUTTON_CTRL	control was down during button state change
BUTTON_ALT	alt was down during button state change
ALL_MOUSE_EVENTS	report all button state changes
REPORT_MOUSE_POSITION	report mouse movement

9.2 Getting Events

Once a class of mouse events have been enabled, the `getch()` method returns `KEY_MOUSE` every time some mouse event happens. Then the mouse event can be retrieved with `curses.getmouse()`.

The code approximately looks like this:

```
ch = win.getch()
if ch == cur.KEY_MOUSE:
   mouse_event = cur.getmouse()
   if mouse_event[4] == cur.BUTTON1_CLICKED:
     . # Do some thing with the BUTTON1 event
```

`getmouse()` returns the event as a 5-tuple: `(Id,X,Y,Z,bstate)`

The `bstate` is the main value we are interested in. It tells us the button state of the mouse – which button was pressed. The `Id` is for handling multiple input devices and the `Z` coordinate is not currently used. `X,Y` give the screen coordinates where the event occured. (Note that the mouse event returns X before Y unlike most curses functions)

A minimal mouse enabled program looks like:

```
import curses as cur
# define mouse event indices
BSTATE = 4

def main(scr):
    # initialize mouse handling
    msk,_ = cur.mousemask(cur.ALL_MOUSE_EVENTS)

    scr.addstr("Click the mouse and see what happens, Q to exit")
    scr.refresh()
    count = 0

     # start event loop
    while True:
       ch = scr.getch()
       if ch in ( ord('q'),ord('Q') ):
          break
       if ch == cur.KEY_MOUSE:   # Handle mouse events
          mev = cur.getmouse()
          count += 1
          scr.addstr(2,0, "Mouse event %d detected" % count)
          if mev[BSTATE] & cur.BUTTON1_CLICKED:
             scr.addstr(20,0,"\nPressed button 1")
             scr.refresh()

cur.wrapper(main)
```

9.3 Putting it all Together

That's pretty much all there is to interfacing with a mouse. Let's create the same menu and enable mouse interaction. To make things simpler, key handling is removed.

Example 11. Access the menu with mouse!

```
import curses as cur
# define global constants
X = 1
Y = 2
BSTATE = -1
WIDTH=30
HEIGHT=10
choices = [
    "Choice 1",
    "Choice 2",
    "Choice 3",
    "Choice 4",
    "Exit",
]
n_choices = len(choices)

def print_menu(m_win):
    x = 2
    y = 2
    m_win.box(0, 0)
    for n,choice in enumerate(choices):
        m_win.addstr(y+n, x, choice)
    m_win.refresh()

# determine if mouse click is inside window
def in_window(win, event):
    y,x = win.getbegyx()  # get window origin
    h,w = win.getmaxyx()  # get window height/width
    return (event[Y] >= y + 2 and
        event[Y] <= y + n_choices + 1 and
        event[X] >= x + 2 and
        event[X] <= x + w -1)

# get the choices index the user clicked on
def report_choice(win,event):
    y,x = win.getbegyx()
    y += 2              # account for borders & margins

    for n,item in enumerate(choices):
       if event[Y]-y == n:          # we pressed on this item
           break;
    if n == n_choices-1: return -1
    else: return n

def main(scr):
    cur.curs_set(0)  # make cursor invisible
```

```
    scr.clear()

    # Try to put the window in the middle of screen
    choice = 0
    choice_fmt = "Choice made is : %d String Chosen is '%10s'"
    startx = (cur.COLS - WIDTH) // 2
    starty = (cur.LINES - HEIGHT) // 2
    scr.addstr(cur.LINES-1, 1, "Click on Exit to quit")
    scr.refresh()

    # Print the menu for the first time
    menu_win = cur.newwin(HEIGHT, WIDTH, starty, startx);
    menu_win.keypad(True)          # to receive mouse events inside window
    print_menu(menu_win)

    # Get all the mouse events
    cur.mousemask(cur.ALL_MOUSE_EVENTS)

    while True:
        c = menu_win.getch()
        if c in [ord('q'),ord('Q')]:
           break
        if c == cur.KEY_MOUSE:
           event = cur.getmouse()
           # When the user clicks left mouse button in the menu box
           if ( (event[BSTATE] & cur.BUTTON1_CLICKED) and
                 in_window(menu_win,event) ):
              choice = report_choice(menu_win, event)
              if choice == -1:   # Exit chosen
                 break
              else:
                  scr.addstr(cur.LINES-5, 1,
                             choice_fmt % (choice, choices[choice]))
                  scr.refresh()
        print_menu(menu_win)

cur.endwin()
```

I'll leave it as an exercise for the reader to combine examples 10 and 11 to handle both mouse and key navigation plus text attributes.

9.4 Miscellaneous Functions

The `curses.mouseinterval()` function sets the maximum time (in thousands of a second) that can elapse between press and release events in order for them to be recognized as a click. This function returns the previous interval value. The default is one fifth of a second.

There is also a function to push a mouse event onto the event queue:

```
curses.ungetmouse(id,x,y,z,bstate)
```

This can be used to push back an existing event or even to push a new artificial mouse event onto the queue for later processing. These are rarely used in practice but occasionally can be useful.

10 Screen Manipulation

In this section, we will look into some methods, which allow us to manage the screen efficiently and to write some fancy programs. This is especially important in writing games.

10.1 getyx() functions

The window method `getyx()` can be used to find out the present cursor coordinates within the window. That is, the `y,x` values returned are relative coordinates not screen coordinates.

The `getparyx()` method gets the beginning coordinates of the sub window relative to its parent window. This is sometimes useful to update a sub-window. When designing fancy stuff like writing multiple menus, it becomes difficult to store the menu positions, their first option coordinates etc. A possible solution to this problem, is to create menus in sub-windows and later find the starting coordinates of the menus by using `getparyx()`.

The `getbegyx()` and `getmaxyx()` methods store the current window's beginning and maximum coordinates. You saw those used in example 11 for the `in_window()` function.

These methods are useful in the same way as above in managing windows and sub windows effectively.

10.2 Screen Dumping

While writing games, some times it becomes necessary to store the state of the screen and restore it back to the same state. In C curses there is a function, `scr_dump()` that can be used to dump the screen contents to a file given as an argument. Later it can be restored by a `scr_restore()` function. Unfortunately these functions are not present in the Python implementation of curses, so we must do some extra work.

10.3 Window Dumping

To store and restore windows, the methods `window.putwin(binfile)` and `curses.getwin(binfile)` can be used. `putwin()` puts the present window state into a file, opened in binary write mode, which can be later restored by getwin() using the same binary file in read mode. The method returns a window object, just like calling `curses.newwin()`

Example 11b Save/restore screen demonstration

```
import curses as cur
import os

def save_windows(winlist, path="/tmp"):
    for num,win in enumerate(winlist):
        fname = path+"/win"+str(num)+".cur"
        with open(fname,'wb') as f:
          win.putwin(f)

def restore_windows(path="/tmp"):
    files = [f for f in os.listdir(path) if f.endswith('.cur')]
    files.sort()
    windows = []
    for f in files:
        fn = os.path.join(path,f)
        win = cur.getwin(open(fn, 'rb'))
        windows.append(win)
    return windows

def main(scr):
    # set up screen and initialise colors
    cur.init_pair(1, cur.COLOR_BLUE,cur.COLOR_YELLOW)
    cur.init_pair(2, cur.COLOR_WHITE, cur.COLOR_BLUE)
    cur.init_pair(3, cur.COLOR_BLACK, cur.COLOR_RED)
    scr.bkgd(' ', cur.color_pair(1))
    scr.refresh()

    # create 2 new colored windows
    win1 = cur.newwin(3,10,cur.LINES//4,cur.COLS//4)
    win1.bkgd(' ',cur.color_pair(2))
    win1.box(0,0)
    win1.refresh()

    win2 = cur.newwin(3,10,cur.LINES//4,(cur.COLS//4)+30)
    win2.bkgd('.',cur.color_pair(3))
    win2.box(0,0)
    win2.refresh()

    # save this screen configuration as our "home screen"
    save_windows([scr,win1,win2])

    # now clear screen and reset colors
    scr.addstr(cur.LINES-2,1, "Hit a key to clear...")
    scr.getch()
    scr.clear()
    scr.bkgd(' ',cur.A_NORMAL)
```

```
    scr.refresh()
    scr.getch()

    # now restore the old screen
    wins = restore_windows()
    for win in wins:
        win.refresh()

    wins[0].addstr(20,2,"Hit enter to exit...")
    wins[0].refresh()

    scr.getch()

cur.wrapper(main)
```

