Urwid Documentation

Release 1.1.2

Ian Ward

February 09, 2014

Contents

CHAPTER 1

Changelog

1.1 Urwid 1.1.2

2013-12-30

- Move to urwid.org and use sphinx docs for generating whole site, move changelog to docs/changelog.rst
- Fix encoding exceptions when unicode used on non-UTF-8 terminal
- Fix for suspend and resume applications with ^Z
- Fix for tmux and screen missing colors on right bug
- Fix Pile zero-weighted items and mouse_event when empty
- Fix Terminal select() not retrying when interrupted by signal
- Fix for Padding.align and width change not invalidating

1.2 Urwid 1.1.1

2012-11-15

- Fix for Pile not changing focus on mouse events
- Fix for Overlay.get_cursor_coords()

1.3 Urwid 1.1.0

2012-10-23

- New common container API: focus, focus_position, contents, options(), get_focus_path(), set_focus_path(), __getitem__, __iter__(), __reversed__() implemented across all included container widgets

  A full description doesn't fit here, see the Container Widgets section in the manual for details
- New Sphinx-based documentation now included in source: Tutorial rewritten, manual revised and new reference based on updated docstrings (by Marco Giusti, Patrick Totzke)
- New list walker SimpleFocusListWalker like SimpleListWalker but updates focus position as items are inserted or removed
- New decoration widget WidgetDisable to disable interaction with the widgets it wraps

- SelectableIcon selectable text widget used by button widgets is now documented (available since 0.9.9)
- Columns widget now tries to keep column in focus visible, hiding columns on the left when necessary
- Padding widget now defaults to ('relative', 100) instead of 'pack' so that left and right parameters are more useful and more child widgets are supported
- New list walker "API Version 2" that is simpler for many list walker uses; "API Version 1" will still continue to be supported
- List walkers may now allow iteration from the absolute top or bottom of the list if they provide a positions() method
- raw_display now erases to the end of the line with EL escape sequence to improve copy+paste behavior for some terminals
- Filler now has top and bottom parameters like Padding's left and right parameters and accepts 'pack' instead of None as a height value for widgets that calculate their own number of rows
- Pile and Columns now accepts 'pack' instead of 'flow' for widgets that calculate their own number of rows or columns
- Pile and Columns now accept 'given' instead of 'fixed' for cases where the number of rows or columns are specified by the container options
- Pile and Columns widgets now accept any iterable to their __init__() methods
- Widget now has a default focus_position property that raises an IndexError when read to be consistent with new common container API
- GridFlow now supports multiple cell widths within the same widget
- BoxWidget, FlowWidget and FixedWidget are deprecated, instead use the sizing() function or _sizing attribute to specify the supported sizing modes for your custom widgets
- Some new shift+arrow and numpad input sequences from RXVT and xterm are now recognized
- Fix for alarms when used with a screen event loop (e.g. curses_display)
- Fix for raw_display when terminal width is 1 column
- Fixes for a Columns.get_cursor_coords() regression and a SelectableIcon.get_cursor_coords() bug
- Fixes for incorrect handling of box columns in a number of Columns methods when that column is selectable
- Fix for Terminal widget input handling with Python 3

1.4 Urwid 1.0.3

2012-11-15

- Fix for alarms when used with a screen event loop (e.g. curses_display)
- Fix for Overlay.get_cursor_coords()

1.5 Urwid 1.0.2

2012-07-13

- Fix for bug when entering Unicode text into Edit widget with bytes caption
- Fix a regression when not running in UTF-8 mode

- Fix for a MainLoop.remove_watch_pipe() bug
- Fix for a bug when packing empty Edit widgets
- Fix for a ListBox "contents too long" error with very large Edit widgets
- Prevent ListBoxes from selecting 0-height selectable widgets when moving up or down
- Fix a number of bugs caused by 0-height widgets in a ListBox

1.6 Urwid 1.0.1

2011-11-28

- Fix for Terminal widget in BSD/OSX
- Fix for a Filler mouse_event() position bug
- Fix support for mouse positions up to x=255, y=255
- Fixes for a number of string encoding issues under Python 3
- Fix for a LineBox border __init__() parameters
- Fix input input of UTF-8 in tour.py example by converting captions to unicode
- Fix tutorial examples' use of TextCanvas and switch to using unicode literals
- Prevent raw_display from calling tcseattr() or tcgetattr() on non-ttys
- Disable curses_display external event loop support: screen resizing and gpm events are not properly supported
- Mark PollingListWalker as deprecated

1.7 Urwid 1.0.0

2011-09-22

- New support for Python 3.2 from the same 2.x code base, requires distribute instead of setuptools (by Kirk McDonald, Wendell, Marien Zwart) everything except TwistedEventLoop and GLibEventLoop is supported
- New experimental Terminal widget with xterm emulation and terminal.py example program (by aszlig)
- Edit widget now supports a mask (for passwords), has a insert_text_result() method for full-field validation and normalizes input text to Unicode or bytes based on the caption type used
- New TreeWidget, TreeNode, ParentNode, TreeWalker and TreeListBox classes for lazy expanding/collapsing tree views factored out of browse.py example program, with new treesample.py example program (by Rob Lanphier)
- MainLoop now calls draw_screen() just before going idle, so extra calls to draw_screen() in user code may now be removed
- New MainLoop.watch_pipe() method for subprocess or threaded communication with the process/thread updating the UI, and new subproc.py example demonstrating its use
- New PopUpLauncher and PopUpTarget widgets and MainLoop option for creating pop-ups and drop-downs, and new pop_up.py example program
- New twisted_serve_ssh.py example (by Ali Afshar) that serves multiple displays over ssh from the same application using Twisted and the TwistedEventLoop

- ListBox now includes a get_cursor_coords() method, allowing nested ListBox widgets
- Columns widget contents may now be marked to always be treated as flow widgets for mixing flow and box widgets more easily
- New lcd_display module with support for CF635 USB LCD panel and lcd_cf635.py example program with menus, slider controls and a custom font
- Shared command_map instance is now stored as Widget._command_map class attribute and may be overridden in subclasses or individual widgets for more control over special keystrokes
- Overlay widget parameters may now be adjusted after creation with set_overlay_parameters() method
- New WidgetPlaceholder widget useful for swapping widgets without having to manipulate a container widget's contents
- LineBox widgets may now include title text
- ProgressBar text content and alignment may now be overridden
- Use reactor.stop() in TwistedEventLoop and document that Twisted's reactor is not designed to be stopped then restarted
- curses_display now supports AttrSpec and external event loops (Twisted or GLib) just like raw_display
- raw_display and curses_display now support the IBMPC character set (currently only used by Terminal widget)
- Fix for a gpm_mev bug preventing user input when on the console
- Fix for leaks of None objects in str_util extension
- Fix for WidgetWrap and AttrMap not working with fixed widgets
- Fix for a lock up when attempting to wrap text containing wide characters into a single character column

1.8 Urwid 0.9.9.2

2011-07-13

- Fix for an Overlay get_cursor_coords(), and Text top-widget bug
- Fix for a Padding rows() bug when used with width=PACK
- Fix for a bug with large flow widgets used in an Overlay
- Fix for a gpm_mev bug
- Fix for Pile and GraphVScale when rendered with no contents
- Fix for a Python 2.3 incompatibility (0.9.9 is the last release to claim support Python 2.3)

1.9 Urwid 0.9.9.1

2010-01-25

- Fix for ListBox snapping to selectable widgets taller than the ListBox itself
- raw_display switching to alternate buffer now works properly with Terminal.app
- Fix for BoxAdapter backwards incompatibility introduced in 0.9.9
- Fix for a doctest failure under powerpc

- Fix for systems with gpm_mev installed but not running gpm

1.10 Urwid 0.9.9

2009-11-15

- New support for 256 and 88 color terminals with raw_display and html_fragment display modules
- New palette_test example program to demonstrate high color modes
- New AttrSpec class for specifying specific colors instead of using attributes defined in the screen's palette
- New MainLoop class ties together widgets, user input, screen display and one of a number of new event loops, removing the need for tedious, error-prone boilerplate code
- New GLibEventLoop allows running Urwid applications with GLib (makes D-Bus integration easier)
- New TwistedEventLoop allows running Urwid with a Twisted reactor
- Added new docstrings and doctests to many widget classes
- New AttrMap widget supports mapping any attribute to any other attribute, replaces AttrWrap widget
- New WidgetDecoration base class for AttrMap, BoxAdapter, Padding, Filler and LineBox widgets creates a common method for accessing and updating their contained widgets
- New left and right values may be specified in Padding widgets
- New command_map for specifying which keys cause actions such as clicking Button widgets and scrolling ListBox widgets
- New tty_signal_keys() method of raw_display.Screen and curses_display.Screen allows changing or disabling the keys used to send signals to the application
- Added helpful __repr__ for many widget classes
- Updated all example programs to use MainLoop class
- Updated tutorial with MainLoop usage and improved examples
- Renamed WidgetWrap.w to _w, indicating its intended use as a way to implement a widget with other widgets, not necessarily as a container for other widgets
- Replaced all tabs with 4 spaces, code is now more aerodynamic (and PEP 8 compliant)
- Added saving of stdin and stdout in raw_display module allowing the originals to be redirected
- Updated BigText widget's HalfBlock5x4Font
- Fixed graph example CPU usage when animation is stopped
- Fixed a memory leak related to objects listening for signals
- Fixed a Popen3 deprecation warning

1.11 Urwid 0.9.8.4

2009-03-13

- Fixed incompatibilities with Python 2.6 (by Friedrich Weber)
- Fixed a SimpleListWalker with emptied list bug (found by Walter Mundt)

- Fixed a curses_display stop()/start() bug (found by Christian Scharkus)
- Fixed an is_wide_character() segfault on bad input data bug (by Andrew Psaltis)
- Fixed a CanvasCache with render() used in both a widget and its superclass bug (found by Andrew Psaltis)
- Fixed a ListBox.ends_visible() on empty list bug (found by Marc Hartstein)
- Fixed a tutorial example bug (found by Kurtis D. Rader)
- Fixed an Overlay.keypress() bug (found by Andreas Klöckner)
- Fixed setuptools configuration (by Andreas Klöckner)

1.12 Urwid 0.9.8.3

2008-07-14

- Fixed a canvas cache memory leak affecting 0.9.8, 0.9.8.1 and 0.9.8.2 (found by John Goodfellow)
- Fixed a canvas fill_attr() bug (found by Joern Koerner)

1.13 Urwid 0.9.8.2

2008-05-19

- Fixed incompatibilities with Python 2.3
- Fixed Pile cursor pref_col bug, WidgetWrap rows caching bug, Button mouse_event with no callback bug, Filler body bug triggered by the tutorial and a LineBox lline parameter typo.

1.14 Urwid 0.9.8.1

2007-06-21

- Fixed a Filler render() bug, a raw_display start()/stop() bug and a number of problems triggered by very small terminal window sizes.

1.15 Urwid 0.9.8

2007-03-23

- Rendering is now significantly faster.
- New Widget base class for all widgets. It includes automatic caching of rows() and render() methods. It also adds a new __super attribute for accessing methods in superclasses.

  Widgets must now call self._invalidate() to notify the cache when their content has changed.

  To disable caching in a widget set the class variable no_cache to a list that includes the string "render".
- Canvas classes have been reorganized: Canvas has been renamed to TextCanvas and Canvas is now the base class for all canvases. New canvas classes include BlankCanvas, SolidCanvas and CompositeCanvas.
- External event loops may now be used with the raw_display module. The new methods get_input_descriptors() and get_input_nonblocking() should be used instead of get_input() to allow input processing without blocking.

- The Columns, Pile and ListBox widgets now choose their first selectable child widget as the focus widget by defaut.
- New ListWalker base class for list walker classes.
- New Signals class that will be used to improve the existing event callbacks. Currently it is used for ListWalker objects to notify their ListBox when their content has changed.
- SimpleListWalker now behaves as a list and supports all list operations. This class now detects when changes are made to the list and notifies the ListBox object. New code should use this class to wrap lists of widgets before passing them to the ListBox constructor.
- New PollingListWalker class is now the default list walker that is used when passing a simple list to the ListBox constructor. This class is intended for backwards compatibility only. When this class is used the ListBox object is unable to cache its render() method.
- The curses_display module can now draw in the lower-right corner of the screen.
- All display modules now have start() and stop() methods that may be used instead of calling run_wrapper().
- The raw_display module now uses an alternate buffer so that the original screen can be restored on exit. The old behaviour is available by seting the alternate_buffer parameter of start() or run_wrapper() to False.
- Many internal string processing functions have been rewritten in C to improve their performance.
- Compatible with Python >= 2.2. Python 2.1 is no longer supported.

1.16 Urwid 0.9.7.2

2007-01-03

- Improved performance in UTF-8 mode when ASCII text is used.
- Fixed a UTF-8 input bug.
- Added a clear() function to the the display modules to force the screen to be repainted on the next draw_screen() call.

1.17 Urwid 0.9.7.1

2006-10-03

- Fixed bugs in Padding and Overlay widgets introduced in 0.9.7.

1.18 Urwid 0.9.7

2006-10-01

- Added initial support for fixed widgets - widgets that have a fixed size on screen. Fixed widgets expect a size parameter equal to (). Fixed widgets must implement the pack(..) function to return their size.
- New BigText class that draws text with fonts made of grids of character cells. BigText is a fixed widget and doesn't do any alignment or wrapping. It is intended for banners and number readouts that need to stand out on the screen.

  Fonts: Thin3x3Font, Thin4x3Font, Thin6x6Font (full ascii)

UTF-8 only fonts: HalfBlock5x4Font, HalfBlock6x5Font, HalfBlockHeavy6x5Font, HalfBlock7x7Font (full ascii)

New function get_all_fonts() may be used to get a list of the available fonts.

- New example program bigtext.py demonstrates use of BigText.
- Padding class now has a clipping mode that pads or clips fixed widgets to make them behave as flow widgets.
- Overlay class can now accept a fixed widget as the widget to display "on top".
- New Canvas functions: pad_trim() and pad_trim_left_right().
- Fixed a bug in Filler.get_cursor_coords() that causes a crash if the contained widget's get_cursor_coords() function returns None.
- Fixed a bug in Text.pack() that caused an infinite loop when the text contained a newline. This function is not currently used by Urwid.
- Edit.__init__() now calls set_edit_text() to initialize its text.
- Overlay.calculate_padding_filler() and Padding.padding_values() now include focus parameters.

