

---

# Programmation Curses avec Python

Version 3.11.8

Guido van Rossum and the Python development team

avril 02, 2024

Python Software Foundation  
Email : docs@python.org

## Table des matières

<b>1</b>	<b>Qu'est-ce que <i>curses</i> ?</b>	<b>1</b>
1.1	Le module <i>curses</i> de Python . . . . .	2
<b>2</b>	<b>Lancement et arrêt une application <i>curses</i></b>	<b>2</b>
<b>3</b>	<b>Fenêtres et tampons (<i>pads</i> en anglais)</b>	<b>3</b>
<b>4</b>	<b>Affichage de texte</b>	<b>5</b>
4.1	Attributs et couleurs . . . . .	5
<b>5</b>	<b>Entrées de l'utilisateur</b>	<b>7</b>
<b>6</b>	<b>Pour aller plus loin</b>	<b>8</b>

---

### Auteur

A.M. Kuchling, Eric S. Raymond

### Version

2.04

### Résumé

Ce document décrit comment utiliser le module d'extension *curses* pour contrôler l'affichage en mode texte.

## 1 Qu'est-ce que *curses* ?

La bibliothèque *curses* fournit une capacité de dessin à l'écran et de gestion du clavier indépendante du terminal pour les terminaux textuels ; ces terminaux comprennent les *VT100*, la console Linux et le terminal simulé fourni par divers programmes. Les terminaux d'affichage prennent en charge divers codes de commande pour effectuer des opérations courantes telles que déplacer le curseur, faire défiler l'écran et effacer des zones. Différents terminaux utilisent des codes très différents et ont souvent leurs propres bizarreries mineures.

Dans un monde d'affichages graphiques, on pourrait se demander « pourquoi s'embêter ? ». Il est vrai que les terminaux d'affichage caractère par caractère sont une technologie obsolète, mais il existe des niches pour lesquelles la possibilité

de faire des choses fantaisistes est encore précieuse. En exemple de niche, on peut citer les systèmes de type Unix de petite taille ou embarqués qui n'utilisent pas de serveur X. Il y a aussi les outils tels que les installateurs d'OS et les outils de configuration du noyau qui doivent être exécutés avant qu'un support graphique ne soit disponible.

La bibliothèque *curses* propose des fonctionnalités assez basiques, fournissant au programmeur une abstraction d'affichage contenant plusieurs fenêtres de texte qui ne se chevauchent pas. Le contenu d'une fenêtre peut être modifié de différentes manières — en ajoutant du texte, en l'effaçant ou en changeant son apparence — et la bibliothèque *curses* trouve quels codes de contrôle doivent être envoyés au terminal pour produire le bon résultat. *curses* ne fournit pas beaucoup de concepts d'interface utilisateur tels que boutons, cases à cocher ou dialogues ; si vous avez besoin de telles fonctionnalités, pensez à une bibliothèque d'interface utilisateur comme [Urwid](#).

La bibliothèque *curses* a été écrite à l'origine pour BSD Unix ; les dernières versions *System V* d'Unix d'AT&T ont ajouté de nombreuses améliorations et de nouvelles fonctions. BSD *curses* n'est plus maintenu, ayant été remplacé par *ncurses*, qui est une implémentation open-source de l'interface AT&T. Si vous utilisez un Unix open-source comme Linux ou FreeBSD, votre système utilise presque certainement *ncurses*. Comme la plupart des versions commerciales actuelles d'Unix sont basées sur le code *System V*, toutes les fonctions décrites ici seront probablement disponibles. Les anciennes versions de *curses* portées par certains Unix propriétaires pourraient ne pas gérer toutes les fonctions.

La version Windows de Python n'inclut pas le module *curses*. Une version portée appelée [UniCurses](#) est disponible.

## 1.1 Le module *curses* de Python

The Python module is a fairly simple wrapper over the C functions provided by *curses* ; if you're already familiar with *curses* programming in C, it's really easy to transfer that knowledge to Python. The biggest difference is that the Python interface makes things simpler by merging different C functions such as `addstr()`, `mvaddstr()`, and `mvwaddstr()` into a single `addstr()` method. You'll see this covered in more detail later.