Note that we created two new functions to save and restore the windows and these used the `curses` module functions to do the work. We pass the list of windows to be saved (in the order in which they must be restored) and an optional path argument. The restore function reads all of the window files in the path and returns them as a list of window objects. We can then refresh each object to recreate the original screen at the time of saving. This is, of course, a simplistic example relying on the numbering of the windows to control sequence. That would fail if there were more than 10 windows, in which case a more robust file naming scheme would need to be designed. Similarly using `/tmp` as a save folder is only useful for short term storage, a dedicated project folder would be safer.

Note that this facility only saves and restores the screen not the application data. You need to recreate the data yourself, possibly by saving/restoring them using standard Python techniques such as the `pickle` or `shelve` modules or a database.

11 Miscellaneous features

Now you know enough features to write a good curses program, with all bells and whistles. There are some miscellaneous functions which are useful in various cases. Let's go headlong into some of those.

11.1 curs_set()

This function can be used to make the cursor invisible. The parameter to this function should be `0` : invisible or `1` : normal or `2` : very visible (usually bold). We saw this function being used back in Examples 7,8 and 11.

11.2 Temporarily Leaving Curses Mode

Some times you may want to get back to “cooked mode” (normal line buffering mode) temporarily. In such a case you will first need to save the tty modes with a call to `def_prog_mode()` and then call `endwin()` to end the curses mode. This will leave you in the original tty mode. To get back to curses once you are done, call `reset_prog_mode()` . This function returns the tty to the state stored by `def_prog_mode()`. Then do `refresh()`, and you are back to the curses mode. Here is an example showing the sequence of things to be done.

Example 12. Temporarily Leaving Curses Mode

```
import curses as cur
import os,time

def main(scr):
    scr.addstr("Hit return to leave curses", cur.A_REVERSE)
    scr.refresh()
    scr.getch()

    cur.def_prog_mode()     # Save the tty modes
    cur.endwin()            # End curses mode temporarily
    # Use the normal python commands for input/output etc
    os.system("clear; stty echo -nl")   # ensure terminal restored fully
    print ("Back in terminal mode")
    time.sleep(1)
    input("Hit return to go back to curses")

    cur.reset_prog_mode()   # Return to the previous tty mode
    scr.refresh()           # restore the Screen contents
    scr.addstr(1,1,"Welcome back!", cur.A_UNDERLINE)
    scr.addstr(22,0,"Hit a key to exit...")
    scr.refresh()
```

```
    scr.getch()

cur.wrapper(main)
```

11.3 ACS_ Variables

If you have ever programmed in DOS, you know about those nifty characters in the extended character set. They are printable only on some terminals. Curses functions like `box()` use these characters. All these variables start with ACS meaning Alternative Character Set. You might have noticed me using these characters in some of the programs above. Here's an example showing some of the ACS characters. The full list is detailed in the Python curses documentation.

Example 13. ACS Variables Example

```
import curses as cur

scr = cur.initscr()
scr.addstr("Upper left corner \t")
scr.addch(cur.ACS_ULCORNER)
scr.addstr("\nLower left corner \t")
scr.addch(cur.ACS_LLCORNER)
scr.addstr("\nLower right corner \t")
scr.addch(cur.ACS_LRCORNER)
scr.addstr("\nTee pointing right \t")
scr.addch(cur.ACS_LTEE)
scr.addstr("\nTee pointing left \t")
scr.addch(cur.ACS_RTEE)
scr.addstr("\nTee pointing up \t")
scr.addch(cur.ACS_BTEE)

scr.getch()
cur.endwin()
```

11.4 And finally...

There are a number of other functions and methods in the `curses` module. Many of them are used to find out information about the terminal capabilities by querying the terminfo database. Others handle so called “wide characters”. Still others control graphics and the refresh mechanism. These are outside the scope of this tutorial but are described in the `curses` module documentation.

12 curses.panel module

Now that you are proficient in curses, you wanted to do some thing big. You created a lot of overlapping windows to give a professional windows-type look. Unfortunately, it soon becomes difficult to manage these.

The multiple refreshes and updates plunge you into a nightmare. The overlapping windows create blotches, whenever you forget to refresh the windows in the proper order. Don't despair. There's an elegant solution provided in the panels module. In the words of developers of ncurses

> *When your interface design is such that windows may dive deeper into the visibility stack or pop to the top at runtime, the resulting book-keeping can be tedious and difficult to get right.*

Hence the panel module.

If you have lot of overlapping windows, then the panel module is the way to go. It obviates the need of doing a series of `noutrefresh() and doupdate()` (we haven't discussed these but essentially they provide a deferred refresh) and relieves the burden of doing it correctly (bottom up). The library maintains information about the order of windows, their overlapping and it updates the screen properly. So why wait? Let's take a close peek into `curses.panel`.

12.1 The Basics

A panel object is a window that is implicitly treated as part of a deck including all other panel objects. The deck is treated as a stack with the top panel being completely visible and the other panels may or may not be obscured according to their positions. So the basic idea is to create a stack of overlapping panels and use the `panel` module to display them correctly. There is a method similar to `refresh()` which, when called , displays panels in the correct order. Methods are provided to hide or show panels, move panels, change size etc. The overlapping problem is managed by the panels library during all the calls to these methods.

The general flow of a panel program goes like this:

1. Create the windows (with `curses.newwin()`) to be attached to the panels.
2. Create panels with the chosen visibility order. Stack them up according to the desired visibility. The function `panel.new_panel()` is used to created panels.
3. Call `panel.update_panels()` to write the panels to the virtual screen in correct visibility order. Do a `window.doupdate()` to show it on the screen.

4. Manipulate the panels with `panel.show()`, `panel.hide()`, `panel.move()` etc. Make use of helper methods like `panel.hidden()` and `panel.window()`. Make use of “user pointer” to store custom data for a panel. Use the methods `panel.set_userptr(object)` and `panel.userptr()` to set and get the user pointer for a panel. (A user pointer is a reference to an object, which can be any arbitrary Python object type. The name is a throw-back to curses C origins.)
5. When you are done with the panel use `del()` to delete the panel.

Let's make the concepts clear with some programs. The following is a simple program which creates 3 overlapping panels and shows them on the screen.