1.19 Urwid 0.9.6

2006-08-22

- Fixed Unicode conversion and locale issues when using Urwid with Python < 2.4. The graph.py example program should now work properly with older versions of Python.
- The docgen_tutorial.py script can now write out the tutorial example programs as individual files.
- Updated reference documentation table of contents to show which widgets are flow and/or box widgets.
- Columns.set_focus(..) will now accept an integer or a widget as its parameter.
- Added detection for rxvt's HOME and END escape sequences.
- Added support for setuptools (improved distutils).

1.20 Urwid 0.9.5

2006-06-14

- Some Unicode characters are now converted to use the G1 alternate character set with DEC special and line drawing characters. These Unicode characters should now "just work" in almost all terminals and encodings.

  When Urwid is run with the UTF-8 encoding the characters are left as UTF-8 and not converted.

  The characters converted are:

  u00A3 (£), u00B0 (°), u00B1 (±), u00B7 (·), u03C0 (π), u2260 (), u2264 (), u2265 (), u23ba (), u23bb (), u23bc (), u23bd (_), u2500 (-), u2502 (|), u250c (), u2510 (), u2514 (), u2518 (), u251c (), u2524 (), u252c (), u2534 (), u253c (), u2592 (), u25c6 ()
- New SolidFill class for filling an area with a single character.
- New LineBox class for wrapping widgets in a box made of line- drawing characters. May be used as a box widget or a flow widget.
- New example program graph.py demonstrates use of BarGraph, LineBox, ProgressBar and SolidFill.

- Pile class may now be used as a box widget and contain a mix of box and flow widgets.
- Columns class may now contain a mix of box and flow widgets. The box widgets will take their height from the maximum height of the flow widgets.
- Improved the smoothness of resizing with raw_display module. The module will now try to stop updating the screen when a resize event occurs during the update.
- The Edit and IntEdit classes now use their set_edit_text() and set_edit_pos() functions when handling keypresses, so those functions may be overridden to catch text modification.
- The set_state() functions in the CheckBox and RadioButton classes now have a do_callback parameter that determines if the callback function registered will be called.
- Fixed a newly introduced incompatibility with python < 2.3.
- Fixed a missing symbol in curses_display when python is linked against libcurses.
- Fixed mouse handling bugs in the Frame and Overlay classes.
- Fixed a Padding bug when the left or right has no padding.

1.21 Urwid 0.9.4

2006-05-30

- Enabled mouse handling across the Urwid library.

  Added a new mouse_event() method to the Widget interface definition and to the following widgets: Edit, CheckBox, RadioButton, Button, GridFlow, Padding, Filler, Overlay, Frame, Pile, Columns, BoxAdapter and ListBox.

  Updated example programs browse.py, calc.py, dialog.py, edit.py and tour.py to support mouse input.
- Released the files used to generate the reference and tutorial documentation: docgen_reference.py, docgen_tutorial.py and tmpl_tutorial.html. The "docgen" scripts write the documentation to stdout. docgen_tutorial.py requires the Templayer HTML templating library to run: http://excess.org/templayer/
- Improved Widget and List Walker interface documentation.
- Fixed a bug in the handling of invalid UTF-8 data. All invalid characters are now replaced with '?' characters when displayed.

1.22 Urwid 0.9.3

2006-05-14

- Improved mouse reporting.

  The raw_display module now detects gpm mouse events by reading /usr/bin/mev output. The curses_display module already supports gpm directly.

  Mouse drag events are now reported by raw_display in terminals that provide button event tracking and on the console with gpm. Note that gpm may report coordinates off the screen if the user drags the mouse off the edge.

  Button release events now report which button was released if that information is available, currently only on the console with gpm.
- Added display of raw keycodes to the input_test.py example program.

- Fixed a text layout bug affecting clipped text with blank lines, and another related to wrapped text starting with a space character.
- Fixed a Frame.keypress() bug that caused it to call keypress on unselectable widgets.

1.23 Urwid 0.9.2

2006-03-18

- Preliminary mouse support was added to the raw_display and curses_display modules. A new Screen.set_mouse_tracking() method was added to enable mouse tracking. Mouse events are returned alongside keystrokes from the Screen.get_input() method.

  The widget interface does not yet include mouse handling. This will be addressed in the next release.
- A new convenience function is_mouse_event() was added to help in separating mouse events from keystrokes.
- Added a new example program input_test.py. This program displays the keyboard and mouse input it receives. It may be run as a CGI script or from the command line. On the command line it defaults to using the curses_display module, use input_test.py raw to use the raw_display module instead.
- Fixed an Edit.render() bug that caused it to render the cursor in a different location than that reported by Edit.get_cursor_coords() in some circumstances.
- Fixed a bug preventing use of UTF-8 characters with Divider widgets.

1.24 Urwid 0.9.1

2006-03-06

- BarGraph and ProgressBar can now display data more accurately by using the UTF-8 vertical and horizontal eighth characters. This behavior will be enabled when the UTF-8 encoding is detected and "smoothed" attributes are passed to the BarGraph or ProgressBar constructors.
- New get_encoding_mode() function to determine how Urwid will treat raw string data.
- New raw_display.signal_init() and raw_display.signal_restore() methods that may be overridden by threaded applications that need to call signal.signal() from their main thread.
- Fixed a bug that prevented the use of UTF-8 strings in text markup.
- Removed some forgotten asserts that broke 8-bit and CJK input.

1.25 Urwid 0.9.0

2006-02-18

- New support for UTF-8 encoding including input, display and editing of narrow and wide (CJK) characters.

  Preliminary combining (zero-width) character support is included, but full support will require terminal behavior detection.

  Right-to-Left input and display are not implemented.

- New raw_display module that handles console display without relying on external libraries. This module was written as a work around for the lack of UTF-8 support in the standard version of ncurses.

  Eliminates “dead corner” in the bottom right of the screen.

  Avoids use of bold text in xterm and gnome-terminal for improved text legibility.
- Fixed Overlay bug related to UTF-8 handling.
- Fixed Edit.move_cursor_to_coords(..) bug related to wide characters in UTF-8 encoding.

1.26 Urwid 0.9.0-pre3

2006-02-13

- Fixed Canvas attribute padding bug related to -pre1 changes.

1.27 Urwid 0.9.0-pre2

2006-02-10

- Replaced the custom align and wrap modes in example program calc.py with a new layout class.
- Fixed Overlay class call to Canvas.overlay() broken by -pre1 changes.
- Fixed Padding bug related to Canvas -pre1 changes.

1.28 Urwid 0.9.0-pre1

2006-02-08

- New support for UTF-8 encoding. Unicode strings may be used and will be converted to the current encoding when output. Regular strings in the current encoding may still be used.

  PLEASE NOTE: There are issues related to displaying UTF-8 characters with the curses_display module that have not yet been resolved.
- New set_encoding() function replaces util.set_double_byte_encoding().
- New supports_unicode() function to query if unicode strings with characters outside the ascii range may be used with the current encoding.
- New TextLayout and StandardTextLayout classes to perform text wrapping and alignment. Text widgets now have a layout parameter to allow use of custom TextLayout objects.
- New layout structure replaces line translation structure. Layout structure now allows arbitrary reordering/positioning of text segments, inclusion of UTF-8 characters and insertion of text not found in the original text string.
- Removed util.register_align_mode() and util.register_wrap_mode(). Their functionality has been replaced by the new layout classes.

1.29 Urwid 0.8.10

2005-11-27

- Expanded tutorial to cover advanced ListBox usage, custom widget classes and the Pile, BoxAdapter, Columns, GridFlow and Overlay classes.
- Added escape sequence for "shift tab" to curses_display.
- Added ListBox.set_focus_valign() to allow positioning of the focus widget within the ListBox.
- Added WidgetWrap class for extending existing widgets without inheriting their complete namespace.
- Fixed web_display/mozilla breakage from 0.8.9. Fixed crash on invalid locale setting. Fixed ListBox slide-back bug. Fixed improper space trimming in calculate_alignment(). Fixed browse.py example program rows bug. Fixed sum definition, use of long ints for python2.1. Fixed warnings with python2.1. Fixed Padding.get_pref_col() bug. Fixed Overlay splitting CJK characters bug.

1.30 Urwid 0.8.9

2005-10-21

- New Overlay class for drawing widgets that obscure parts of other widgets. May be used for drop down menus, combo boxes, overlapping "windows", caption text etc.
- New BarGraph, GraphVScale and ProgressBar classes for graphical display of data in Urwid applications.
- New method for configuring keyboard input timeouts and delays: curses_display.Screen.set_input_timeouts().
- Fixed a ListBox.set_focus() bug.

1.31 Urwid 0.8.8

2005-06-13

- New web_display module that emulates a console display within a web browser window. Application must be run as a CGI script under Apache.

  Supports font/window resizing, keepalive for long-lived connections, limiting maximum concurrent connections, polling and connected update methods. Tested with Mozilla Firefox and Internet Explorer.
- New BoxAdapter class for using box widgets in places that usually expect flow widgets.
- New curses_display input handling with better ESC key detection and broader escape code support.
- Shortened resize timeout on gradual resize to improve responsiveness.

1.32 Urwid 0.8.7

2005-05-21

- New widget classes: Button, RadioButton, CheckBox.
- New layout widget classes: Padding, GridFlow.
- New dialog.py example program that behaves like dialog(1) command.

- Pile widgets now support selectable items, focus changing with up and down keys and setting the cursor position.
- Frame widgets now support selectable items in the header and footer.
- Columns widgets now support fixed width and relative width columns, a minimum width for all columns, selectable items within columns containing flow widgets (already supported for box widgets), focus changing with left and right keys and setting the cursor position.
- Filler widgets may now wrap box widgets and have more alignment options.
- Updated tour.py example program to show new widget types and features.
- Avoid hogging cpu on gradual window resize and fix for slow resize with cygwin's broken curses implementation.
- Fixed minor CJK problem and curs_set() crash under MacOSX and Cygwin.
- Fixed crash when deleting cells in calc.py example program.

1.33 Urwid 0.8.6

2005-01-03

- Improved support for CJK double-byte encodings: BIG5, UHC, GBK, GB2312, CN-GB, EUC-KR, EUC-CN, EUC-JP (JISX 0208 only) and EUC-TW (CNS 11643 plain 1 only)
- Added support for ncurses' use_default_colors() function to curses_display module (Python >= 2.4).

  register_palette() and register_palette_entry() now accept "default" as foreground and/or background. If the terminal's default attributes cannot be detected black on light gray will be used to accomodate terminals with always-black cursors.

  "default" is now the default for text with no attributes. This means that areas with no attributes will change from light grey on black (curses default) to black on light gray or the terminal's default.
- Modified examples to not use black as background of Edit widgets.
- Fixed curses_display curs_set() call so that cursor is hidden when widget in focus has no cursor position.

1.34 Urwid 0.8.5

2004-12-15

- New tutorial covering basic operation of: curses_display.Screen, Canvas, Text, FlowWidget, Filler, BoxWidget, AttrWrap, Edit, ListBox and Frame classes
- New widget class: Filler
- New ListBox functions: get_focus(), set_focus()
- Debian packages for Python 2.4.
- Fixed curses_display bug affecting text with no attributes.

1.35 Urwid 0.8.4

2004-11-20

- Improved support for Cyrillic and other simple 8-bit encodings.
- Added new functions to simplify taking screenshots: html_fragment.screenshot_init() and html_fragment.screenshot_collect()
- Improved urwid/curses_display.py input debugging
- Fixed cursor in screenshots of CJK text. Fixed "end" key in Edit boxes with CJK text.

1.36 Urwid 0.8.3

2004-11-15

- Added support for CJK double-byte encodings.

  Word wrapping mode "space" will wrap on edges of double width characters. Wrapping and clipping will not split double width characters.

  curses_display.Screen.get_input() may now return double width characters. Text and Edit classes will work with a mix of regular and double width characters.
- Use new method Edit.set_edit_text() instead of Edit.update_text().
- Minor improvements to edit.py example program.

1.37 Urwid 0.8.2

2004-11-08

- Re-released under GNU Lesser General Public License.

1.38 Urwid 0.8.1

2004-10-29

- Added support for monochrome terminals. see curses_display.Screen.register_palette_entry() and example programs. set TERM=xterm-mono to test programs in monochrome mode.
- Added unit testing code test_urwid.py to the examples.
- Can now run urwid/curses_display.py to test your terminal's input and colour rendering.
- Fixed an OSX browse.py compatibility issue. Added some OSX keycodes.

1.39 Urwid 0.8.0

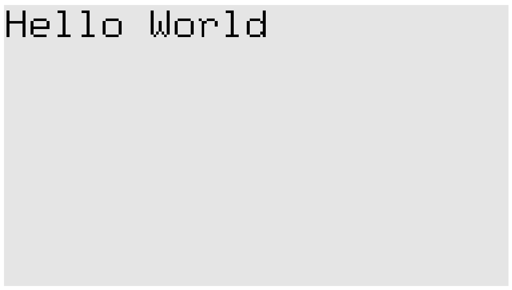
2004-10-17

- Initial Release

CHAPTER 2

Urwid Tutorial

2.1 Minimal Application



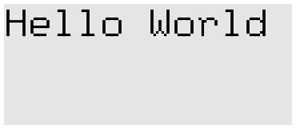
```
Hello World
```

This program displays the string `Hello World` in the top left corner of the screen and will run until interrupted with *CTRL+C* (*^C*).

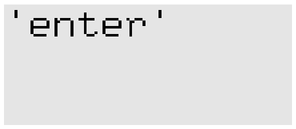
```python
import urwid

txt = urwid.Text(u"Hello World")
fill = urwid.Filler(txt, 'top')
loop = urwid.MainLoop(fill)
loop.run()
```

- The *txt* `Text` widget handles formatting blocks of text, wrapping to the next line when necessary. Widgets like this are called “flow widgets” because their sizing can have a number of columns given, in this case the full screen width, then they will flow to fill as many rows as necessary.
- The *fill* `Filler` widget fills in blank lines above or below flow widgets so that they can be displayed in a fixed number of rows. This Filler will align our Text to the top of the screen, filling all the rows below with blank lines. Widgets which are given both the number of columns and number of rows they must be displayed in are called “box widgets”.
- The `MainLoop` class handles displaying our widgets as well as accepting input from the user. The widget passed to `MainLoop` is called the “topmost” widget. The topmost widget is used to render the whole screen and so it must be a box widget. In this case our widgets can’t handle any user input so we need to interrupt the program to exit with *^C*.