Ce guide pratique est une introduction à l'écriture de programmes en mode texte avec *curses* et Python. Il n'essaie pas d'être un guide complet de l'API *curses* ; pour cela, consultez la section du guide de la bibliothèque Python sur *ncurses* et les pages du manuel C pour *ncurses*. Il vous donne cependant les idées de base.

## 2 Lancement et arrêt une application *curses*

Before doing anything, *curses* must be initialized. This is done by calling the `initscr()` function, which will determine the terminal type, send any required setup codes to the terminal, and create various internal data structures. If successful, `initscr()` returns a window object representing the entire screen ; this is usually called `stdscr` after the name of the corresponding C variable.

```
import curses
stdscr = curses.initscr()
```

Habituellement, les applications *curses* désactivent l'écho automatique des touches à l'écran, afin de pouvoir lire les touches et ne les afficher que dans certaines circonstances. Cela nécessite d'appeler la fonction `noecho()`.

```
curses.noecho()
```

Également, les applications réagissent généralement instantanément aux touches sans qu'il soit nécessaire d'appuyer sur la touche Entrée ; c'est ce qu'on appelle le mode *cbreak*, par opposition au mode d'entrée habituel avec un tampon.

```
curses.cbreak()
```

Les terminaux renvoient généralement les touches spéciales, telles que les touches de curseur ou les touches de navigation (Page précédente et Accueil par exemple), comme une séquence d'échappement sur plusieurs octets. Bien que vous puissiez écrire votre application pour vous attendre à de telles séquences et les traiter en conséquence, *curses* peut le faire pour vous, renvoyant une valeur spéciale telle que `curses.KEY_LEFT`. Pour que *curses* fasse le travail, vous devez activer le mode *keypad*.

```
stdscr.keypad(True)
```

Arrêter une application *curses* est beaucoup plus facile que d'en démarrer une. Appelez :

```
curses.nocbreak()
stdscr.keypad(False)
curses.echo()
```

pour inverser les réglages du terminal mis en place pour *curses*. Ensuite, appelez la fonction `enddwinn()` pour restaurer le terminal dans son mode de fonctionnement original.

```
curses.endwin()
```

Un problème courant lors du débogage d'une application *curses* est de se retrouver avec un terminal sans queue ni tête lorsque l'application meurt sans restaurer le terminal à son état précédent. Avec Python, cela arrive souvent lorsque votre code est bogué et lève une exception non interceptée. Les touches ne sont plus répétées à l'écran lorsque vous les tapez, par exemple, ce qui rend l'utilisation de l'interface de commande du *shell* difficile.

En Python, vous pouvez éviter ces complications et faciliter le débogage en important la fonction `curses.wrapper()` et en l'utilisant comme suit :

```
from curses import wrapper

def main(stdscr):
    # Clear screen
    stdscr.clear()

    # This raises ZeroDivisionError when i == 10.
    for i in range(0, 11):
        v = i-10
        stdscr.addstr(i, 0, '10 divided by {} is {}'.format(v, 10/v))

    stdscr.refresh()
    stdscr.getkey()

wrapper(main)
```

The `wrapper()` function takes a callable object and does the initializations described above, also initializing colors if color support is present. `wrapper()` then runs your provided callable. Once the callable returns, `wrapper()` will restore the original state of the terminal. The callable is called inside a `try...except` that catches exceptions, restores the state of the terminal, and then re-raises the exception. Therefore your terminal won't be left in a funny state on exception and you'll be able to read the exception's message and traceback.

### 3 Fenêtres et tampons (*pads* en anglais)

Les fenêtres sont l'abstraction de base de *curses*. Un objet fenêtre représente une zone rectangulaire de l'écran qui gère des méthodes pour afficher du texte, l'effacer, permettre à l'utilisateur de saisir des chaînes, etc.

L'objet `stdscr` renvoyé par la fonction `initscr()` est un objet fenêtre qui couvre l'écran entier. De nombreux programmes peuvent n'avoir besoin que de cette fenêtre unique, mais vous pouvez diviser l'écran en fenêtres plus petites, afin de les redessiner ou de les effacer séparément. La fonction `newwin()` crée une nouvelle fenêtre d'une taille donnée, renvoyant le nouvel objet fenêtre.

```
begin_x = 20; begin_y = 7
height = 5; width = 40
win = curses.newwin(height, width, begin_y, begin_x)
```

Notez que le système de coordonnées utilisé dans *curses* est inhabituel. Les coordonnées sont toujours passées dans l'ordre *y,x* et le coin supérieur gauche d'une fenêtre a pour coordonnées (0,0). Ceci rompt la convention normale des coordonnées où la coordonnée *x* vient en premier. C'est une différence malheureuse par rapport à la plupart des autres

applications informatiques, mais elle fait partie de *curses* depuis qu'il a été écrit et il est trop tard pour changer les choses maintenant.