Example 14. Panel basics

```
import curses as cur
import curses.panel as pan

def main(scr):
    lines = 10
    cols = 40
    y = 2
    x = 4
    my_wins = []
    my_panels = []

    cur.start_color()
    cur.init_pair(1, cur.COLOR_WHITE,cur.COLOR_RED)
    cur.init_pair(2, cur.COLOR_BLUE,cur.COLOR_YELLOW)
    cur.init_pair(3, cur.COLOR_BLACK,cur.COLOR_GREEN)

    # Create overlapping windows for the panels
    my_wins.append(cur.newwin(lines, cols, y, x))
    my_wins.append(cur.newwin(lines, cols, y+1, x+5))
    my_wins.append(cur.newwin(lines, cols, y+2, x+10))

    # Create borders around windows to see the effect
    for win in my_wins:
       win.box(0, 0)

    # Attach a panel to each window - order bottom up
    my_panels.append(pan.new_panel(my_wins[0])) # order: stdscr-0
    my_panels.append(pan.new_panel(my_wins[1])) # order: stdscr-0-1
    my_panels.append(pan.new_panel(my_wins[2])) # order: stdscr-0-1-2

    # write some identifying text
    my_wins[0].addstr(1,1,"Window 1",cur.color_pair(1))
    my_wins[1].addstr(1,1,"Window 2",cur.color_pair(2))
    my_wins[2].addstr(1,1,"Window 3",cur.color_pair(3))
```

```
    # Update the stacking order. panel 2 will be on top
    pan.update_panels()
    cur.doupdate()      # Show it on the screen

    scr.getch()  # pause...

cur.wrapper(main)
```

As you can see, the program follows the simple flow as explained above. The windows are created with `curses.newwin()` and then they are attached to panels with `panel.new_panel()`. As we attach one panel after another, the stack of panels gets updated. To put them on screen `panel.update_panels()` and `curses.doupdate()` are called.

12.2 Panel Window Browsing

A slightly more complex example is given below. This program creates 3 windows which can be cycled through using the tab key. Have a look at the code.

Example 15. Panel Window Browsing Example

```
import curses as cur
import curses.panel as pan

NLINES=10
NCOLS=40
KEY_TAB = 9   # define a new Key constant

def main(scr):
   my_panels = []
   scr.keypad(True)

   # Initialize all the colors
   cur.start_color()
   cur.init_pair(1, cur.COLOR_RED, cur.COLOR_BLACK);
   cur.init_pair(2, cur.COLOR_GREEN, cur.COLOR_BLACK);
   cur.init_pair(3, cur.COLOR_BLUE, cur.COLOR_BLACK);
   cur.init_pair(4, cur.COLOR_CYAN, cur.COLOR_BLACK);

   my_wins = init_wins(3)

   # Attach a panel to each window, Order is bottom up
   my_panels.append(pan.new_panel(my_wins[0]))
   my_panels.append(pan.new_panel(my_wins[1]))
   my_panels.append(pan.new_panel(my_wins[2]))
```

```
   # Set up the user pointers to the next panel
   my_panels[0].set_userptr(my_panels[1])
   my_panels[1].set_userptr(my_panels[2])
   my_panels[2].set_userptr(my_panels[0])

   # Update the stacking order. panel 2 will be on top
   pan.update_panels()

   # Show it on the screen
   scr.attron(cur.color_pair(4))
   scr.addstr(cur.LINES-2, 0,
              "Use tab to browse through the windows (F1 to Exit)")
   scr.attroff(cur.color_pair(4))
   cur.doupdate()

   top = my_panels[-1]

   # start event loop
   while True:
       ch = scr.getch()
       if ch == cur.KEY_F1:
           break  # exit event loop
       if ch == KEY_TAB:
          next_top = pan.top_panel().userptr()
          next_top.top()
       pan.update_panels();
       cur.doupdate()

# Create all the windows
def init_wins(n_wins):
    y = 2
    x = 10
    wins = []
    for n in range(n_wins):
        win = cur.newwin(NLINES, NCOLS, y+(3*n), x+(7*n))
        lbl = "Window Number %d" % (n+1)
        win_show(win, lbl, n+1)
        wins.append(win)
    return wins

# Show the window with a border and a label
def win_show(win, label, label_color):
    starty,startx = win.getbegyx()
    height,width = win.getmaxyx()
    win.box(0, 0)
    win.addch(2, 0, cur.ACS_LTEE)
    win.hline(2, 1, cur.ACS_HLINE, width-2)
```

```
    win.addch(2, width-1, cur.ACS_RTEE)
    print_in_middle(win, 1, label, cur.color_pair(label_color))

# Print label in middle of line
def print_in_middle(win, line, label, col_pair):
    if win == None:
       win = cur.stdscr
    _,width = win.getmaxyx()
    length = len(label)
    x = (width-length) // 2
    win.addstr(line, x, label, col_pair)
    win.attroff(col_pair)
    win.refresh()

cur.wrapper(main)
```

12.3 Using User Pointers

In the above example I used user pointers to find out the next window in the cycle. We can attach custom information to the panel by specifying a user pointer, which can point to any information you want to store. In this case I stored the pointer to the next panel in the cycle. The user pointer for a panel can be set with the method `Panel.set_userptr()`. It can be accessed using the method `Panel.userptr()` which will return the user pointer. After finding the next panel in the cycle it's brought to the top by calling its method `Panel.top()`.

12.4 Moving and Resizing Panels

The function `move_panel()` can be used to move a panel to the desired location. It does not change the position of the panel in the stack. Make sure that you use `move_panel()` instead of using `mvwin()` on the window associated with the panel.

Resizing a panel is slightly complex. There is no straight forward function just to resize the window associated with a panel. A solution to resize a panel is to create a new window with the desired sizes, change the window associated with the panel using `replace_panel()`. Don't forget to delete the old window. The window associated with a panel can be found by using the function `panel_window()`.

The following program shows these concepts, in supposedly simple program. You can cycle through the window with <TAB> as usual. To resize or move the active panel press 'r' for resize 'm' for moving. Then use arrow keys to resize or move it to the desired way and press enter to end your resizing or moving. This example makes use of a user defined class to hold the data required for the operations and stores it in the `userptr` field.