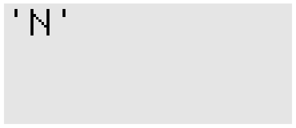
2.2 Global Input



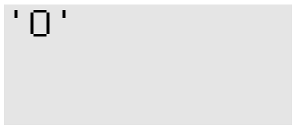
```
Hello World
```



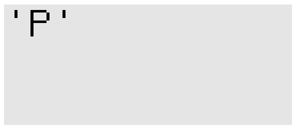
```
'enter'
```



```
'N'
```



```
'O'
```



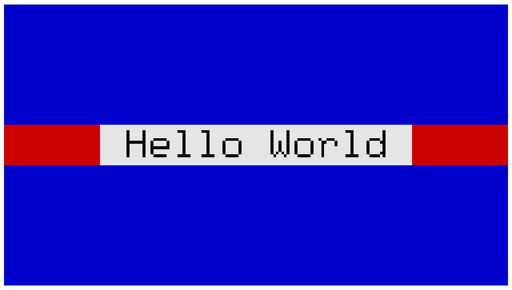
```
'P'
```

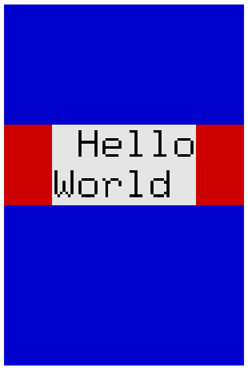
This program initially displays the string `Hello World`, then it displays each key pressed, exiting when the user presses *Q*.

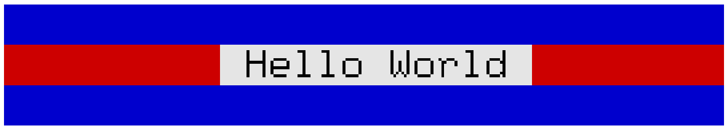
```python
import urwid

def show_or_exit(key):
    if key in ('q', 'Q'):
        raise urwid.ExitMainLoop()
    txt.set_text(repr(key))

txt = urwid.Text(u"Hello World")
fill = urwid.Filler(txt, 'top')
loop = urwid.MainLoop(fill, unhandled_input=show_or_exit)
loop.run()
```

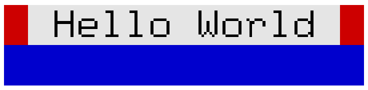
- The `MainLoop` class has an optional function parameter *unhandled_input*. This function will be called once for each keypress that is not handled by the widgets being displayed. Since none of the widgets being displayed here handle input, every key the user presses will be passed to the *show_or_exit* function.
- The `ExitMainLoop` exception is used to exit cleanly from the `MainLoop.run()` function when the user presses *Q*. All other input is displayed by replacing the current Text widget's content.

2.3 Display Attributes


Hello World


Hello
World


Hello World


Hello World

This program displays the string `Hello World` in the center of the screen. It uses different attributes for the text, the space on either side of the text and the space above and below the text. It waits for a keypress before exiting.

The screenshots above show how these widgets react to being resized.

```python
import urwid

def exit_on_q(key):
    if key in ('q', 'Q'):
        raise urwid.ExitMainLoop()

palette = [
    ('banner', 'black', 'light gray'),
    ('streak', 'black', 'dark red'),
    ('bg', 'black', 'dark blue'),]

txt = urwid.Text(('banner', u" Hello World "), align='center')
map1 = urwid.AttrMap(txt, 'streak')
fill = urwid.Filler(map1)
map2 = urwid.AttrMap(fill, 'bg')
loop = urwid.MainLoop(map2, palette, unhandled_input=exit_on_q)
loop.run()
```

- Display attributes are defined as part of a palette. Valid foreground, background and setting values are documented in *Foreground and Background Settings* A palette is a list of tuples containing:

1. Name of the display attribute, typically a string
2. Foreground color and settings for 16-color (normal) mode
3. Background color for normal mode
4. Settings for monochrome mode (optional)
5. Foreground color and settings for 88 and 256-color modes (optional, see next example)
6. Background color for 88 and 256-color modes (optional)

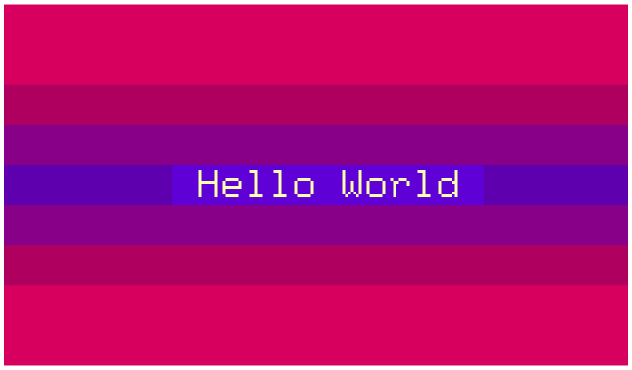
- A `Text` widget is created containing the string `" Hello World "` with display attribute `'banner'`. The attributes of text in a Text widget is set by using a (*attribute*, *text*) tuple instead of a simple text string. Display attributes will flow with the text, and multiple display attributes may be specified by combining tuples into a list. This format is called *Text Markup*.
- An `AttrMap` widget is created to wrap the text widget with display attribute `'streak'`. `AttrMap` widgets allow you to map any display attribute to any other display attribute, but by default they will set the display attribute of everything that does not already have a display attribute. In this case the text has an attribute, so only the areas around the text used for alignment will be have the new attribute.
- A second `AttrMap` widget is created to wrap the `Filler` widget with attribute `'bg'`.

When this program is run you can now clearly see the separation of the text, the alignment around the text, and the filler above and below the text.

See also:

Using Display Attributes

2.4 High Color Modes



This program displays the string `Hello World` in the center of the screen. It uses a number of 256-color-mode colors to decorate the text, and will work in any terminal that supports 256-color mode. It will exit when *Q* is pressed.

```python
import urwid

def exit_on_q(key):
    if key in ('q', 'Q'):
        raise urwid.ExitMainLoop()

palette = [
    ('banner', '', '', '', '#ffa', '#60d'),
    ('streak', '', '', '', 'g50', '#60a'),
    ('inside', '', '', '', 'g38', '#808'),
    ('outside', '', '', '', 'g27', '#a06'),
    ('bg', '', '', '', 'g7', '#d06'),]

placeholder = urwid.SolidFill()
loop = urwid.MainLoop(placeholder, palette, unhandled_input=exit_on_q)
loop.screen.set_terminal_properties(colors=256)
loop.widget = urwid.AttrMap(placeholder, 'bg')
loop.widget.original_widget = urwid.Filler(urwid.Pile([]))

div = urwid.Divider()
outside = urwid.AttrMap(div, 'outside')
inside = urwid.AttrMap(div, 'inside')
txt = urwid.Text(('banner', u" Hello World "), align='center')
streak = urwid.AttrMap(txt, 'streak')
pile = loop.widget.base_widget # .base_widget skips the decorations
for item in [outside, inside, streak, inside, outside]:
    pile.contents.append((item, pile.options()))

loop.run()
```

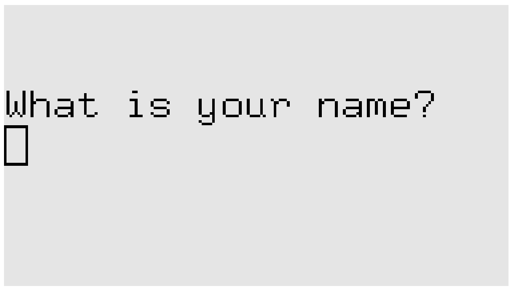
This palette only defines values for the high color foregroundand backgrounds, because only the high colors will be used. A real application should define values for all the modes in their palette. Valid foreground, background and setting values are documented in *Foreground and Background Settings*.

- Behind the scenes our `MainLoop` class has created a `raw_display.Screen` object for drawing the screen. The program is put into 256-color mode by using the screen object's `set_terminal_properties()` method.

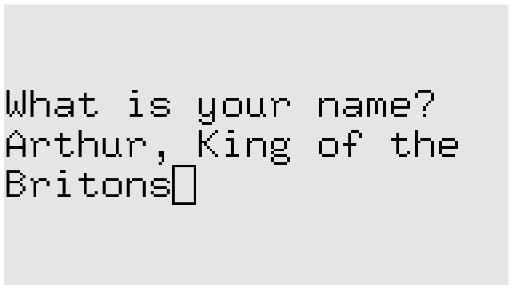
This example also demonstrates how you can build the widgets to display in a top-down order instead of the usual bottom-up order. In some places we need to use a *placeholder* widget because we must provide a widget before the correct one has been created.

- We change the topmost widget used by the `MainLoop` by assigning to its `MainLoop.widget` property.
- *Decoration Widgets* like `AttrMap` have an `original_widget` property that we can assign to to change the widget they wrap.
- `Divider` widgets are used to create blank lines, colored with `AttrMap`.
- *Container Widgets* like `Pile` have a `contents` property that we can treat like a list of (*widget*, *options*) tuples. `Pile.contents` supports normal list operations including `append()` to add child widgets. `Pile.options()` is used to generate the default options for the new child widgets.

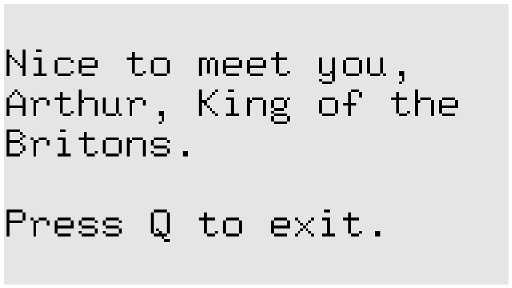
2.5 Question and Answer



```
What is your name?
```



```
What is your name?
Arthur, King of the
Britons
```



```
Nice to meet you,
Arthur, King of the
Britons.

Press Q to exit.
```

This program asks for your name then responds `Nice to meet you, (your name).`

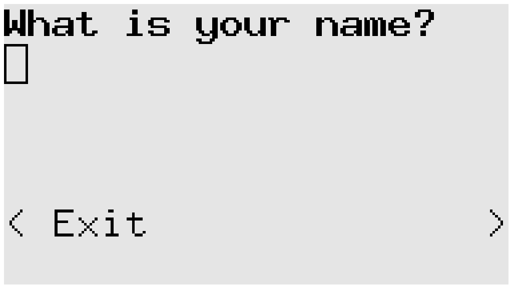
```python
import urwid

def exit_on_q(key):
    if key in ('q', 'Q'):
        raise urwid.ExitMainLoop()

class QuestionBox(urwid.Filler):
    def keypress(self, size, key):
        if key != 'enter':
            return super(QuestionBox, self).keypress(size, key)
        self.original_widget = urwid.Text(
            u"Nice to meet you,\n%s.\n\nPress Q to exit." %
            edit.edit_text)

edit = urwid.Edit(u"What is your name?\n")
fill = QuestionBox(edit)
loop = urwid.MainLoop(fill, unhandled_input=exit_on_q)
loop.run()
```

The `Edit` widget is based on the `Text` widget but it accepts keyboard input for entering text, making corrections and moving the cursor around with the *HOME*, *END* and arrow keys.

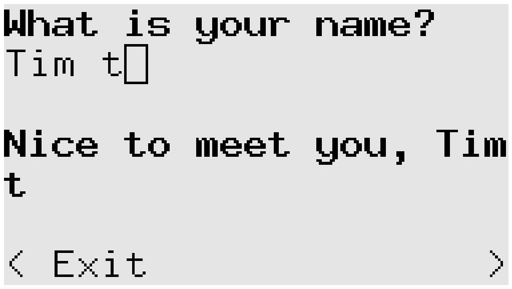
Here we are customizing the `Filler` decoration widget that is holding our `Edit` widget by subclassing it and defining a new `keypress()` method. Customizing decoration or container widgets to handle input this way is a common pattern in Urwid applications. This pattern is easier to maintain and extend than handling all special input in an *unhandled_input* function.

- In *QuestionBox.keypress()* all keypresses except *ENTER* are passed along to the default `Filler.keypress()` which sends them to the child `Edit.keypress()` method.
- Note that names containing *Q* can be entered into the `Edit` widget without causing the program to exit because `Edit.keypress()` indicates that it has handled the key by returning `None`. See `Widget.keypress()` for more information.
- When *ENTER* is pressed the child widget `original_widget` is changed to a `Text` widget.
- `Text` widgets don't handle any keyboard input so all input ends up in the *unhandled_input* function *exit_on_q*, allowing the user to exit the program.