Votre application peut déterminer la taille de l'écran en utilisant les variables `curses.LINES` et `curses.COLS` pour obtenir les tailles *y* et *x*. Les coordonnées licites s'étendent alors de `(0,0)` à `(curses.LINES - 1, curses.COLS - 1)`.

Quand vous appelez une méthode pour afficher ou effacer du texte, l'affichage ne le reflète pas immédiatement. Vous devez appeler la méthode `refresh()` des objets fenêtre pour mettre à jour l'écran.

This is because *curses* was originally written with slow 300-baud terminal connections in mind; with these terminals, minimizing the time required to redraw the screen was very important. Instead *curses* accumulates changes to the screen and displays them in the most efficient manner when you call `refresh()`. For example, if your program displays some text in a window and then clears the window, there's no need to send the original text because they're never visible.

In practice, explicitly telling *curses* to redraw a window doesn't really complicate programming with *curses* much. Most programs go into a flurry of activity, and then pause waiting for a keypress or some other action on the part of the user. All you have to do is to be sure that the screen has been redrawn before pausing to wait for user input, by first calling `stdscr.refresh()` or the `refresh()` method of some other relevant window.

Un tampon (*pad* en anglais) est une forme spéciale de fenêtre; il peut être plus grand que l'écran effectif et il est possible de n'afficher qu'une partie du tampon à la fois. La création d'un tampon nécessite de fournir sa hauteur et sa largeur, tandis que pour le rafraîchissement du tampon, vous devez fournir les coordonnées de la zone de l'écran où une partie du tampon sera affichée.

```
pad = curses.newpad(100, 100)
# These loops fill the pad with letters; addch() is
# explained in the next section
for y in range(0, 99):
    for x in range(0, 99):
        pad.addch(y,x, ord('a') + (x*x+y*y) % 26)

# Displays a section of the pad in the middle of the screen.
# (0,0) : coordinate of upper-left corner of pad area to display.
# (5,5) : coordinate of upper-left corner of window area to be filled
#         with pad content.
# (20, 75) : coordinate of lower-right corner of window area to be
#           : filled with pad content.
pad.refresh( 0,0, 5,5, 20,75)
```

The `refresh()` call displays a section of the pad in the rectangle extending from coordinate `(5,5)` to coordinate `(20,75)` on the screen; the upper left corner of the displayed section is coordinate `(0,0)` on the pad. Beyond that difference, pads are exactly like ordinary windows and support the same methods.

If you have multiple windows and pads on screen there is a more efficient way to update the screen and prevent annoying screen flicker as each part of the screen gets updated. `refresh()` actually does two things :

- 1) elle appelle la méthode `noutrefresh()` de chaque fenêtre pour mettre à jour les données sous-jacentes qui permettent d'obtenir l'affichage voulu;
- 2) elle appelle la fonction `doupdate()` pour modifier l'écran physique afin de correspondre à l'état défini par les données sous-jacentes.

Instead you can call `noutrefresh()` on a number of windows to update the data structure, and then call `doupdate()` to update the screen.

## 4 Affichage de texte

From a C programmer's point of view, *curses* may sometimes look like a twisty maze of functions, all subtly different. For example, `addstr()` displays a string at the current cursor location in the `stdscr` window, while `mvaddstr()` moves to a given *y,x* coordinate first before displaying the string. `waddstr()` is just like `addstr()`, but allows specifying a window to use instead of using `stdscr` by default. `mvwaddstr()` allows specifying both a window and a coordinate.

Heureusement, l'interface Python masque tous ces détails. `stdscr` est un objet fenêtre comme les autres et les méthodes telles que `addstr()` acceptent leurs arguments sous de multiples formes, habituellement quatre.