Example 16. Panel Moving and Resizing example

```
import curses as cur
import curses.panel as pan

class Panel_Data:  # data record for panel userptr
    def __init__(self, x, y, w, h, label, color):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.label = label
        self.color = color
        self.next = None

NLINES = 10
NCOLS  = 40
KEY_TAB = 9
KEY_RETURN = 10

def  main(scr):
    resize,move = False,False

    # Initialize all the colors
    cur.init_pair(1, cur.COLOR_RED, cur.COLOR_BLACK)
    cur.init_pair(2, cur.COLOR_GREEN, cur.COLOR_BLACK)
    cur.init_pair(3, cur.COLOR_BLUE, cur.COLOR_BLACK)
    cur.init_pair(4, cur.COLOR_CYAN, cur.COLOR_BLACK)


    # Attach a panel to each window. Order is bottom up
    my_wins = init_wins(4)      # create 4 windows
    my_panels = []
    for w in my_wins:
        my_panels.append(pan.new_panel(w))
    set_user_ptrs(my_panels)
    pan.update_panels()

    # Show it on the screen
    scr.attron(cur.color_pair(4))
    scr.addstr(cur.LINES-3, 0, "Use 'm' for moving, 'r' for resizing")
    scr.addstr(cur.LINES-2, 0, "Use tab to select window (F1 to Exit)")
    scr.attroff(cur.color_pair(4))
    cur.doupdate()

    stack_top = my_panels[-1]   # use last one added
    top_data  = stack_top.userptr()

    # start event loop
```

```
while True:
    ch = scr.getch()
    if ch == cur.KEY_F1:  # exit
        break
    if ch == KEY_TAB:
        stack_top = top_data.next       # get next panel
        top_data = stack_top.userptr()  # get new panel data
        stack_top.top()                 # set new top panel
    elif ch == ord('r'): # Re-Size
       resize = True
       scr.attron(cur.color_pair(4));
       scr.addstr(cur.LINES-4, 0,
              "Entered Resizing :Use Arrows to resize, <ENTER> to end")
       scr.refresh()
       scr.attroff(cur.color_pair(4))
    elif ch == ord('m'): # Move
       move = True
       scr.attron(cur.color_pair(4))
       scr.addstr(cur.LINES-4, 0,
               "Entered Moving: Use Arrows to Move, <ENTER> to end")
       scr.refresh()
       scr.attroff(cur.color_pair(4))
    elif ch == cur.KEY_LEFT:
       if resize:
          top_data.x -= 1
          top_data.w += 1
       if move:
          top_data.x -= 1
    elif ch == cur.KEY_RIGHT:
       if resize:
          top_data.x += 1
          top_data.w -= 1
       if move:
          top_data.x += 1
    elif ch == cur.KEY_UP:
       if resize:
          top_data.y -= 1
          top_data.h += 1
       if move:
          top_data.y -= 1
    elif ch == cur.KEY_DOWN:
       if resize:
          top_data.y += 1
          top_data.h -= 1
       if move:
          top_data.y += 1
    elif ch == KEY_RETURN:
```

```
                scr.move(cur.LINES-4, 0)
                scr.clrtoeol();
                scr.refresh()
                if resize:
                    resize = False
                    new_win = cur.newwin(top_data.h, top_data.w,
                                         top_data.y, top_data.x)
                    win_show(new_win, top_data.label, top_data.color)
                    stack_top.replace(new_win)
                    stack_top.move(top_data.y, top_data.x)   # avoids gaps
                if move:
                    move = False
                    stack_top.move(top_data.y, top_data.x)
            scr.attron(cur.color_pair(4))
            scr.addstr(cur.LINES-3, 0,
                       "Use 'm' for moving, 'r' for resizing")
            scr.addstr(cur.LINES-2, 0,
                       "Use tab to browse through the windows (F1 to Exit)")
            scr.attroff(cur.color_pair(4))
            scr.refresh()
            pan.update_panels()
            cur.doupdate()

# Set the userptr data for individual panels
def set_user_ptrs(panels):
    ptrs = []
    for n,panel in enumerate(panels):
       w = panel.window()
       y,x = w.getbegyx()
       h,w = w.getmaxyx()
       data = Panel_Data(x,y,w,h,"Window Number %d" % (n+1), n+1)
       if n == len(panels)-1:  # at last panel
          data.next = panels[0]
       else: data.next = panels[n+1]
       panel.set_userptr(data)

# Create all the windows
def init_wins(n_wins):
    y = 2
    x = 10
    wins = []
    for n in range(n_wins):
        win = cur.newwin(NLINES, NCOLS, y+(3*n), x+(7*n))
        lbl = "Window Number %d" % (n+1)
        win_show(win, lbl, n+1)
        wins.append(win)
    return wins
```

```
# Show the window with a border and a label
def win_show(win, label, label_color):
    starty,startx = win.getbegyx()
    height,width = win.getmaxyx()
    win.box(0, 0)
    win.addch(2, 0, cur.ACS_LTEE)
    win.hline(2, 1, cur.ACS_HLINE, width-2)
    win.addch(2, width-1, cur.ACS_RTEE)
    print_in_middle(win, 1, label, cur.color_pair(label_color))
    win.refresh()

def print_in_middle(win, line, label, col_pair):
    if win == None:
       win = cur.stdscr
    _,width = win.getmaxyx()
    length = len(label)
    x = (width-length) // 2
    win.addstr(line, x, label, col_pair)
    win.attroff(col_pair)
    win.refresh()

cur.wrapper(main)
```

Concentrate on the main while loop. Once it finds out the type of key pressed, it takes appropriate action. If 'r' is pressed resizing mode is started. After this the new sizes are updated as the user presses the arrow keys.

When the user presses <ENTER> present selection ends and panel is resized by using the concept explained.

When the user presses 'm' the move mode starts. This is a bit simpler than resizing. As the arrow keys are pressed the new position is updated and pressing of <ENTER> causes the panel to be moved by calling the `move()` method.

While in resizing or moving modes the program doesn't show how the window is getting resized or moved. It's left as an exercise to the reader to print a dotted border showng the new size or position.

In this program the user data which is stored in the `Panel_Data` objects, plays very important role in finding the associated information with a panel. As written in the comments, the `Panel_Data` stores the panel sizes, label, label color and the next panel in the cycle.

Note: The `move()` method is required in both the move and resize actions since without it the window will not draw itself completely, there will be missing blocks.

12.5 Hiding and Showing Panels

A panel can be hidden by using the method `Panel.hide()`. This method merely removes it form the stack of panels, thus hiding it on the screen once you do `panel.update_panels()` and `cur.doupdate()`. It doesn't destroy the Panel object structure associated with the hidden panel. It can be shown again by using the `Panel.show()` method.

The following program shows the hiding of panels. Press 'a' or 'b' or 'c' to show or hide first, second and third windows respectively. It uses a user data with a small variable hide, which keeps track of whether the window is hidden or not. (There may be a bug in the underlying ncurses panel code for the `Panel.hidden()` method which tells whether a panel is hidden or not. A bug report was presented by Michael Andres. It is not clear whether this has been fixed at the time of writing.)

Example 17. Panel Hiding and Showing example

```
import curses as cur
import curses.panel as pan

NLINES = 10
NCOLS  = 40
VISIBLE = True
HIDDEN = False

def main(scr):

    # Initialize all the colors
    cur.init_pair(1, cur.COLOR_RED, cur.COLOR_BLACK)
    cur.init_pair(2, cur.COLOR_GREEN, cur.COLOR_BLACK)
    cur.init_pair(3, cur.COLOR_BLUE, cur.COLOR_BLACK)
    cur.init_pair(4, cur.COLOR_CYAN, cur.COLOR_BLACK)

    # Attach a panel to each window. Set all visible
    my_wins = init_wins(4)   # create 4 wndows
    my_panels = []
    for w in my_wins:
        my_panels.append(pan.new_panel(w))
        my_panels[-1].set_userptr(VISIBLE)
    pan.update_panels()

    # Show it on the screen
    scr.attron(cur.color_pair(4))
    scr.addstr(cur.LINES-3, 0, "Use '1-4' to toggle visibility")
    scr.addstr(cur.LINES-2, 0, "F1 to Exit")
    scr.attroff(cur.color_pair(4))
    cur.doupdate()

    # Start event loop
    while True:
        ch = scr.getch()
        if ch == cur.KEY_F1:
            break
        if ch in  [ord('1'), ord('2'), ord('3'), ord('4')]:
            index = int(chr(ch)) - 1   # zero index!
            thePanel = my_panels[index]
            if thePanel.userptr() == HIDDEN:
               thePanel.set_userptr(VISIBLE)
               thePanel.show()
            else:
                thePanel.set_userptr(HIDDEN)
                thePanel.hide()
```

```
        pan.update_panels()
        cur.doupdate()

# Create all the windows
def init_wins(n_wins):
    y = 2
    x = 10
    wins = []
    for n in range(n_wins):
        win = cur.newwin(NLINES, NCOLS, y+(3*n), x+(7*n))
        lbl = "Window Number %d" % (n+1)
        win_show(win, lbl, n+1)
        wins.append(win)
    return wins

# Show the window with a border and a label
def win_show(win, label, label_color):
    starty,startx = win.getbegyx()
    height,width = win.getmaxyx()
    win.box(0, 0)
    win.addch(2, 0, cur.ACS_LTEE)
    win.hline(2, 1, cur.ACS_HLINE, width-2)
    win.addch(2, width-1, cur.ACS_RTEE)
    print_in_middle(win, 1, label, cur.color_pair(label_color))
    win.refresh()

def print_in_middle(win, line, label, col_pair):
    if win == None:
       win = cur.stdscr
    _,width = win.getmaxyx()
    length = len(label)
    x = (width-length) // 2
    win.addstr(line, x, label, col_pair)
    win.attroff(col_pair)
    win.refresh()

cur.wrapper(main)
```

12.6 panel_above() & panel_below() Methods

The `Panel.above()` and `Panel.below()` methods can be used to find out the panel above and below a panel. If the argument to these functions is `None`, then they return the top panel and bottom panel respectively.