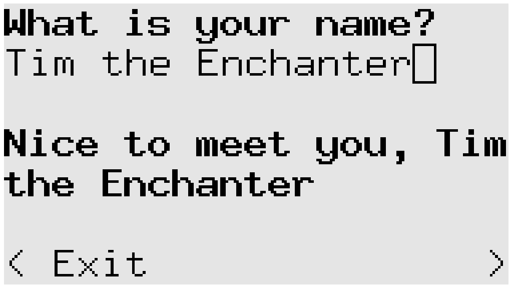
2.6 Signal Handlers



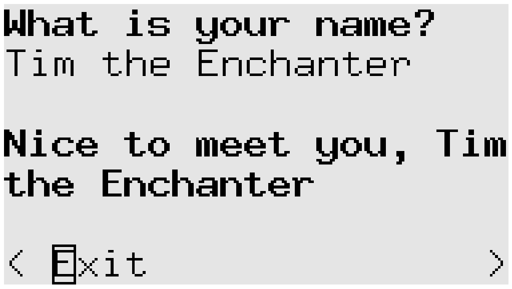
```
What is your name?
 



< Exit             >
```



```
What is your name?
Tim t

Nice to meet you, Tim
t

< Exit             >
```



```
What is your name?
Tim the Enchanter

Nice to meet you, Tim
the Enchanter

< Exit             >
```



```
What is your name?
Tim the Enchanter

Nice to meet you, Tim
the Enchanter

< Exit             >
```

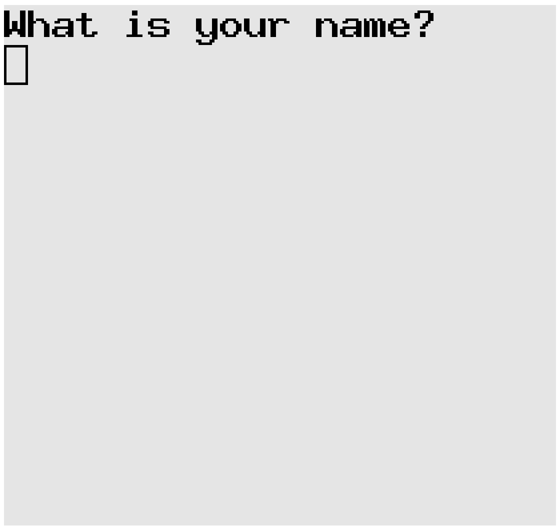
This program asks for your name and responds `Nice to meet you, (your name)` *while* you type your name. Press *DOWN* then *SPACE* or *ENTER* to exit.

```python
import urwid

palette = [('I say', 'default,bold', 'default', 'bold'),]
ask = urwid.Edit(('I say', u"What is your name?\n"))
reply = urwid.Text(u"")
button = urwid.Button(u'Exit')
div = urwid.Divider()
pile = urwid.Pile([ask, div, reply, div, button])
top = urwid.Filler(pile, valign='top')

def on_ask_change(edit, new_edit_text):
    reply.set_text(('I say', u"Nice to meet you, %s" % new_edit_text))
```

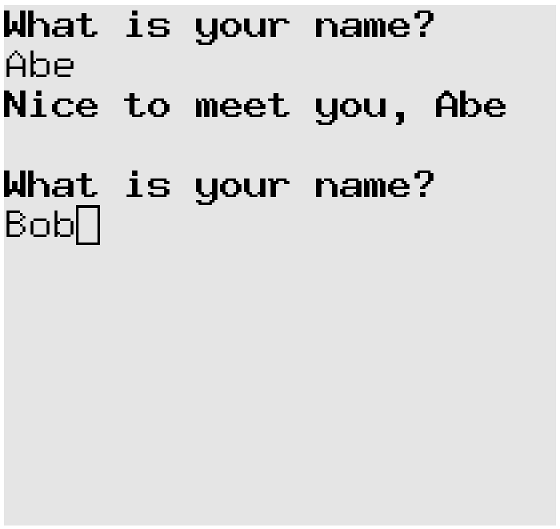
```

def on_exit_clicked(button):
    raise urwid.ExitMainLoop()

urwid.connect_signal(ask, 'change', on_ask_change)
urwid.connect_signal(button, 'click', on_exit_clicked)

urwid.MainLoop(top, palette).run()
```

- An `Edit` widget and a `Text` reply widget are created, like in the previous example.
- The `connect_signal()` function is used to attach our *on_ask_change()* function to our `Edit` widget's `'change'` signal. Now any time the content of the `Edit` widget changes *on_ask_change()* will be called and passed the new content.
- Finally we attach our *on_exit_clicked()* function to our exit `Button`'s `'click'` signal.
- *on_ask_change()* updates the reply text as the user enters their name and *on_exit_click()* exits.

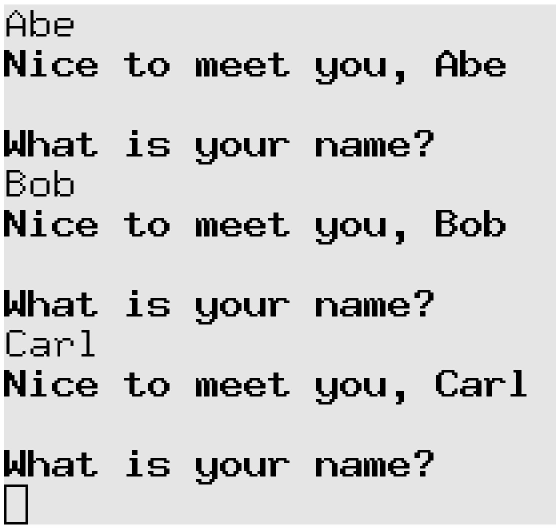
2.7 Multiple Questions



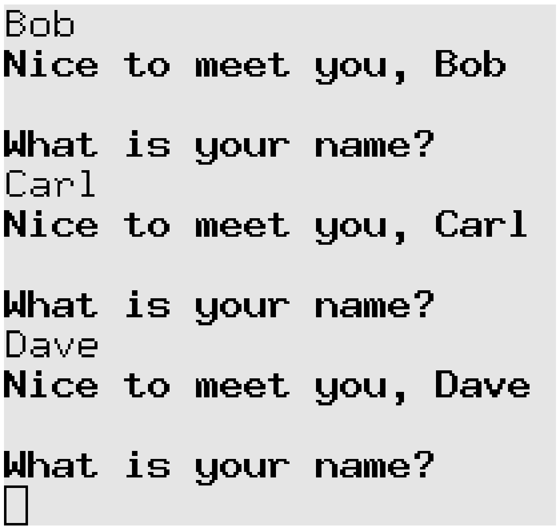
```
What is your name?
```



```
What is your name?
Abe
Nice to meet you, Abe

What is your name?
Bob
```



```
Abe
Nice to meet you, Abe

What is your name?
Bob
Nice to meet you, Bob

What is your name?
Carl
Nice to meet you, Carl

What is your name?
```



```
Bob
Nice to meet you, Bob

What is your name?
Carl
Nice to meet you, Carl

What is your name?
Dave
Nice to meet you, Dave

What is your name?
```

This program asks for your name and responds `Nice to meet you, (your name)`. It then asks again, and again. Old values may be changed and the responses will be updated when you press *ENTER*. *ENTER* on a blank line exits.

```python
import urwid

def question():
    return urwid.Pile([urwid.Edit(('I say', u"What is your name?\n"))])

def answer(name):
    return urwid.Text(('I say', u"Nice to meet you, " + name + "\n"))

class ConversationListBox(urwid.ListBox):
    def __init__(self):
        body = urwid.SimpleFocusListWalker([question()])
        super(ConversationListBox, self).__init__(body)

    def keypress(self, size, key):
        key = super(ConversationListBox, self).keypress(size, key)
        if key != 'enter':
            return key
        name = self.focus[0].edit_text
        if not name:
            raise urwid.ExitMainLoop()
        # replace or add response
        self.focus.contents[1:] = [(answer(name), self.focus.options())]
        pos = self.focus_position
        # add a new question
        self.body.insert(pos + 1, question())
        self.focus_position = pos + 1

palette = [('I say', 'default,bold', 'default'),]
urwid.MainLoop(ConversationListBox(), palette).run()
```

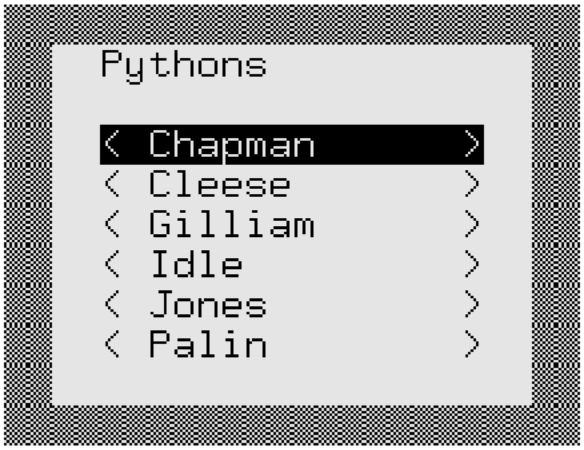
`ListBox` widgets let you scroll through a number of flow widgets vertically. It handles *UP*, *DOWN*, *PAGE UP* and *PAGE DOWN* keystrokes and changing the focus for you. *ListBox Contents* are managed by a "list walker", one of the list walkers that is easiest to use is `SimpleFocusListWalker`.

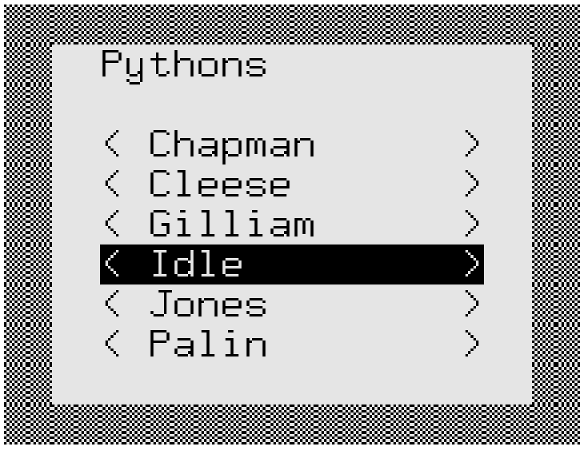
`SimpleFocusListWalker` is like a normal python list of widgets, but any time you insert or remove widgets the focus position is updated automatically.

Here we are customizing our `ListBox`'s keypress handling by overriding it in a subclass.

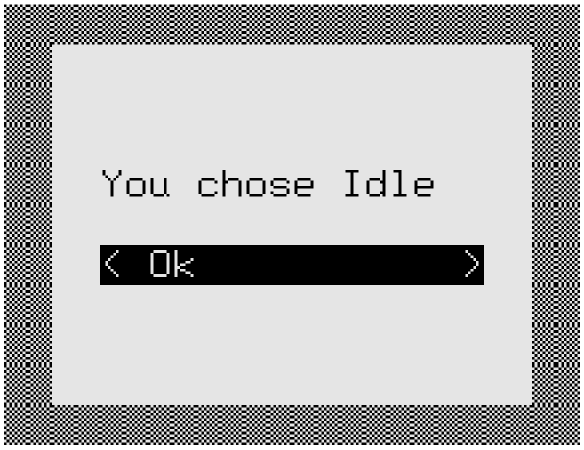
- The *question()* function is used to build widgets to communicate with the user. Here we return a `Pile` widget with a single `Edit` widget to start.
- We retrieve the name entered with `ListBox.focus` to get the `Pile` in focus, the standard *container widget* method `[0]` to get the first child of the pile and `Edit.edit_text` to get the user-entered text.
- For the response we use the fact that we can treat `Pile.contents` like a list of (*widget*, *options*) tuples to create or replace any existing response by assigning a one-tuple list to *contents[1:]*. We create the default options using `Pile.options()`.
- To add another question after the current one we treat our `SimpleFocusListWalker` stored as `ListBox.body` like a normal list of widgets by calling *insert()*, then update the focus position to the widget we just created.

2.8 Simple Menu


```
Pythons

< Chapman     >
< Cleese      >
< Gilliam     >
< Idle        >
< Jones       >
< Palin       >
```



```
Pythons

< Chapman        >
< Cleese         >
< Gilliam        >
< Idle           >
< Jones          >
< Palin          >
```



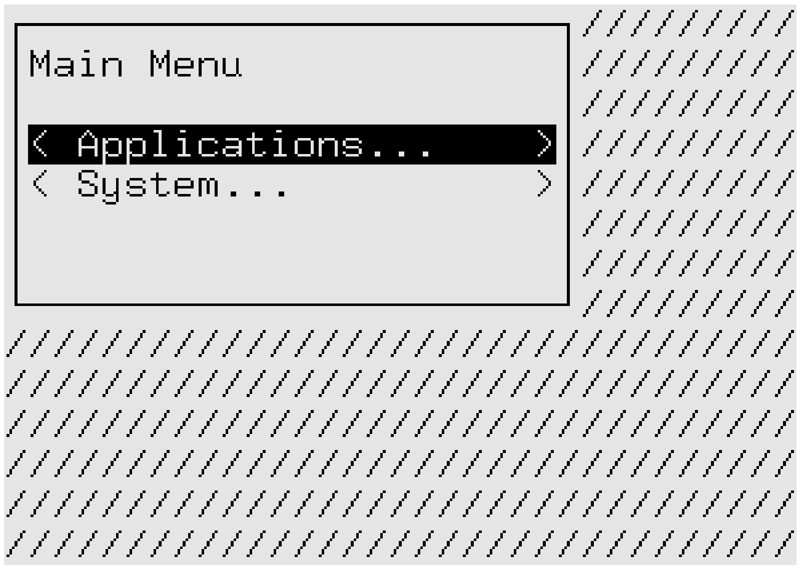
```
You chose Idle

< Ok             >
```

We can create a very simple menu using a list of `Button` widgets. This program lets you choose an option then repeats what you chose.

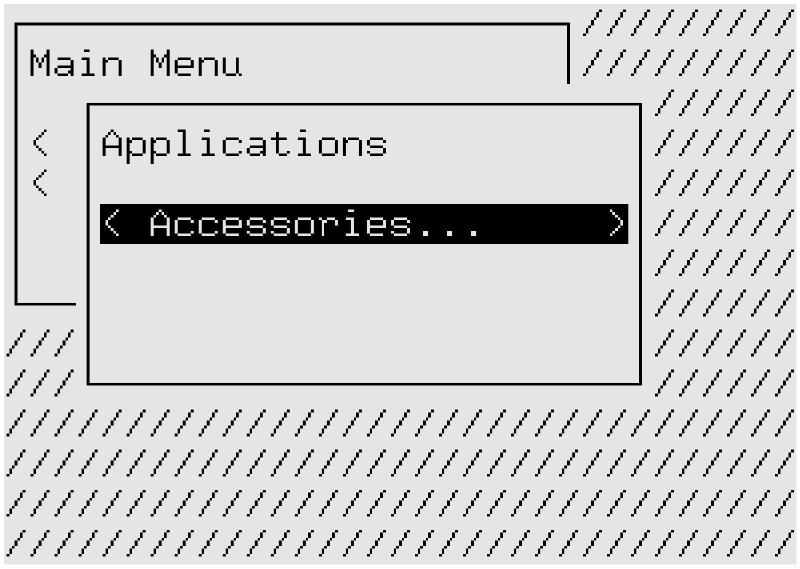
```python
import urwid

choices = u'Chapman Cleese Gilliam Idle Jones Palin'.split()

def menu(title, choices):
    body = [urwid.Text(title), urwid.Divider()]
    for c in choices:
        button = urwid.Button(c)
        urwid.connect_signal(button, 'click', item_chosen, c)
        body.append(urwid.AttrMap(button, None, focus_map='reversed'))
    return urwid.ListBox(urwid.SimpleFocusListWalker(body))

def item_chosen(button, choice):
    response = urwid.Text([u'You chose ', choice, u'\n'])
    done = urwid.Button(u'Ok')
    urwid.connect_signal(done, 'click', exit_program)
    main.original_widget = urwid.Filler(urwid.Pile([response,
        urwid.AttrMap(done, None, focus_map='reversed')]))

def exit_program(button):
    raise urwid.ExitMainLoop()

main = urwid.Padding(menu(u'Pythons', choices), left=2, right=2)
top = urwid.Overlay(main, urwid.SolidFill(u'\N{MEDIUM SHADE}'),
    align='center', width=('relative', 60),
    valign='middle', height=('relative', 60),
    min_width=20, min_height=9)
urwid.MainLoop(top, palette=[('reversed', 'standout', '')]).run()
```