Forme	Description
<i>str</i> ou <i>ch</i>	Affiche la chaîne <i>str</i> ou le caractère <i>ch</i> à la position actuelle
<i>str</i> ou <i>ch</i> , <i>attr</i>	Affiche la chaîne <i>str</i> ou le caractère <i>ch</i> , en utilisant l'attribut <i>attr</i> à la position actuelle
<i>y</i> , <i>x</i> , <i>str</i> ou <i>ch</i>	Se déplace à la position <i>y,x</i> dans la fenêtre et affiche la chaîne <i>str</i> ou le caractère <i>ch</i>
<i>y</i> , <i>x</i> , <i>str</i> ou <i>ch</i> , <i>attr</i>	Se déplace à la position <i>y,x</i> dans la fenêtre et affiche la chaîne <i>str</i> ou le caractère <i>ch</i> en utilisant l'attribut <i>attr</i>

Les attributs permettent de mettre en valeur du texte : gras, souligné, mode vidéo inversé ou en couleur. Nous les voyons plus en détail dans la section suivante.

The `addstr()` method takes a Python string or bytestring as the value to be displayed. The contents of bytestrings are sent to the terminal as-is. Strings are encoded to bytes using the value of the window's `encoding` attribute; this defaults to the default system encoding as returned by `locale.getencoding()`.

Les méthodes `addch()` prennent un caractère, soit sous la forme d'une chaîne de longueur 1, d'une chaîne d'octets de longueur 1 ou d'un entier.

Des constantes sont disponibles pour étendre les caractères; ces constantes sont des entiers supérieurs à 255. Par exemple, `ACS_PLMINUS` correspond au symbole +/- et `ACS_ULCORNER` correspond au coin en haut et à gauche d'une boîte (utile pour dessiner des encadrements). Vous pouvez aussi utiliser les caractères Unicode adéquats.

Windows se souvient de l'endroit où le curseur était positionné lors de la dernière opération, de manière à ce que si vous n'utilisez pas les coordonnées *y,x*, l'affichage se produit au dernier endroit utilisé. Vous pouvez aussi déplacer le curseur avec la méthode `move(y, x)`. Comme certains terminaux affichent un curseur clignotant, vous pouvez ainsi vous assurer que celui-ci est positionné à un endroit où il ne distrait pas l'utilisateur (il peut être déroutant d'avoir un curseur qui clignote à des endroits apparemment aléatoires).

Si votre application n'a pas besoin d'un curseur clignotant, vous pouvez appeler `curs_set(False)` pour le rendre invisible. Par souci de compatibilité avec les anciennes versions de *curses*, il existe la fonction `leaveok(bool)` qui est un synonyme de `curs_set()`. Quand *bool* vaut `True`, la bibliothèque *curses* essaie de supprimer le curseur clignotant et vous n'avez plus besoin de vous soucier de le laisser trainer à des endroits bizarres.

### 4.1 Attributs et couleurs

Les caractères peuvent être affichés de différentes façons. Les lignes de statut des applications en mode texte sont généralement affichées en mode vidéo inversé; vous pouvez avoir besoin de mettre en valeur certains mots. À ces fins, *curses* vous permet de spécifier un attribut pour chaque caractère à l'écran.

Un attribut est un entier dont chaque bit représente un attribut différent. Vous pouvez essayer d'afficher du texte avec plusieurs attributs définis simultanément mais *curses* ne garantit pas que toutes les combinaisons soient prises en compte ou que le résultat soit visuellement différent. Cela dépend de la capacité de chaque terminal utilisé, il est donc plus sage de se cantonner aux attributs les plus communément utilisés, dont la liste est fournie ci-dessous.

Attribut	Description
A_BLINK	Texte clignotant
A_BOLD	Texte en surbrillance ou en gras
A_DIM	Texte en demi-ton
A_REVERSE	Texte en mode vidéo inversé
A_STANDOUT	Le meilleur mode de mise en valeur pour le texte
A_UNDERLINE	Texte souligné

Ainsi, pour mettre la ligne de statut située en haut de l'écran en mode vidéo inversé, vous pouvez coder :

```
stdscr.addstr(0, 0, "Current mode: Typing mode",
              curses.A_REVERSE)
stdscr.refresh()
```

La bibliothèque *curses* gère également les couleurs pour les terminaux compatibles. Le plus répandu de ces terminaux est sûrement la console Linux, suivie par *xterm* en couleurs.