13 Tools and Widget Libraries

Now that you have seen the capabilities of ncurses and its sister library panels, you are rolling your sleeves up and gearing for a project that heavily manipulates the screen. However, it can be pretty difficult to write and maintain complex GUI widgets in plain ncurses or even with panels. There are some other ready -to-use tools and widget libraries that can be used instead of writing your own widgets. You can use some of them, get ideas from the code, or even extend them. Unfortunately most of the C libraries are not ported to Python so you would have to create the wrapper yourself. There are some curses extras available in the PyPI repository and a search might be worth while. (https://pypi.org/)

13.1 curses.textpad

While Python does not support all of the C libraries for curses there is one bonus feature that is available in Python that is not part of regular curses and that is a text editor widget that supports the basic emacs keystrokes (Ctrl-a = start of line, Ctrl-k = delete to end of line, etc.) It also supports the cursor movement arrow keys etc. It is known as the textpad and this section provides a basic example.

Example 18 curses.textpad example

```
import curses as cur
from curses.textpad import Textbox

def main(scr):
    s = "Here is a string to insert..."
    scr.addstr(0,0,"Edit the text then hit Ctrl-G to exit")
    scr.refresh()

    # create the text box with border around the outside.
    tb_border = cur.newwin(12,52,4,4)
    tb_border.box(0,0)
    tb_border.refresh()
    tb_body = cur.newwin(10,50,5,5)
    tb = Textbox(tb_body)

    for c in s:          #insert the starting text
        tb.do_command(c)
    tb.edit()            # start the editor running, Ctr-G ends
    s2 = tb.gather()     # fetch the contents

    scr.clear()          # clear the screen
```

```
    scr.addstr(0,0,"The text in the box was:\n")
    scr.addstr(3,0,s2)   # display edited contents of textbox
    scr.refresh()

    scr.getch()


cur.wrapper(main)
```

Note that by default the textbox has no border, you need to create a window one row and one column larger all round and add the border there.

13.2 dialog

The `dialog` command is a real gem in making professional-looking dialog boxes with ease. It creates a variety of dialog boxes, menus, check lists etc. It is usually installed by default on Linux systems. The man page has the details.

`dialog` was initially designed to be used with shell scripts. However there is a Python wrapper of `dialog` that can be accessed programmatically, including from within curses. You need `dialog` to be installed on your OS, but most Linux systems include it. The Python module can be installed from the Python Package Index (PyPI) with

```
$ python3 -m pip install pythondialog
```

A simple example showing a curses program displaying a message using dialog follows.

Example 19 curses and dialog example

```
import curses as cur
import dialog


def main(scr):
    d = dialog.Dialog() # initialize the dialog

    # use curses to get the input.
    cur.echo()    # show the input as typed
    scr.addstr(3,3,"Hello, what's your name? ")
    scr.refresh()
    nm = scr.getstr()

    # use dialog to display the name
    d.msgbox("Hi %s, nice to meet you." % nm.decode('utf-8'))

    # go back to curses to clear up
    scr.clear()
```

```
    scr.addstr("\nGoodbye")
    scr.refresh()
    cur.napms(1000)

cur.wrapper(main)
```

The dialog system includes over 20 different widgets including file browsers, menus, progress gauges, text browsers, calendars and more. The web page provides a tutorial and complete reference:

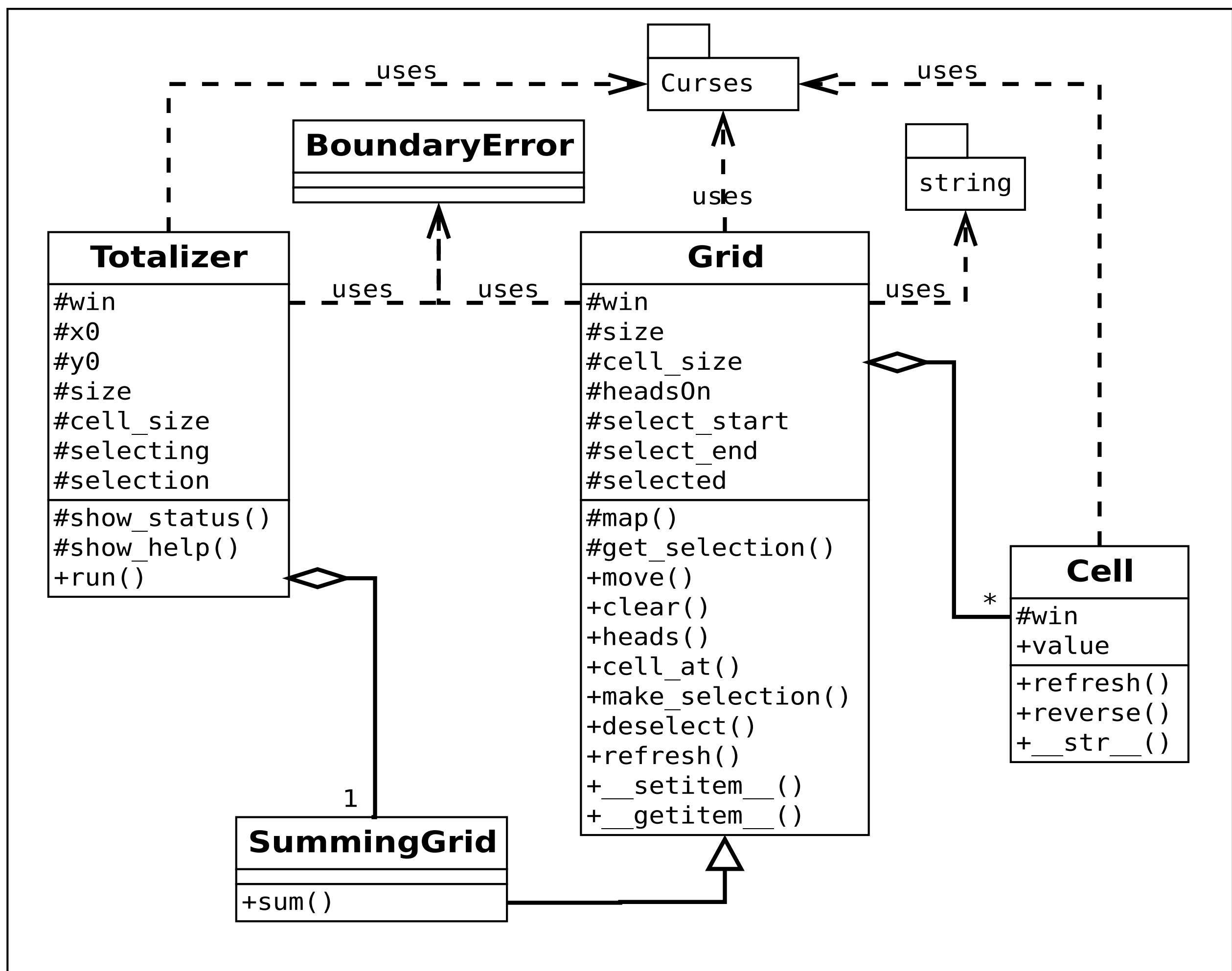
http://pythondialog.sourceforge.net/doc/index.html

14 A Case Study – The Totalizer

So far we have used curses to build some short demonstration programs. In this chapter I want to look at a more real-world type of project using curses as the display mechanism but incorporating it into an object-oriented application, typical of the sort created for real users. We will build a mini spreadsheet-like application with a grid display of cells into which data can be entered and the total of the current column displayed. We will build it from a set of classes which are sufficiently general to be used for other types of grid based applications. It will incorporate some error checking although not to full industrial standards to save space. For the same reasons we will not write (or even discuss) unit tests such as would be used in a real project.