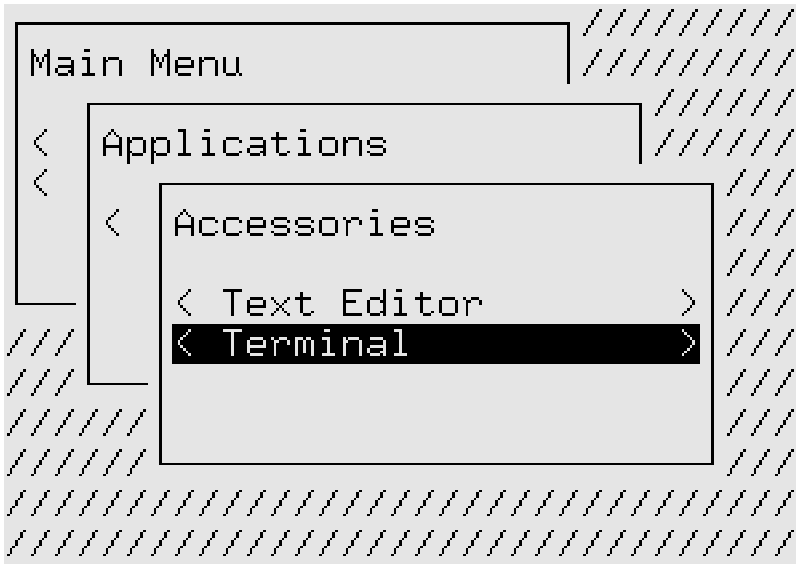
- *menu()* builds a `ListBox` with a *title* and a sequence of `Button` widgets. Each button has its `'click'` signal attached to *item_chosen*, with item name is passed as data. The buttons are decorated with an `AttrMap` that applies a display attribute when a button is in focus.
- *item_chosen()* replaces the menu displayed with text indicating the users' choice.
- *exit_program()* causes the program to exit on any keystroke.
- The menu is created and decorated with an `Overlay` using a `SolidFill` as the background. The `Overlay` is given a miniumum width and height but is allowed to expand to 60% of the available space if the user's terminal window is large enough.

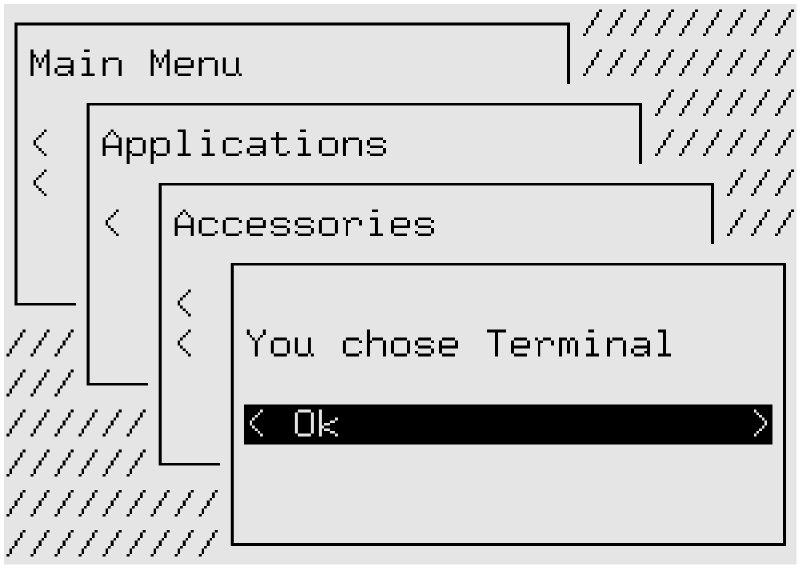
2.9 Cascading Menu



```
Main Menu

< Applications...    >
< System...          >
```






You chose Terminal

A nested menu effect can be created by having some buttons open new menus. This program lets you choose an option from a nested menu that cascades across the screen. You may return to previous menus by pressing *ESC*.

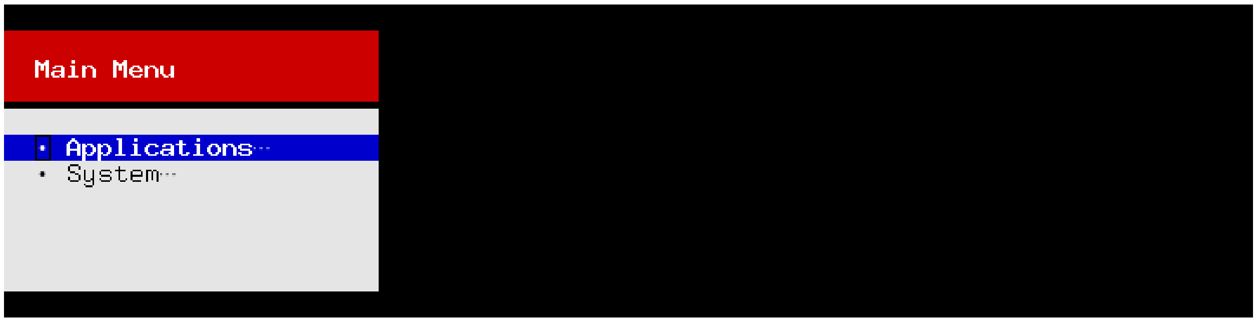
```
import urwid

def menu_button(caption, callback):
    button = urwid.Button(caption)
    urwid.connect_signal(button, 'click', callback)
    return urwid.AttrMap(button, None, focus_map='reversed')

def sub_menu(caption, choices):
    contents = menu(caption, choices)
    def open_menu(button):
        return top.open_box(contents)
    return menu_button([caption, u'...'], open_menu)

def menu(title, choices):
    body = [urwid.Text(title), urwid.Divider()]
    body.extend(choices)
    return urwid.ListBox(urwid.SimpleFocusListWalker(body))

def item_chosen(button):
    response = urwid.Text([u'You chose ', button.label, u'\n'])
    done = menu_button(u'Ok', exit_program)
    top.open_box(urwid.Filler(urwid.Pile([response, done])))

def exit_program(button):
    raise urwid.ExitMainLoop()

menu_top = menu(u'Main Menu', [
    sub_menu(u'Applications', [
        sub_menu(u'Accessories', [
            menu_button(u'Text Editor', item_chosen),
            menu_button(u'Terminal', item_chosen),
        ]),
    ]),
    sub_menu(u'System', [
        sub_menu(u'Preferences', [
            menu_button(u'Appearance', item_chosen),
        ]),
        menu_button(u'Lock Screen', item_chosen),
    ]),
])

class CascadingBoxes(urwid.WidgetPlaceholder):
    max_box_levels = 4

    def __init__(self, box):
        super(CascadingBoxes, self).__init__(urwid.SolidFill(u'/'))
        self.box_level = 0
        self.open_box(box)

    def open_box(self, box):
        self.original_widget = urwid.Overlay(urwid.LineBox(box),
            self.original_widget,
            align='center', width=('relative', 80),
            valign='middle', height=('relative', 80),
            min_width=24, min_height=8,
            left=self.box_level * 3,
            right=(self.max_box_levels - self.box_level - 1) * 3,
            top=self.box_level * 2,
            bottom=(self.max_box_levels - self.box_level - 1) * 2)
        self.box_level += 1

    def keypress(self, size, key):
        if key == 'esc' and self.box_level > 1:
            self.original_widget = self.original_widget[0]
            self.box_level -= 1
        else:
            return super(CascadingBoxes, self).keypress(size, key)

top = CascadingBoxes(menu_top)
urwid.MainLoop(top, palette=[('reversed', 'standout', '')]).run()
```

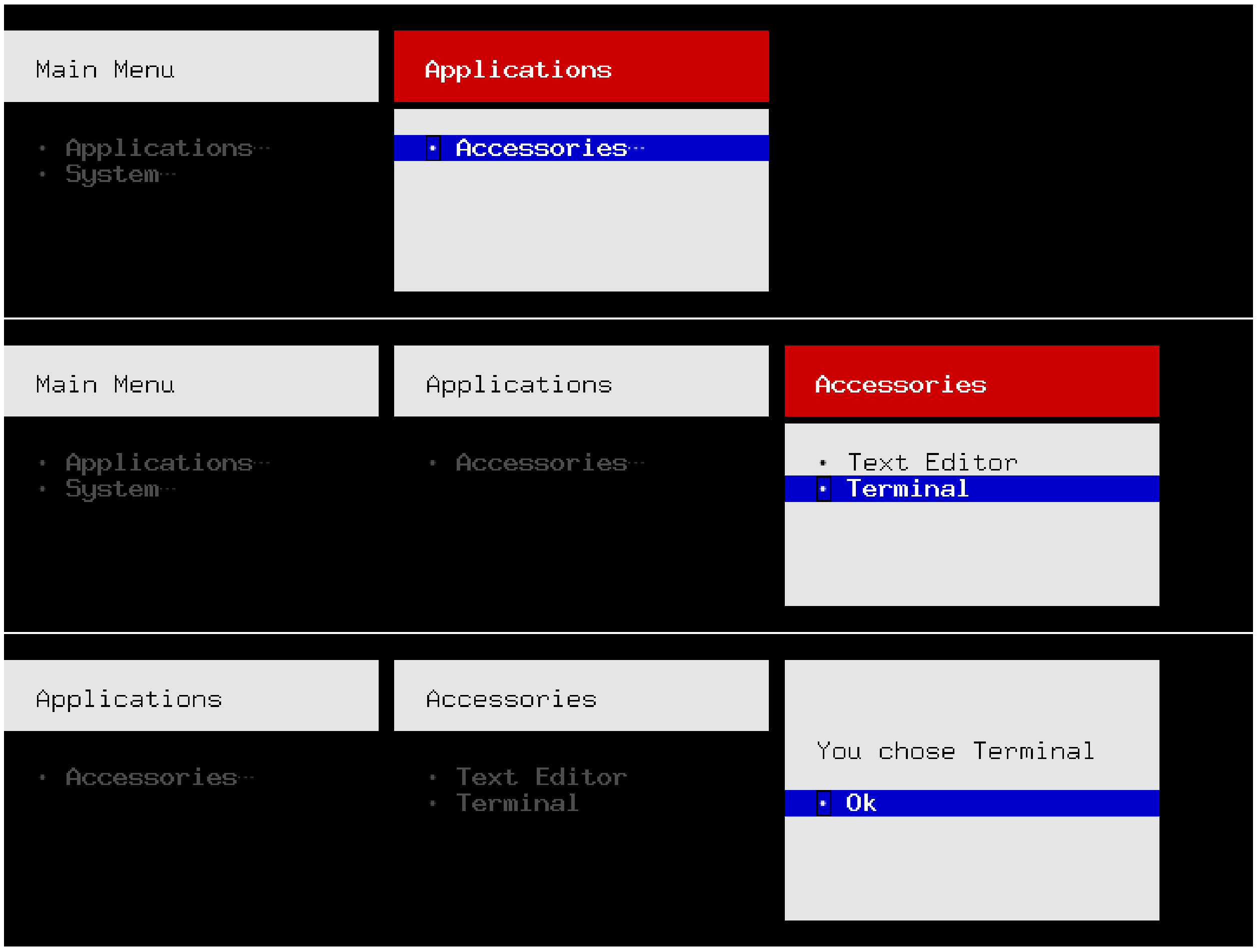
- *menu_button()* returns an `AttrMap`-decorated `Button` and attaches a *callback* to the the its `'click'` signal. This function is used for both sub-menus and final selection buttons.
- *sub_menu()* creates a menu button and a closure that will open the the menu when that button is clicked. Notice that *text markup* is used to add `'...'` to the end of the *caption* passed to *menu_button()*.
- *menu()* builds a `ListBox` with a *title* and a sequence of widgets.
- *item_chosen()* displays the users' choice similar to the previous example.
- *menu_top* is the top level menu with all of its child menus and options built using the functions above.

This example introduces `WidgetPlaceholder`. `WidgetPlaceholder` is a *decoration widget* that does nothing to the widget it decorates. It is useful if you need a simple way to replace a widget that doesn't involve knowing its position in a *container*, or in this case as a base class for a widget that will be replacing its own contents regularly.

- *CascadingBoxes* is a new widget that extends `WidgetPlaceholder`. It provides an *open_box()* method that displays a box widget *box* "on top of" all the previous content with an `Overlay` and a `LineBox`. The position of each successive box is shifted right and down from the previous one.
- *CascadingBoxes.keypress()* intercepts *ESC* keys to cause the current box to be removed and the previous one to be shown. This allows the user to return to a previous menu level.