Pour utiliser les couleurs, vous devez d'abord appeler la fonction `start_color()` juste après avoir appelé `initscr()` afin d'initialiser (la fonction `curses.wrapper()` le fait automatiquement). Ensuite, la fonction `has_colors()` renvoie `True` si le terminal utilisé gère les couleurs (note : *curses* utilise l'orthographe américaine *color* et non pas l'orthographe britannique ou canadienne *colour* ; si vous êtes habitué à l'orthographe britannique, vous devrez vous résigner à mal l'orthographier tant que vous utilisez *curses*).

La bibliothèque *curses* maintient un nombre restreint de paires de couleurs, constituées d'une couleur de texte (*foreground*) et de fond (*background*). Vous pouvez obtenir la valeur des attributs correspondant à une paire de couleur avec la fonction `color_pair()` ; cette valeur peut être combinée bit par bit (avec la fonction *OR*) avec les autres attributs tels que `A_REVERSE`, mais là encore, de telles combinaisons risquent de ne pas fonctionner sur tous les terminaux.

Un exemple d'affichage d'une ligne de texte en utilisant la paire de couleur 1 :

```
stdscr.addstr("Pretty text", curses.color_pair(1))
stdscr.refresh()
```

Comme indiqué auparavant, une paire de couleurs est constituée d'une couleur de texte et d'une couleur de fond. La fonction `init_pair(n, f, b)` change la définition de la paire de couleurs *n*, en définissant la couleur de texte à *f* et la couleur de fond à *b*. La paire de couleurs 0 est codée en dur à blanc sur noir et ne peut être modifiée.

Les couleurs sont numérotées et `start_color()` initialise 8 couleurs basiques lors de l'activation du mode en couleurs. Ce sont : 0 pour noir (*black*), 1 pour rouge (*red*), 2 pour vert (*green*), 3 pour jaune (*yellow*), 4 pour bleu (*blue*), 5 pour magenta, 6 pour cyan et 7 pour blanc (*white*). Le module *curses* définit des constantes nommées pour chacune de ces couleurs : `curses.COLOR_BLACK`, `curses.COLOR_RED` et ainsi de suite.

Testons tout ça. Pour changer la couleur 1 à rouge sur fond blanc, appelez :

```
curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
```

Quand vous modifiez une paire de couleurs, tout le texte déjà affiché qui utilise cette paire de couleur voit les nouvelles couleurs s'appliquer à lui. Vous pouvez aussi afficher du nouveau texte dans cette couleur avec :

```
stdscr.addstr(0,0, "RED ALERT!", curses.color_pair(1))
```

Les terminaux « de luxe » peuvent définir les couleurs avec des valeurs *RGB*. Cela vous permet de modifier la couleur 1, habituellement rouge, en violet ou bleu voire toute autre couleur selon votre goût. Malheureusement, la console Linux ne gère pas cette fonctionnalité, je suis donc bien incapable de la tester et de vous en fournir un exemple. Vous pouvez vérifier si votre terminal la prend en charge en appelant `can_change_color()`, qui renvoie `True` en cas de succès. Si vous avez la chance d'avoir un terminal aussi perfectionné, consultez les pages du manuel de votre système pour obtenir plus d'informations.

## 5 Entrées de l'utilisateur

La bibliothèque C *curses* ne propose que quelques mécanismes très simples pour les entrées. Le module *curses* y ajoute un *widget* basique d'entrée de texte (d'autres bibliothèques telles que *Urwid* ont un ensemble de *widgets* plus conséquent).

Il y a deux méthodes pour obtenir des entrées dans une fenêtre :

- `getch()` rafraîchit l'écran et attend que l'utilisateur appuie sur une touche, affichant cette touche si `echo()` a été appelé auparavant. Vous pouvez en option spécifier des coordonnées où positionner le curseur avant la mise en pause ;
- `getkey()` effectue la même chose mais convertit l'entier en chaîne. Les caractères individuels sont renvoyés en chaînes de longueur 1 alors que les touches spéciales (telles que les touches de fonction) renvoient des chaînes plus longues contenant le nom de la touche (tel que `KEY_UP` ou `^G`).