14.1 Totalizer Design Summary

The design for the Totalizer application is shown in the following simplified UML diagram.



The most basic class is the `Cell`, which simply holds a value and displays it. It also performs the valuable service of abstracting most of the low level curses code away from the higher level application classes. We can toggle inverse video on or off.

The next level of abstraction is a `Grid` component which uses a table structure of `Cells` and can, optionally, display headings for rows and columns. The clients can move the cursor to individual cells and store values into specified cells. There is a specialization of the `Grid` called a `SummingGrid` which adds the ability to sum the column in which the cursor is currently located.

Finally, we have the `Totalizer` application class. This creates a `SummingGrid` and starts an event loop. It processes the navigation events and the data assignment, summation and exit commands.

Let’s look at each of these classes in more detail.

14.2 The Cell Class

The `Cell` class is based on a `curses` subwindow. It stores the original data in a “private” field, `_value` which is exposed as a Python property, `value`. There is a private `refresh()` method (conventionally called `_refresh()` in Python) which simply refreshes the underlying curses subwindow.

The public `reverse()` operation turns inverse video on or off for the cell.

Setting the cell value automatically updates the cell display. A `ValueError` is raised if the value is too big to fit the cell display. A `__str__()` operator is provided to simplify the display code.

The code is shown below:

```
import curses as cur

class Cell:
    def __init__(self, win, w,y,x, v=None):
        self.att = cur.A_NORMAL
        self.width = w
        self.isReversed = False
        self.theCell = win.subwin(3,w+2,y,x)
        self.theCell.box(0,0)
        self.value = v
        self.refresh()

    @property
    def value(self): return self._value

    @value.setter
    def value(self,v):
        v = v if v else ''
```

```
        if len(str(v)) > self.width:
            raise ValueError("%d too large value for cell"% v)
        self._value = v   # actually store the value
        self.theCell.addstr(1,1,str(v).ljust(self.width))
        self.refresh()    # now display it

    def reverse(self):
        self.att = cur.A_NORMAL if self.isReversed else cur.A_REVERSE
        self.isReversed = not self.isReversed
        self.refresh()

    def refresh(self):
        self.theCell.attrset(self.att)
        self.theCell.box(0,0)
        self.theCell.addstr(1,1,str(self.value).ljust(self.width))
        self.theCell.refresh()

    def __str__(self):
        return str(self.value)
```

14.3 The Grid Class

The `Grid` is essentially a two dimensional array of `Cells`. In a sense it is like a bigger form of a curses window (which is an array of chars) and supports similar concepts.

The `move()` operation, like its curses counterpart, locates the cursor into a cell based on grid `y,x` coordinates. There is a private `_map()` method that converts the grid coordinates into underlying window coordinates so that the curses `move()` call ends up in the appropriate place. The coordinate values are checked for size and if outside the grid boundaries a bespoke `BoundaryError` is raised.

The `clear()` operation simply deletes all of the cell contents and resets the cursor to the first cell.

The `heads()` operation turns on a spreadsheet-like header display with numbers along the top row and uppercase letters down the leftmost column, both displayed in inverse video. The application does not (currently!) use these letters and numbers for navigation, they are merely for the users convenience in locating specific cells.

A `cell_at()` method translates the window coordinates into a grid cell coordinate-pair which can be used by the `move()` method. This is needed to handle mouse events.

The `make_selection()` and `deselect()` operations create and remove a highlighted selection area within the grid. A `get_selection()` private method returns a list of cells within the current selection.

Finally, we implement the `getitem/setitem` pair of operators to provide indexing into the grid. This allows clients to access cells as if using a table directly. We only implent single a index which returns a list of `Cells`. This list will handle the second index and the slicing operations needed for the `sum()` method. It is a convenience feature which hides the internal cell array.

The code is as follows:

```
import curses as cur
from cell import Cell
import string

class BoundaryError(ValueError): pass

class Grid:
    ''' creates a grid of cells.
    Cells are accessed using grid rather than window
    coordinates. Allows movement to a cell, and reversing
    of display'''

    def __init__(self,win, ht, wd, y, x, cell_size=8):
        self.win = win
        self.headsOn = False
        self.selected = False
        self.select_start = None
        self.select_end = None
        self.size = (ht,wd)
        self.origin = (0,0)    # initial origin with heads off
        self.yx = self.origin  # initial active cell
        self.cell_size = cell_size
        self.cells = []
        for r in range(ht):   # generate empty grid
            row_y = y + (r*3)
            row = [Cell(win,cell_size, row_y,col*(cell_size+2)+x)
                   for col in range(wd)]
            self.cells.append(row)

    def _map(self,y,x):   # helper function
        ''' maps grid coords to window coords'''
        wy = 1 + (3*y)
        wx = 1 + (self.cell_size+2) * x
        return wy,wx
```

```
def _get_selection(self):
    ‘’’ return a ist of all cells currently selected’’’
    if not self.selected:
        return []
    start_row,start_col = self.select_start
    end_row,end_col = self.select_end
    cells = []
    for row in range(start_row,end_row+1):
        for col in range(start_col,end_col+1):
            cells.append(self[row][col])
    return cells

def move(self,y,x):
    ''' move cursor to cell y,x in grid'''
    if (y >= self.size[0] or  # check within boundaries
        x >= self.size[1] or
        y < self.origin[0] or
        x < self.origin[1]):
        raise BoundaryError("%d or %d outside grid"%(y,x))
    ypt,xpt = self._map(y,x)
    self.win.move(ypt,xpt)     # uses curses move()
    self.yx = (y,x)

def clear(self):
    ‘’’ clear all cells ‘’’
    for row in self[self.origin[0]:]:
        for cell in row[self.origin[1]:]:
            cell.value = None
            cell.refresh()
    self.move(*self.origin)

def heads(self):
    ''' insert numbers along row 0 and letters down col 0
        reverses cells in row 0 and column 0
    '''
    letters = string.ascii_uppercase
    for num,cell in enumerate(self.cells[0]):
        if num > 0:
           cell.value= str(num).center(self.cell_size)
        cell.reverse()
    for index,row in enumerate(self.cells[1:]):
          row[0].value = letters[index]
          row[0].reverse()        for row in self[self.origin[0]]:
        for cell in row:
            cell.value = None
    self.headsOn = True
    self.origin = (1,1)
```

```
        self.move(*self.origin)

    def cell_at(self, y,x):
        ''' Find cell with containing curses coords y,x.
            return grid y,x coordinates of cell '''
        row = y//3        for row in self[self.origin[0]]:
            for cell in row:
                cell.value = None
        col = x//(self.cell_size+2)
        return row,col

    def make_selection(self,start,end):
    ''' show selected cells in inverse video and set flag attributes'''
        self.selected = True
        self.select_start = start
        self.select_end = end
        for row in range(start[0],end[0]+1):
            for col in range(start[1],end[1]+1):
                self[row][col].reverse()
        if start == end:  # need to reinvert first cell
            self[start[0]][start[1]].reverse()

    def deselect(self):
    ''' deselect the grid by reversing cells and resetting attributes'''
        for row in range(self.select_start[0],self.select_end[0]+1):
            for col in range(self.select_start[1],self.select_end[1]+1):
                self[row][col].reverse()
        self.selected = False
        self.select_start = None
        self.select_end = None

    def refresh(self):
        ‘’’ redraw the grid ‘’’
        for row in self.cells:
            for cell in row:
                cell.refresh()

    # allow access to cells via indexing of grid
    def __setitem__(self,index,cell):
        self.cells[index] = cell

    def __getitem__(self,index):
        return self.cells[index]
```

14.4 The SummingGrid Class

The SummingGrid class is a simple subclass of Grid. It adds a single operation, sum(), which adds all the values in the current column which are not zero (or null) and returns the result. It takes account of whether headers exist or not. If a selection exists it returns the sum of all cells within the selection.