2.10 Horizontal Menu





This example is like the previous but new menus appear on the right and push old menus off the left side of the screen. The look of buttons and other menu elements are heavily customized and new widget classes are used instead of factory functions.

```python
import urwid

class MenuButton(urwid.Button):
    def __init__(self, caption, callback):
        super(MenuButton, self).__init__("")
        urwid.connect_signal(self, 'click', callback)
        self._w = urwid.AttrMap(urwid.SelectableIcon(
            [u'  \N{BULLET} ', caption], 2), None, 'selected')

class SubMenu(urwid.WidgetWrap):
    def __init__(self, caption, choices):
        super(SubMenu, self).__init__(MenuButton(
            [caption, u"\N{HORIZONTAL ELLIPSIS}"], self.open_menu))
        line = urwid.Divider(u'\N{LOWER ONE QUARTER BLOCK}')
        listbox = urwid.ListBox(urwid.SimpleFocusListWalker([
            urwid.AttrMap(urwid.Text([u"\n  ", caption]), 'heading'),
            urwid.AttrMap(line, 'line'),
            urwid.Divider()] + choices + [urwid.Divider()]))
        self.menu = urwid.AttrMap(listbox, 'options')

    def open_menu(self, button):
        top.open_box(self.menu)
```

```

class Choice(urwid.WidgetWrap):
    def __init__(self, caption):
        super(Choice, self).__init__(
            MenuButton(caption, self.item_chosen))
        self.caption = caption

    def item_chosen(self, button):
        response = urwid.Text([u'  You chose ', self.caption, u'\n'])
        done = MenuButton(u'Ok', exit_program)
        response_box = urwid.Filler(urwid.Pile([response, done]))
        top.open_box(urwid.AttrMap(response_box, 'options'))

def exit_program(key):
    raise urwid.ExitMainLoop()

menu_top = SubMenu(u'Main Menu', [
    SubMenu(u'Applications', [
        SubMenu(u'Accessories', [
            Choice(u'Text Editor'),
            Choice(u'Terminal'),
        ]),
    ]),
    SubMenu(u'System', [
        SubMenu(u'Preferences', [
            Choice(u'Appearance'),
        ]),
        Choice(u'Lock Screen'),
    ]),
])

palette = [
    (None,  'light gray', 'black'),
    ('heading', 'black', 'light gray'),
    ('line', 'black', 'light gray'),
    ('options', 'dark gray', 'black'),
    ('focus heading', 'white', 'dark red'),
    ('focus line', 'black', 'dark red'),
    ('focus options', 'black', 'light gray'),
    ('selected', 'white', 'dark blue')]
focus_map = {
    'heading': 'focus heading',
    'options': 'focus options',
    'line': 'focus line'}

class HorizontalBoxes(urwid.Columns):
    def __init__(self):
        super(HorizontalBoxes, self).__init__([], dividechars=1)

    def open_box(self, box):
        if self.contents:
            del self.contents[self.focus_position + 1:]
        self.contents.append((urwid.AttrMap(box, 'options', focus_map),
            self.options('given', 24)))
        self.focus_position = len(self.contents) - 1

top = HorizontalBoxes()
top.open_box(menu_top.menu)
```

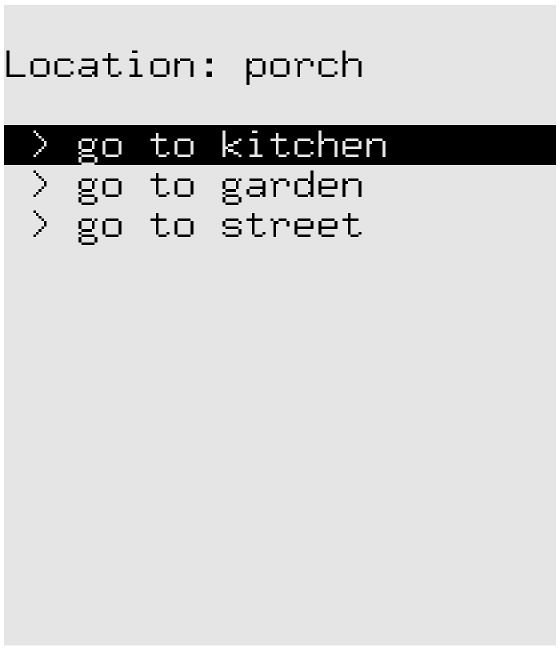
```
urwid.MainLoop(urwid.Filler(top, 'middle', 10), palette).run()
```

- *MenuButton* is a customized `Button` widget. `Button` uses `WidgetWrap` to create its appearance and this class replaces the display widget created by `Button` by the wrapped widget in *self._w*.
- *SubMenu* is implemented with a *MenuButton* but uses `WidgetWrap` to hide the implementation instead of inheriting from *MenuButton*. The constructor builds a widget for the menu that this button will open and stores it in *self.menu*.
- *Choice* is like *SubMenu* but displays the item chosen instead of another menu.

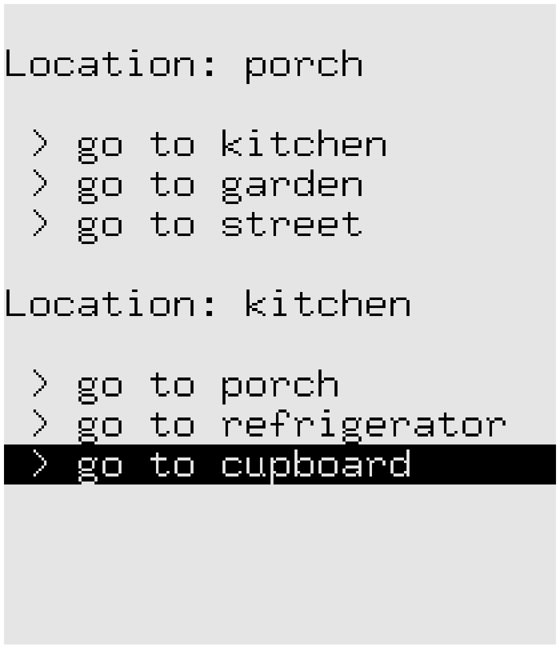
The *palette* used in this example includes an entry with the special name `None`. The foreground and background specified in this entry are used as a default when no other display attribute is specified.

- *HorizontalBoxes* arranges the menus displayed similar to the previous example. There is no special handling required for going to previous menus here because `Columns` already handles switching focus when *LEFT* or *RIGHT* is pressed. `AttrMap` with the *focus_map* dict is used to change the appearance of a number of the display attributes when a menu is in focus.

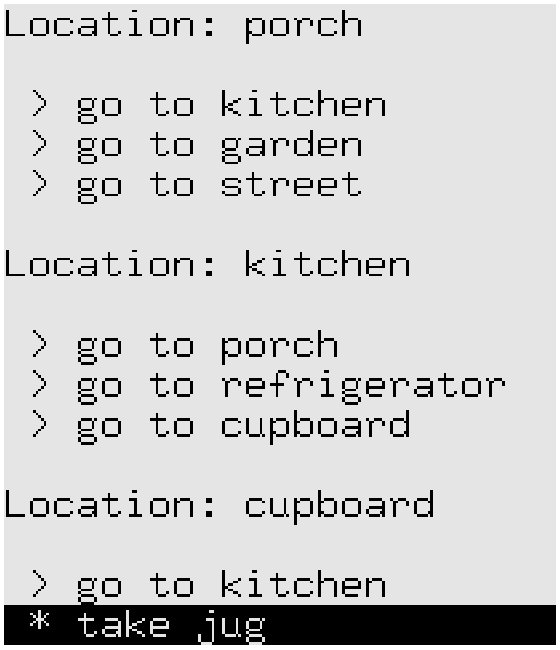
2.11 Adventure Game



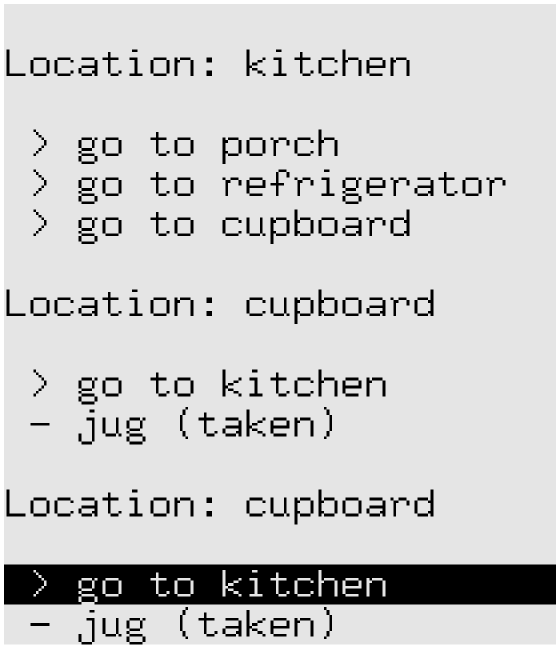
```
Location: porch

 > go to kitchen
 > go to garden
 > go to street
```



```

Location: porch

 > go to kitchen
 > go to garden
 > go to street

Location: kitchen

 > go to porch
 > go to refrigerator
 > go to cupboard
```



```
Location: porch

 > go to kitchen
 > go to garden
 > go to street

Location: kitchen

 > go to porch
 > go to refrigerator
 > go to cupboard

Location: cupboard

 > go to kitchen
 * take jug
```



```
Location: kitchen

 > go to porch
 > go to refrigerator
 > go to cupboard

Location: cupboard

 > go to kitchen
 - jug (taken)

Location: cupboard

 > go to kitchen
 - jug (taken)
```

We can use the same sort of code to build a simple adventure game. Instead of menus we have “places” and instead of submenus and parent menus we just have “exits”. This example scrolls previous places off the top of the screen, allowing you to scroll back to view but not interact with previous places.

```python
import urwid

class ActionButton(urwid.Button):
    def __init__(self, caption, callback):
        super(ActionButton, self).__init__("")
        urwid.connect_signal(self, 'click', callback)
        self._w = urwid.AttrMap(urwid.SelectableIcon(caption, 1),
            None, focus_map='reversed')

class Place(urwid.WidgetWrap):
    def __init__(self, name, choices):
        super(Place, self).__init__(
            ActionButton([u" > go to ", name], self.enter_place))
        self.heading = urwid.Text([u"\nLocation: ", name, "\n"])
        self.choices = choices
        # create links back to ourself
        for child in choices:
            getattr(child, 'choices', []).insert(0, self)

    def enter_place(self, button):
        game.update_place(self)

class Thing(urwid.WidgetWrap):
    def __init__(self, name):
        super(Thing, self).__init__(
            ActionButton([u" * take ", name], self.take_thing))
        self.name = name

    def take_thing(self, button):
        self._w = urwid.Text(u" - %s (taken)" % self.name)
        game.take_thing(self)

```

```
def exit_program(button):
    raise urwid.ExitMainLoop()

map_top = Place(u'porch', [
    Place(u'kitchen', [
        Place(u'refrigerator', []),
        Place(u'cupboard', [
            Thing(u'jug'),
        ]),
    ]),
    Place(u'garden', [
        Place(u'tree', [
            Thing(u'lemon'),
            Thing(u'bird'),
        ]),
    ]),
    Place(u'street', [
        Place(u'store', [
            Thing(u'sugar'),
        ]),
        Place(u'lake', [
            Place(u'beach', []),
        ]),
    ]),
])

class AdventureGame(object):
    def __init__(self):
        self.log = urwid.SimpleFocusListWalker([])
        self.top = urwid.ListBox(self.log)
        self.inventory = set()
        self.update_place(map_top)

    def update_place(self, place):
        if self.log: # disable interaction with previous place
            self.log[-1] = urwid.WidgetDisable(self.log[-1])
        self.log.append(urwid.Pile([place.heading] + place.choices))
        self.top.focus_position = len(self.log) - 1
        self.place = place

    def take_thing(self, thing):
        self.inventory.add(thing.name)
        if self.inventory >= set([u'sugar', u'lemon', u'jug']):
            response = urwid.Text(u'You can make lemonade!\n')
            done = ActionButton(u' - Joy', exit_program)
            self.log[:] = [response, done]
        else:
            self.update_place(self.place)

game = AdventureGame()
urwid.MainLoop(game.top, palette=[('reversed', 'standout', '')]).run()
```

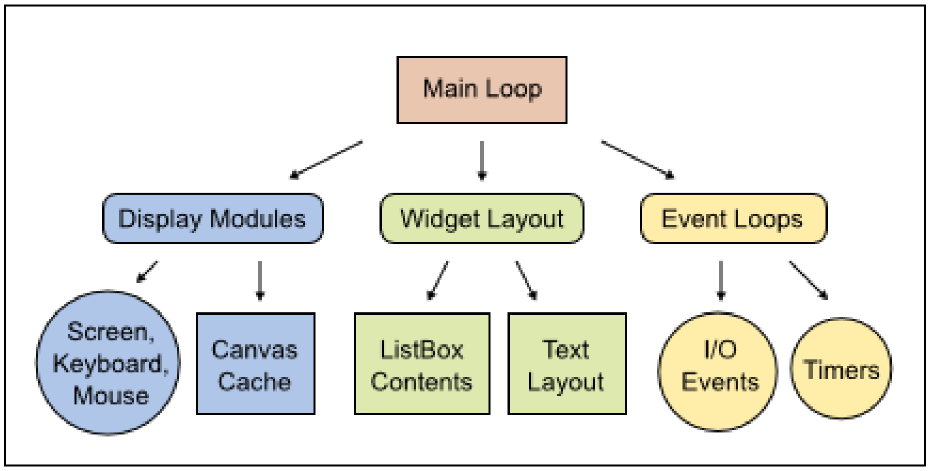
This example starts to show some separation between the application logic and the widgets that have been created. The *AdventureGame* class is responsible for all the changes that happen through the game and manages the topmost widget, but isn't a widget itself. This is a good pattern to follow as your application grows larger.

CHAPTER 3

Urwid Manual

3.1 Library Overview

Urwid is a console user interface library for Python. Urwid offers an alternative to using Python's curses module directly and handles many of the difficult and tedious tasks for you.



Each Urwid component is loosely coupled and designed to be extended by the user.

Display modules are responsible for accepting *user input* and converting escape sequences to lists of keystrokes and mouse events. They also draw the screen contents and convert attributes used in the canvases rendered to the actual colors that appear on screen.

The included widgets are simple building blocks and examples that try not to impose a particular style of interface. It may be helpful to think of Urwid as a console widget construction set rather than a finished UI library like GTK or Qt. The `Widget base class` describes the widget interface and *widget layout* describes how widgets are nested and arranged on the screen.

Text is the bulk of what will be displayed in any console user interface. Urwid supports a number of *text encodings* and Urwid comes with a configurable *text layout* that handles the most of the common alignment and wrapping modes. If you need more flexibility you can also write your own text layout classes.

Urwid supports a range of common *display attributes*, including 256-color foreground and background settings, bold, underline and standout settings for displaying text. Not all of these are supported by all terminals, so Urwid helps you write applications that support different color modes depending on what the user's terminal supports and what they choose to enable.

`ListBox` is one of Urwid's most powerful widgets, and you may control of the *listbox contents* by using a built-in list walker class or by writing one yourself. This is very useful for scrolling through lists of any significant length, or with nesting, folding and other similar features.

When a widget renders a canvas to be drawn on screen, a weak reference to it is stored in the *canvas cache*. This cache is used any time a widget needs to be rendered again, reducing the amount of work required to update the screen. Since only weak references are used, Urwid's display modules will hold on to a reference to the canvas that they are currently displaying as a way to keep the cache alive and populated with current data.

Urwid's *main loop* simplifies handling of input and updating the screen. It also lets you use one of a number of *the event loops*, allowing integration with Twisted's reactor or Glib's event loop if desired.