It's possible to not wait for the user using the `nodelay()` window method. After `nodelay(True)`, `getch()` and `getkey()` for the window become non-blocking. To signal that no input is ready, `getch()` returns `curses.ERR` (a value of -1) and `getkey()` raises an exception. There's also a `halfdelay()` function, which can be used to (in effect) set a timer on each `getch()` ; if no input becomes available within a specified delay (measured in tenths of a second), *curses* raises an exception.

The `getch()` method returns an integer ; if it's between 0 and 255, it represents the ASCII code of the key pressed. Values greater than 255 are special keys such as Page Up, Home, or the cursor keys. You can compare the value returned to constants such as `curses.KEY_PPAGE`, `curses.KEY_HOME`, or `curses.KEY_LEFT`. The main loop of your program may look something like this :

```
while True:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break # Exit the while loop
    elif c == curses.KEY_HOME:
        x = y = 0
```

Le module `curses.ascii` fournit des fonctions pour déterminer si l'entier ou la chaîne de longueur 1 passés en arguments font partie de la classe ASCII ; elles peuvent s'avérer utile pour écrire du code plus lisible dans ce genre de boucles. Il fournit également des fonctions de conversion qui prennent un entier ou une chaîne de longueur 1 en entrée et renvoient le type correspondant au nom de la fonction. Par exemple, `curses.ascii.ctrl()` renvoie le caractère de contrôle correspondant à son paramètre.

Il existe aussi une méthode pour récupérer une chaîne entière, `getstr()`. Elle n'est pas beaucoup utilisée car son utilité est limitée : les seules touches d'édition disponibles sont le retour arrière et la touche Entrée, qui termine la chaîne. Elle peut, en option, être limitée à un nombre fixé de caractères.

```
curses.echo() # Enable echoing of characters

# Get a 15-character string, with the cursor on the top line
s = stdscr.getstr(0,0, 15)
```

Le module `curses.textpad` fournit un type de boîte texte qui gère des touches de fonctions à la façon d'*Emacs*. Plusieurs méthodes de la classe `Textbox` gèrent l'édition avec la validation des entrées et le regroupement de l'entrée avec ou sans les espaces de début et de fin. Par exemple :

```
import curses
from curses.textpad import Textbox, rectangle

def main(stdscr):
    stdscr.addstr(0, 0, "Enter IM message: (hit Ctrl-G to send)")

    editwin = curses.newwin(5,30, 2,1)
    rectangle(stdscr, 1,0, 1+5+1, 1+30+1)
```

(suite sur la page suivante)

```

stdscr.refresh()

box = Textbox(editwin)

# Let the user edit until Ctrl-G is struck.
box.edit()

# Get resulting contents
message = box.gather()

```

Consultez la documentation de la bibliothèque pour plus de détails sur `curses.textpad`.

## 6 Pour aller plus loin

Ce guide pratique ne couvre pas certains sujets avancés, tels que la lecture du contenu de l'écran ou la capture des événements relatifs à la souris dans une instance *xterm*, mais la page de la bibliothèque Python du module `curses` est maintenant suffisamment complète. Nous vous encourageons à la parcourir.

If you're in doubt about the detailed behavior of the `curses` functions, consult the manual pages for your `curses` implementation, whether it's `ncurses` or a proprietary Unix vendor's. The manual pages will document any quirks, and provide complete lists of all the functions, attributes, and ACS\_\* characters available to you.

Étant donné que l'API `curses` est si volumineuse, certaines fonctions ne sont pas prises en charge dans l'interface Python. Souvent, ce n'est pas parce qu'elles sont difficiles à implémenter, mais parce que personne n'en a eu encore besoin. De plus, Python ne prend pas encore en charge la bibliothèque de gestion des menus associée à `ncurses`. Les correctifs ajoutant cette prise en charge seraient bienvenus ; reportez-vous au [guide du développeur Python](#) pour apprendre comment soumettre des améliorations à Python.

- [Writing Programs with NCURSES](#) : a lengthy tutorial for C programmers.
- [La page de manuel ncurses](#)
- [The ncurses FAQ](#)
- ["Use curses... don't swear"](#) : vidéo d'une conférence lors de la PyCon 2013 sur la gestion des terminaux à l'aide de `curses` et `Urwid` (vidéo en anglais).
- ["Console Applications with Urwid"](#) : video of a PyCon CA 2012 talk demonstrating some applications written using `Urwid`.