The code follows:

```
class SummingGrid(Grid):
    '''Adds ability to total a column to basic Grid'''

    def __init__(self,win,ht,wd,y,x, cell_size=8):
        super().__init__(win,ht,wd,y,x,cell_size)

    def sum(self,col):
        ''' sums all values in current column'''
        first = 1 if self.headsOn else 0
        if self.selected:
            cells = self._get_selection()
            vals = [cell.value for cell in cells if cell.value]
        else:
            vals = [row[col].value for row in self[first:] if row[col].value]
        return sum(vals)
```

14.5 The Totalizer Class

The Totalizer is the application class. It creates a SummingGrid and turns on headers. Each cell displays itself as it is added and if it cannot fit into its window curses will raise an error. There is a convenience method, show_status(), for displaying messages on the bottom line of the window. The show_help() method displays a new window with help instructions. Any keypress or mouse click will close it and the grid will be refreshed. There is a single public operation: run().

run() starts the event loop and processes the events. Allowed events include:

- arrow movements (or the vi editor key equivalents 'hjkl') to navigate the grid (complete with boundary checking).
- "=" an assignment operation comprised of the equals sign followed by a value, for example, typing "=42<RETURN>" will insert 42 into the current cell and move the curse down to the next cell in the column.
- "+" which totals the current column (or selection if one is active), displaying the result as a status message,
- "C" which clears the contents of the grid

- “S” which controls selection. The first ‘S’ starts a selection and the chosen cell is highlighted. A second ‘S’ marks the end of the selection and the range of cells is highlighted. Any addition operations while the selection is active will apply to the selection. A third ‘S’ will deselect the region returning things to normal.
- “X” exits the application.

Finally we handle mouse clicks to navigate to a new cell.

The code is as follows:

```
import curses as cur
from grid import Grid, BoundaryError

# define mouse event indices
X_COORD = 1
Y_COORD = 2
BUTTON_STATE = 4

class Totalizer:
    ''' spreadsheet-like grid that can display totals of columns.
        values must be integers.'''

    help_string = "Press F1 for help, X to eXit."


    def __init__(self, win, ht,wd, cell_size=8):
        self.win = win
        self.size = (ht,wd)
        self.cell_size = cell_size
        self.selecting = False
        self.selection = None
        self.grid = SummingGrid(win,ht,wd, 0,0, cell_size)
        self.grid.heads()
        cur.mousemask(cur.ALL_MOUSE_EVENTS)

    def _show_status(self,msg):
        ''' display message on bottom line of window '''
        Y,X = self.win.getmaxyx()
        y,x = self.grid.yx
        self.win.addstr(Y-2,1, ' '*(X-2))  #clear line but not border
        self.win.addstr(Y-2,1, msg)
        self.grid.move(y,x)                # move cursor back to active cell

    def _show_help(self):
        ‘’’Show help screen in new window, remove window on any key’’’
```

```
        ht,wd = self.win.getmaxyx()
        top = (ht-15)//2
        left = (wd-50)//2
        win = cur.newwin(15,50,top,left)
        win.keypad(True)   # accept mouse clicks and special keys
        win.box(0,0)
        win.addstr(1,2,"Arrow keys move cursor")
        win.addstr(3,2,"Mouse positions cursor")
        win.addstr(5,2,"=N<RETURN> inserts value")
        win.addstr(7,2,"+ Sums the current column")
        win.addstr(9,2,”S Start/end/cancel selection”)
        win.addstr(11,2,”C Clear grid”)
        win.addstr(13,2,"X eXit the application")
        win.refresh()
        win.getch()    # any key or mouse click clears help screen
        win.erase()
        self.grid.refresh()
        self.grid.move(*self.grid.yx) # restore cursor to previous cell

    def run(self):
        ''' start the event loop, process the actions '''
        Y,X = self.win.getmaxyx()
        self.win.addstr(Y-3,1, self.help_string)
        self.grid.move(*self.grid.origin)  # starting position
        while True:
           key = self.win.getch()
           if key == ord('X'): break
           if key == cur.KEY_F1:
              self._show_help()
           if key in [cur.KEY_UP, ord('k')]:
              y,x = self.grid.yx
              try: self.grid.move(y-1,x)
              except BoundaryError as e:
                  self._show_status(e.args[0])
           elif key in [cur.KEY_DOWN, ord('j')]:
              y,x = self.grid.yx
              try: self.grid.move(y+1,x)
              except BoundaryError as e:
                  self._show_status(e.args[0])
           elif key in [ cur.KEY_LEFT, ord('h')]:
              y,x = self.grid.yx
              try: self.grid.move(y,x-1)
              except BoundaryError as e:
                  self._show_status(e.args[0])
           elif key in [ cur.KEY_RIGHT, ord('l')]:
              y,x = self.grid.yx
              try: self.grid.move(y,x+1)
```

```
            except BoundaryError as e:
                self._show_status(e.args[0])
        elif key == ord('='):
            val = self.win.getstr()      # now read the data value
            y,x = self.grid.yx
            try:
                self.grid[y][x].value = int(val)  #only want integers
                self.grid.move(y+1,x)
            except BoundaryError: pass   # leave it as-is
            except ValueError:
                self._show_status("Error: invalid value - %s" % val)
        elif key == ord('+'):
            y,x = self.grid.yx
            tot = self.grid.sum(x)
            self._show_status("Column %d total = %d" % (x,tot))
        elif key == ord('S'):
             if not self.selecting and not self.selection:   # first S
                 self.selecting = True
                 self.selection = self.grid.yx
                 self.grid.make_selection(self.selection,self.selection)
                 self._show_status("Selection ON")
             elif self.selecting:     # second S
                 self.selecting = False
                 self.grid.make_selection(self.selection, self.grid.yx)
                 self._show_status("Selection complete")
             else:                    # third S
                 self.grid.deselect()
                 self.selection = None
                 self._show_status("Selection off")
        elif key == ord("C"):
             self.grid.clear()
        elif key == cur.KEY_MOUSE:
            m_event = cur.getmouse()
            if m_event[BUTTON_STATE] | cur.BUTTON1_CLICKED:
                m_y = m_event[Y_COORD]
                m_x = m_event[X_COORD]
                g_y,g_x = self.grid.cell_at(m_y,m_x)
                try: self.grid.move(g_y,g_x)
                except BoundaryError: pass   # leave it as-is
```