3.2 Main Loop

The `MainLoop` class ties together a *display module*, a set of widgets and an *event loop*. It handles passing input from the display module to the widgets, rendering the widgets and passing the rendered canvas to the display module to be drawn.

You may filter the user's input before it is passed to the widgets with your own code by using `MainLoop.input_filter()`, or have special code to handle input not handled by the widgets by using `MainLoop.unhandled_input()`.

You may set alarms to create timed events using `MainLoop.set_alarm_at()` or `MainLoop.set_alarm_in()`. These methods automatically add a call to `MainLoop.draw_screen()` after calling your callback. `MainLoop.remove_alarm()` may be used to remove alarms.

When the main loop is running, any code that raises an `ExitMainLoop` exception will cause the loop to exit cleanly. If any other exception reaches the main loop code, it will shut down the screen to avoid leaving the terminal in an unusual state then re-raise the exception for normal handling.

Using `MainLoop` is highly recommended, but if it does not fit the needs of your application you may choose to use your own code instead. There are no dependencies on `MainLoop` in other parts of Urwid.

3.2.1 Widgets Displayed

The topmost widget displayed by `MainLoop` must be passed as the first parameter to the constructor. If you want to change the topmost widget while running, you can assign a new widget to the `MainLoop` object's `MainLoop.widget` attribute. This is useful for applications that have a number of different modes or views.

The displayed widgets will be handling user input, so it is better to extend the widgets that are displayed with your application-specific input handling so that the application's behaviour changes when the widgets change. If all your custom input handling is done from `MainLoop.unhandled_input()`, it will be difficult to extend as your application gets more complicated.

3.2.2 Event Loops

Urwid's event loop classes handle waiting for things for the `MainLoop`. The different event loops allow you to integrate with Twisted or Glib libraries, or use a simple `select`-based loop. Event loop classes abstract the particulars of waiting for input and calling functions as a result of timeouts.

You will typically only have a single event loop in your application, even if you have more than one `MainLoop` running.

You can add your own files to watch to your event loop, with the `watch_file()` method. Using this interface gives you the special handling of `ExitMainLoop` and other exceptions when using Glib or Twisted.

SelectEventLoop

This event loop is based on `select.select()`. This is the default event loop created if none is passed to `MainLoop`.

```
# same as urwid.MainLoop(widget, event_loop=urwid.SelectEventLoop())
loop = urwid.MainLoop(widget)
```

See also:

`SelectEventLoop reference`

TwistedEventLoop

This event loop uses Twisted's reactor. It has been set up to emulate `SelectEventLoop`'s behaviour and will start the reactor and stop it on an error. This is not the standard way of using Twisted's reactor, so you may need to modify this behaviour for your application.

```
loop = urwid.MainLoop(widget, event_loop=urwid.TwistedEventLoop())
```

See also:

`TwistedEventLoop reference`

GLibEventLoop

This event loop uses GLib's event loop. This is useful if you are building an application that depends on DBus events, but don't want to base your application on Twisted.

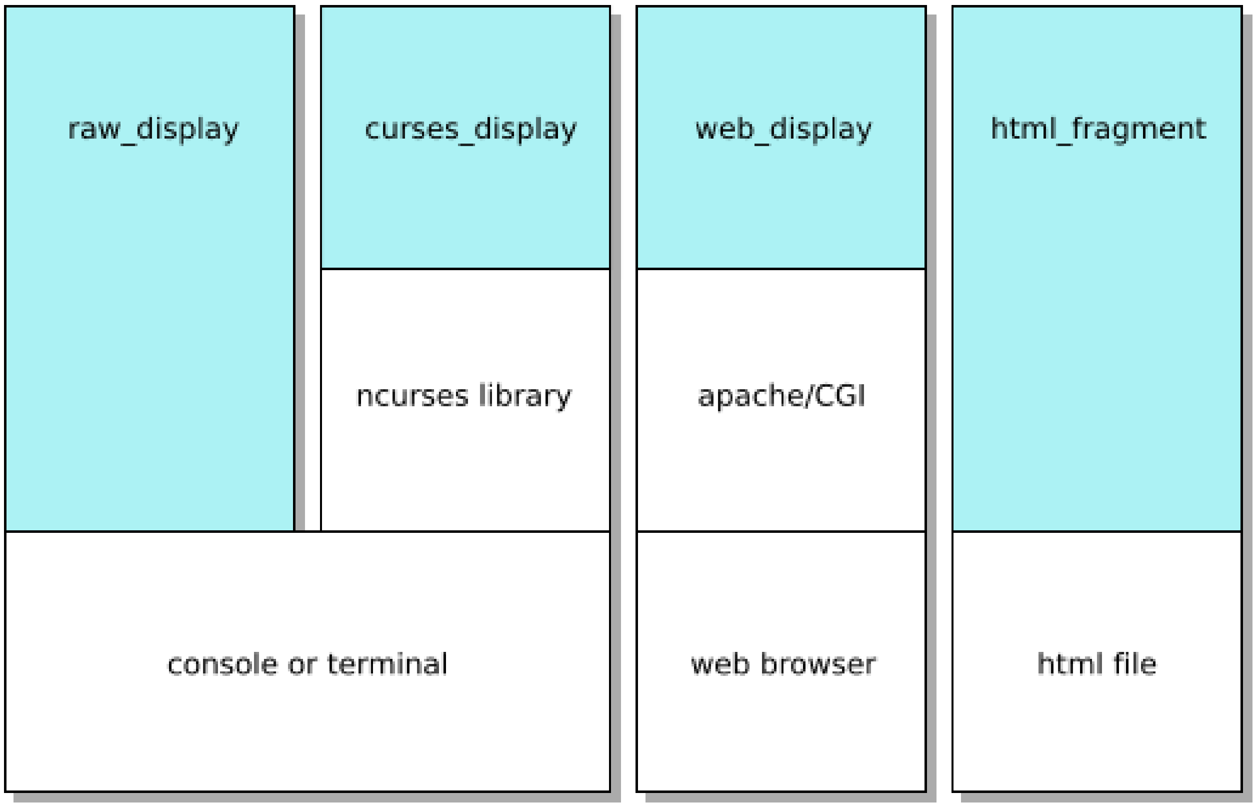
```
loop = urwid.MainLoop(widget, event_loop=urwid.GLibEventLoop())
```

See also:

`GLibEventLoop reference`

3.3 Display Modules

Urwid's display modules provide a layer of abstraction for drawing to the screen and reading user input. The display module you choose will depend on how you plan to use Urwid.



Typically you will select a display module by passing it to your `MainLoop` constructor, eg:

```
loop = MainLoop(widget, ..., screen=urwid.curses_display.Screen())
```

If you don't specify a display module, the default main loop will use `raw_display.Screen` by default

```
# These are the same
loop = MainLoop(widget, ...)
loop = MainLoop(widget, ..., screen=urwid.raw_display.Screen())
```

3.3.1 Raw and Curses Display Modules

Urwid has two display modules for displaying to terminals or the console.

The `raw_display.Screen` module is a pure-python display module with no external dependencies. It sends and interprets terminal escape sequences directly. This is the default display module used by `MainLoop`.

The `curses_display.Screen` module uses the curses or ncurses library provided by the operating system. The library does some optimization of screen updates and uses termcap to adjust to the user's terminal.

The (n)curses library will disable colors if it detects a monochrome terminal, so a separate set of attributes should be given for monochrome mode when registering a palette with `curses_display.Screen` High colors will not be used by the `curses_display.Screen` module. See *Setting a Palette* below.

This table summarizes the differences between the two modules:

	raw_display	curses_display
optimized C code	no	YES
compatible with any terminal	no	YES [1]
UTF-8 support	YES	YES [2]
bright foreground without bold	YES [3]	no
88- or 256-color support	YES	no
mouse dragging support	YES	no
external event loop support	YES	no

3.3.2 Other Display Modules

CGI Web Display Module `web_display`

The `urwid.web_display` module lets you run your application as a CGI script under Apache instead of running it in a terminal.

This module is a proof of concept. There are security and responsiveness issues that need to be resolved before this module is recommended for production use.

The tour.py and calc.py example programs demonstrate use of this module.

Screenshot Display Module `html_fragment`

Screenshots of Urwid interfaces can be rendered in plain HTML. The `html_fragment.HtmlGenerator` display module lets you do this by simulating user input and capturing the screen as fragments of HTML each time `html_fragemnt.HtmlGenerator.draw_screen()` is called.

These fragments may be included in HTML documents. They will be rendered properly by any browser that uses a monospaced font for text that appears in `<pre>` tags. HTML screenshots have text that is searchable and selectable in a web browser, and they will shrink and grow when a user changes their browser's text size.

The example screenshots are generated with this display module.

LCD Display Module `lcd_display`

Almost any device that displays characters in a grid can be used as a screen. The `lcd_display` module has some base classes for simple LCD character display devices and a complete implementation of a `lcd_display.CF635Screen` for Crystal Fontz 635 USB displays with 6 buttons.

The lcd_cf635.py example program demonstrates use of this module.

See also:

Urwid on a Crystalfontz 635 LCD

[1] if the termcap entry exists and TERM environment variable is set correctly
[2] if python is linked against the wide version of ncurses
[3] when using xterm or gnome-terminal

3.3.3 Setting a Palette

The `MainLoop` constructor takes a *palette* parameter that it passes to the `register_palette()` method of your display module.

A palette is a list of *display attribute* names and foreground and background settings. Display modules may be run in monochrome, normal or high color modes and you can set different foregrounds and backgrounds for each mode as part of your palette. eg:

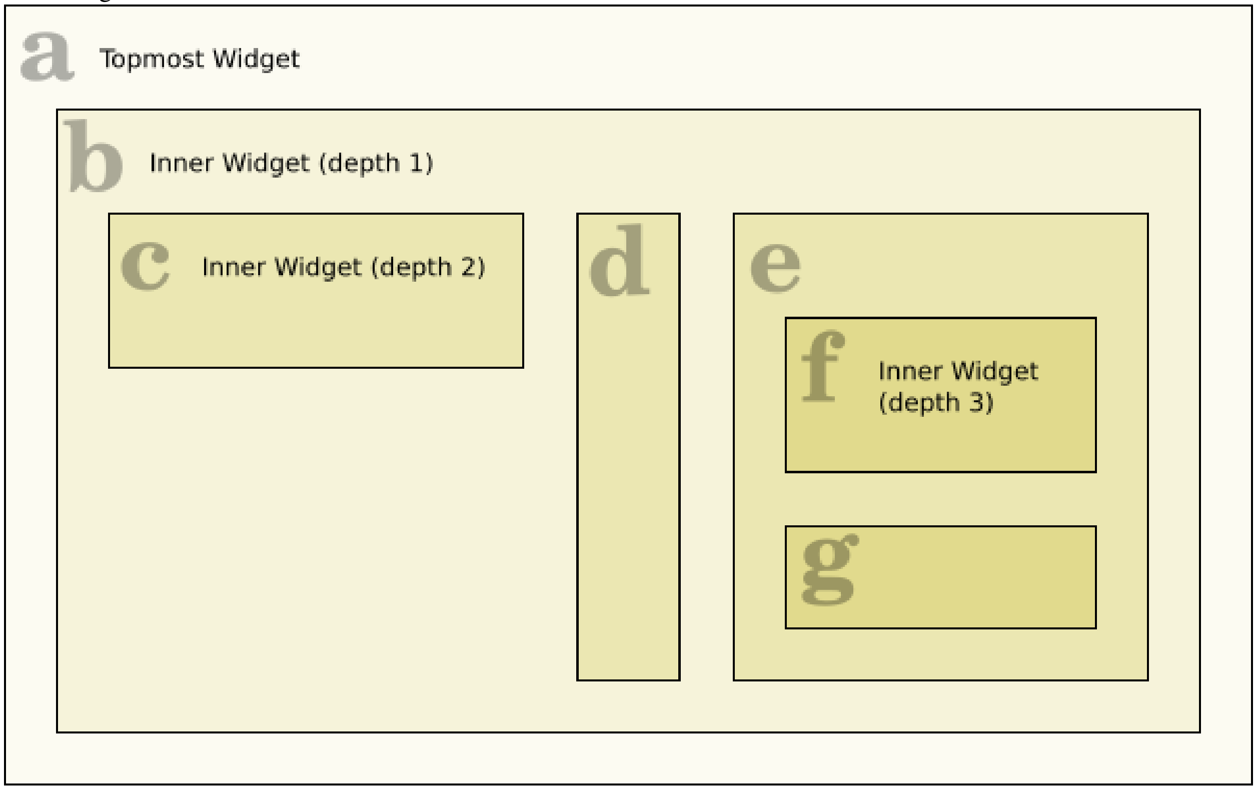
```
loop = MainLoop(widget, palette=[
    ('headings', 'white,underline', 'black', 'bold,underline'), # bold text in monochrome mode
    ('body_text', 'dark cyan', 'light gray'),
    ('buttons', 'yellow', 'dark green', 'standout'),
    ('section_text', 'body_text'), # alias to body_text
    ])
```

The *Display Attributes* section of this manual describes all the options available.

3.4 Widgets

3.4.1 Widget Layout

Urwid uses widgets to divide up the available screen space. This makes it easy to create a fluid interface that moves and changes with the user's terminal and font size.



The result of rendering a widget is a canvas suitable for displaying on the screen. When we render the topmost widget:

1. The topmost widget *(a)* is rendered the full size of the screen
2. *(a)* renders *(b)* any size up to the full size of the screen

3. *(b)* renders *(c)*, *(d)* and *(e)* dividing its available screen columns between them
4. *(e)* renders *(f)* and *(g)* dividing its available screen rows between them
5. *(e)* combines the canvases from *(f)* and *(g)* and returns them
6. *(b)* combines the canvases from *(c)*, *(d)* and *(e)* and returns them
7. *(a)* possibly modifies the canvas from *(b)* and returns it

Widgets *(a)*, *(b)* and *(e)* are called container widgets because they contain other widgets. Container widgets choose the size and position their contained widgets.

Container widgets must also keep track of which one of their contained widgets is in focus. The focus is used when handling keyboard input. If in the above example *(b)* 's focus widget is *(e)* and *(e)* 's focus widget is *(f)* then keyboard input will be handled this way:

1. The keypress is passed to the topmost widget *(a)*
2. *(a)* passes the keypress to *(b)*
3. *(b)* passes the keypress to *(e)*, its focus widget
4. *(e)* passes the keypress to *(f)*, its focus widget
5. *(f)* either handles the keypress or returns it
6. *(e)* has an opportunity to handle the keypress if it was returned from *(f)*
7. *(b)* has an opportunity to handle the keypress if it was returned from *(e)*
8. *(a)* has an opportunity to handle the keypress if it was returned from *(b)*

3.4.2 Box, Flow and Fixed Widgets

The size of a widget is measured in screen columns and rows. Widgets that are given an exact number of screen columns and rows are called box widgets. The topmost widget is always a box widget.