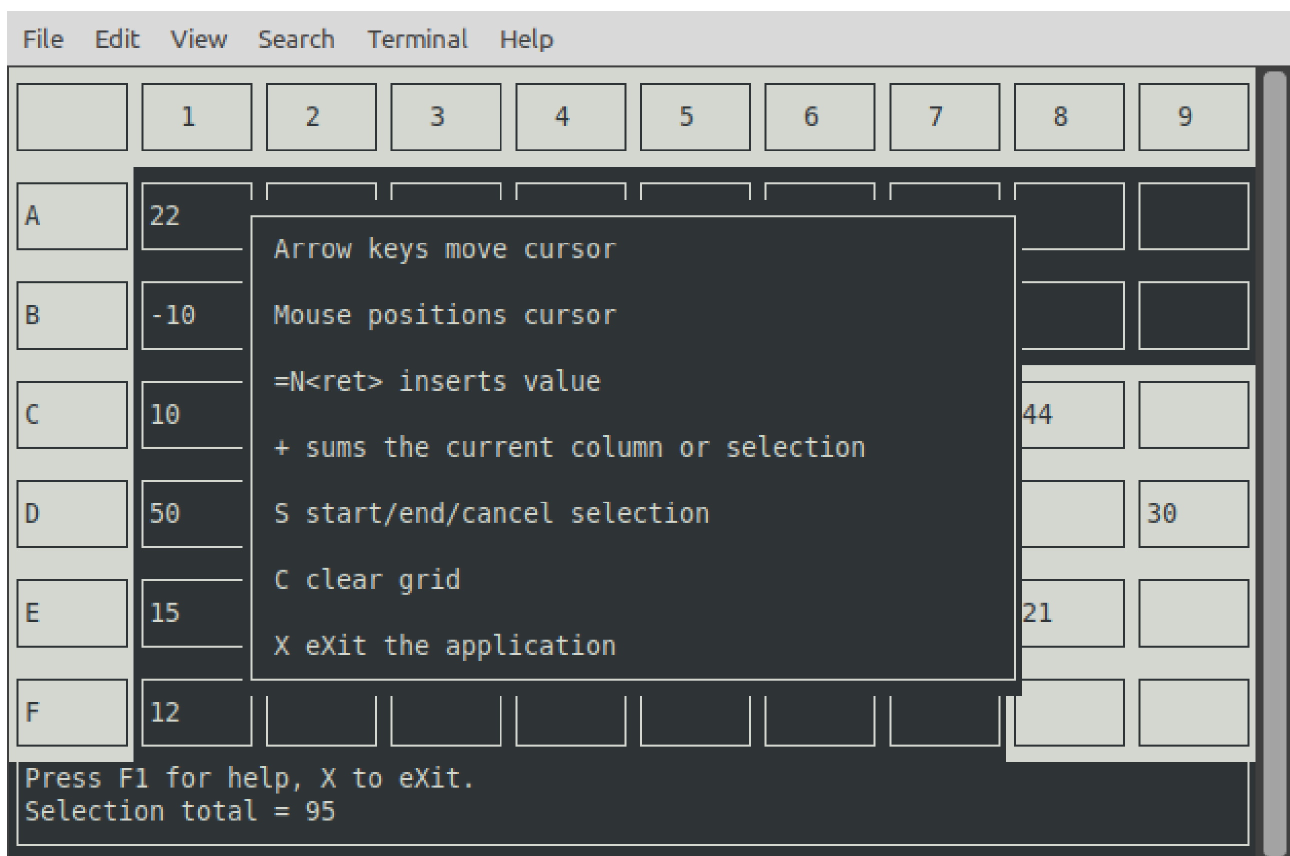
14.6 The Driver Code

Finally the driver code for the application in the `main()` function is called by the `curses.wrapper()`. It draws a box around `stdscr` then creates a `Totalizer`. Finally the event loop is started by sending the `run()` message to the instance.

The code follows:

```
import curses as cur
...
def main(scr):
    scr.box(0,0)
    scr.refresh()
    totalizer = Totalizer(scr,7,10,6)
    totalizer.run()
...
if __name__ == "__main__": cur.wrapper(main)
```

The following screenshot of the finished application shows the grid with headings, some entered values and the total of a selected range displayed at the bottom. (Notice that negative numbers are handled correctly too.)



14.7 Things to Consider

The code is reasonably functional but could be improved in many ways.

- The error handling could be enhanced, for example, to prevent a Totalizer bigger than the available window being created (or any curses errors raised by the Cells could be caught).

- Extra operations could be added – for example a sort operation.. File based saving and reading could be introduced (possibly utilizing the dialog module to select folders and filenames). Ultimately, you could reinvent the spreadsheet and relive the glory days of Visicalc, Lotus 123 and Supercalc.
- A right-click mouse menu could also be introduced, especially if more operations were added. Simply pop up a new window, just as we did for the help screen, with the list of options then allow the user to select one before closing the menu window again and performing the chosen operation.
- Different sub classes of grid could be used to build different applications such as a text based grid for crossword puzzles and other word games, or grids for games such as sudoku, oxo, magic square etc.
- The design of curses is not naturally conducive to the use of a Model-View-Controller (MVC) framework but with some extra effort such a framework could be constructed and thus have the data values stored completely separately from the display components. In that scenario the cells would become views of the data and the grid a view of the cells. However, creating such an architecture would involve writing almost as much code again as we have here.

All of these suggestions are left as exercises for the reader.

15 Just For Fun !!!

This section describes a few programs written by Pradeep (using C curses) just for fun. They don't signify a better programming practice or the best way of using ncurses. They are provided here so as to allow beginners to get ideas.

If you fetch the code you can translate it into Python using the material in this document combined with the curses module documentation.

Pradeep’s source code tarball can be downloaded from here:

http://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/ncurses_programs.tar.gz

15.1 The Game of Life

Game of life is a wonder of math. In Paul Callahan's words

> *The Game of Life (or simply Life) is not a game in the conventional sense. There are no players, and no winning or losing. Once the "pieces" are placed in the starting position, the rules determine everything that happens later. Nevertheless, Life is full of surprises! In most cases, it is impossible to look at a starting position (or pattern) and see what will happen in the future. The only way to find out is to follow the rules of the game.*

This program starts with a simple inverted U pattern and shows how wonderful life works. There is a lot of room for improvement in the program. You can let the user enter pattern of his choice or even take input from a file. You can also change rules and play with a lot of variations. Search on Google for interesting information on game of life.

File Path: JustForFun/life.c

15.2 Magic Square

Magic Square, another wonder of math, is very simple to understand but very difficult to make. In a magic square sum of the numbers in each row, each column is equal. Even a diagonal sum can be equal. There are many variations which have special properties.

This program creates a simple magic square of odd order.

File Path: JustForFun/magic.c

15.3 Towers of Hanoi

The famous towers of hanoi solver. The aim of the game is to move the disks on the first peg to the last peg, using the middle peg as a temporary stay. The catch is not to place a larger disk over a smaller disk at any time.

File Path: JustForFun/hanoi.c

15.4 Queens Puzzle

The objective of the famous N−Queen puzzle is to put N queens on an N x N chess board without attacking each other. This program solves it with a simple backtracking technique.

File Path: JustForFun/queens.c

15.5 Shuffle

A fun game, if you have time to kill.

File Path: JustForFun/shuffle.c

15.6 Typing Tutor

A simple typing tutor, I created more out of need than for ease of use. If you know how to put your fingers correctly on the keyboard, but lack practice, this can be helpful.

File Path: JustForFun/tt.c

16 References

16.1 Online

- NCURSES FAQ pages
  http://invisible−island.net/ncurses/ncurses.faq.html
- Writing programs with NCURSES by Eric Raymond and Zeyd M. Ben−Halim at
  http://invisible−island.net/ncurses/ncurses−intro.html
  This is now somewhat obsolete. Pradeep was inspired by this document and the structure of this HOWTO follows from the original document
- Pradeep's original ncurses How-To document:
  https://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/
- Python curses module documentation:
  https://docs.python.org/3/library/curses.html#module-curses
- Python Panel documentation:
  https://docs.python.org/3/library/curses.panel.html

Also don't forget the man pages on your OS. The curses functions are all covered under section (3) – subroutines. The pages treat related groups of functions together. They only cover the native C ncurses functions, not the Python module.

16.2 Books

There are also a few dead-tree books on the ncurses C library that might prove useful. (I don't know of any published Python curses books.) I can personally recommend the following:

- *Dan Gookins Guide to Ncurses Programming* by Dan Gookin. Self published on Kindle. A low cost tutorial that covers slightly more than this How-To and in a bit more depth.
- *Programming with Curses* by John Strang. Published by O'Reilly Now discontinued but you may be able to pick up a used copy. It focuses on the original BSD library rather than ncurses so there is no discussion of text attributes, color or mouse interaction but for the basics of text manipulation it is pretty comprehensive. If you want to understand how curses works under the covers this is your best bet short of reading the C source code.