Much of the information displayed in a console user interface is text and the best way to display text is to have it flow from one screen row to the next. Widgets like this that require a variable number of screen rows are called flow widgets. Flow widgets are given a number of screen columns and can calculate how many screen rows they need.

Occasionally it is also useful to have a widget that knows how many screen columns and rows it requires, regardless of the space available. This is called a fixed widget.

Table 3.1: How a Widget's Size is Determined

sizing mode	width	height
`'box'`	container decides	container decides
`'flow'`	container decides	widget's `rows()` method
`'fixed'`	widget's `pack()` method	widget's `pack()` method

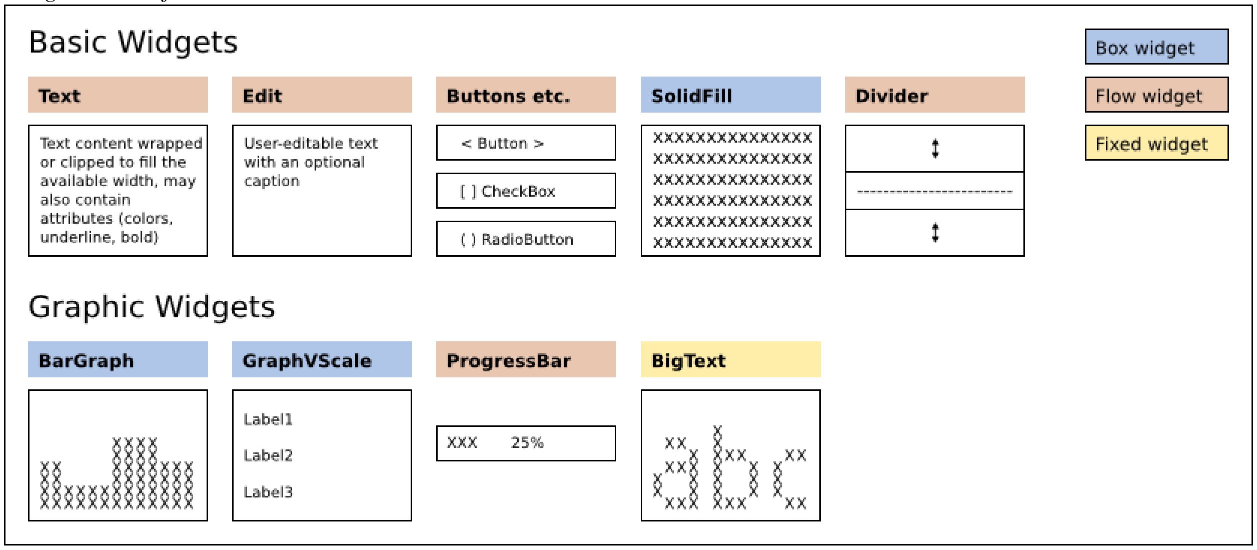
It is an Urwid convention to use the variables `maxcol` and `maxrow` to store a widget's size. Box widgets require both of `(maxcol, maxrow)` to be specified.

Flow widgets expect a single-element tuple `(maxcol,)` instead because they calculate their `maxrow` based on the `maxcol` value.

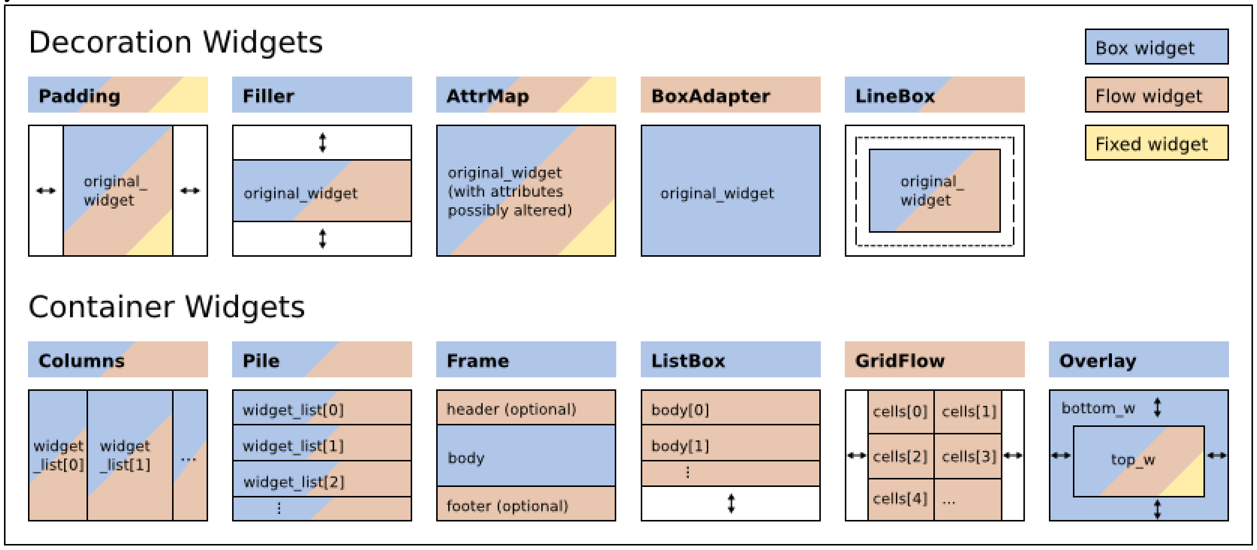
Fixed widgets expect the value `()` to be passed in to functions that take a size because they know their `maxcol` and `maxrow` values.

3.4.3 Included Widgets

Widget class reference



Basic and graphic widgets are the content with which users interact. They may also be used as part of custom widgets you create.



3.4.4 Decoration Widgets

Decoration widgets alter the appearance or position of a single other widget. The widget they wrap is available as the `original_widget` property. If you might be using more than one decoration widget you may use the `base_widget` property to access the "most" original_widget. `Widget.base_widget` points to `self` on all non-decoration widgets, so it is safe to use in any situation.

3.4.5 Container Widgets

Container widgets divide their available space between their child widgets. This is how widget layouts are defined. When handling selectable widgets container widgets also keep track of which of their child widgets is in focus.

Container widgets may be nested, so the actual widget in focus may be many levels below the topmost widget.

Urwid's container widgets have a common API you can use, regardless of the container type. Backwards compatibility is still maintained for the old container-specific ways of accessing and modifying contents, but this API is now the preferred way of modifying and traversing containers.

```
container.focus
```

is a read-only property that returns the widget in focus for this container. Empty containers and non-container widgets (that inherit from Widget) return `None`.

```
container.focus_position
```

is a read/write property that provides access to the position of the container's widget in focus. This will often be a integer value but may be any object. `Columns`, `Pile`, `GridFlow`, `Overlay` and `ListBox` with a `SimpleListWalker` or `SimpleFocusListWalker` as its body use integer positions. `Frame` uses `'body'`, `'header'` and `'footer'`; `ListBox` with a custom list walker will use the positions the list walker returns.

Reading this value on an empty container or on any non-container widgets (that inherit from Widget) raises an IndexError. Writing to this property with an invalid position will also raise an IndexError. Writing a new value automatically marks this widget to be redrawn and will be reflected in `container.focus`.

```
container.contents
```

is a read-only property (read/write in some cases) that provides access to a mapping- or list-like object that contains the child widgets and the options used for displaying those widgets in this container. The mapping- or list-like object always allows reading from positions with the usual `__getitem__()` method and may support assignment and deletion with `__setitem__()` and `__delitem__()` methods. The values are `(child widget, option)` tuples. When this object or its contents are modified the widget is automatically flagged to be redrawn.

`Columns`, `Pile` and `GridFlow` allow assigning an iterable to `container.contents` to overwrite the values in with the ones provided.

`Columns`, `Pile`, `GridFlow`, `Overlay` and `Frame` support `container.contents` item assignment and deletion.

```
container.options(...)
```

is a method that returns options objects for use in items added to `container.contents`. The arguments are specific to the container type, and generally match the `__init__()` arguments for the container. The objects returned are currently tuples of strings and integers or `None` for containers without child widget options. This method exists to allow future versions of Urwid to add new options to existing containers. Code that expects the option tuples to remain the same size will fail when new options are added, so defensive programming with options tuples is strongly encouraged.

```
container.__getitem__(x)
# a.k.a.
container[x]
```

is a short-cut method behaving identically to: `container.contents[x][0].base_widget`. Which means roughly "give me the child widget at position *x* and skip all the decoration widgets wrapping it". Decoration widgets include `Padding`, `Filler`, `AttrMap` etc.

```
container.get_focus_path()
```

is a method that returns the focus position for this container *and* all child containers along the path defined by their focus settings. This list of positions is the closest thing we have to the singular widget-in-focus in other UI frameworks, because the ultimate widget in focus in Urwid depends on the focus setting of all its parent container widgets.

```
container.set_focus_path(p)
```

is a method that assigns to the focus_position property of each container along the path given by the list of positions *p*. It may be used to restore focus to a widget as returned by a previous call to `container.get_focus_path()`.

```
container.__iter__()
# typically
for x in container: ...

container.__reversed__()
# a.k.a
reversed(container)
```

are methods that allow iteration over the *positions* of this container. Normally the order of the positions generated by __reversed__() will be the opposite of __iter__(). The exception is the case of `ListBox` with certain custom list walkers, and the reason goes back to the original way list walker interface was defined. Note that a custom list walker might also generate an unbounded number of positions, so care should be used with this interface and `ListBox`.

Pile Widgets

`Pile` widgets are used to combine multiple widgets by stacking them vertically. A Pile can manage selectable widgets by keeping track of which widget is in focus and it can handle moving the focus between widgets when the user presses the *UP* and *DOWN* keys. A Pile will also work well when used within a `ListBox`.

A Pile is selectable only if its focus widget is selectable. If you create a Pile containing one Text widget and one Edit widget the Pile will choose the Edit widget as its default focus widget.

Columns Widgets

`Columns` widgets may be used to arrange either flow widgets or box widgets horizontally into columns. Columns widgets will manage selectable widgets by keeping track of which column is in focus and it can handle moving the focus between columns when the user presses the *LEFT* and *RIGHT* keys. Columns widgets also work well when used within a `ListBox`.

Columns widgets are selectable only if the column in focus is selectable. If a focus column is not specified the first selectable widget will be chosen as the focus column.

GridFlow Widgets

The `GridFlow` widget is a flow widget designed for use with `Button`, `CheckBox` and `RadioButton` widgets. It renders all the widgets it contains the same width and it arranges them from left to right and top to bottom.

The GridFlow widget uses Pile, Columns, Padding and Divider widgets to build a display widget that will handle the keyboard input and rendering. When the GridFlow widget is resized it regenerates the display widget to accommodate the new space.

Overlay Widgets

The `Overlay` widget is a box widget that contains two other box widgets. The bottom widget is rendered the full size of the Overlay widget and the top widget is placed on top, obscuring an area of the bottom widget. This widget can be used to create effects such as overlapping “windows” or pop-up menus.

The Overlay widget always treats the top widget as the one in focus. All keyboard input will be passed to the top widget.

If you want to use a flow flow widget for the top widget, first wrap the flow widget with a `Filler` widget.

3.4.6 ListBox Contents

`ListBox` is a box widget that contains flow widgets. Its contents are displayed stacked vertically, and the `ListBox` allows the user to scroll through its content. One of the flow widgets displayed in the `ListBox` is its focus widget.

ListBox Focus and Scrolling

The `ListBox` is a box widget that contains flow widgets. Its contents are displayed stacked vertically, and the `ListBox` allows the user to scroll through its content. One of the flow widgets displayed in the `ListBox` is the focus widget. The `ListBox` passes key presses to the focus widget to allow the user to interact with it. If the focus widget does not handle a keypress then the `ListBox` may handle the keypress by scrolling and/or selecting another widget to become the focus widget.

The `ListBox` tries to do the most sensible thing when scrolling and changing focus. When the widgets displayed are all `Text` widgets or other unselectable widgets then the `ListBox` will behave like a web browser does when the user presses *UP*, *DOWN*, *PAGE UP* and *PAGE DOWN*: new text is immediately scrolled in from the top or bottom. The `ListBox` chooses one of the visible widgets as its focus widget when scrolling. When scrolling up the `ListBox` chooses the topmost widget as the focus, and when scrolling down the `ListBox` chooses the bottommost widget as the focus.

The `ListBox` remembers the location of the widget in focus as either an “offset” or an “inset”. An offset is the number of rows between the top of the `ListBox` and the beginning of the focus widget. An offset of zero corresponds to a widget with its top aligned with the top of the `ListBox`. An inset is the fraction of rows of the focus widget that are “above” the top of the `ListBox` and not visible. The `ListBox` uses this method of remembering the focus widget location so that when the `ListBox` is resized the text displayed will stay roughly aligned with the top of the `ListBox`.

When there are selectable widgets in the `ListBox` the focus will move between the selectable widgets, skipping the unselectable widgets. The `ListBox` will try to scroll all the rows of a selectable widget into view so that the user can see the new focus widget in its entirety. This behavior can be used to bring more than a single widget into view by using composite widgets to combine a selectable widget with other widgets that should be displayed at the same time.

Dynamic ListBox with ListWalker

While the `ListBox` stores the location of its focus widget, it does not directly store the actual focus widget or other contents of the `ListBox`. The storage of a `ListBox`‘s content is delegated to a “List Walker” object. If a list of widgets is passed to the `ListBox` constructor then it creates a `SimpleListWalker` object to manage the list.

When the `ListBox` is rendering a canvas or handling input it will:

1. Call the `get_focus()` method of its list walker object. This method will return the focus widget and a position object.
2. Optionally call the `get_prev()` method of its List Walker object one or more times, initially passing the focus position and then passing the new position returned on each successive call. This method will return the widget and position object “above” the position passed.
3. Optionally call the `get_next()` method of its List Walker object one or more times, similarly, to collect widgets and position objects “below” the focus position.
4. Optionally call the `set_focus()` method passing one of the position objects returned in the previous steps.

This is the only way the `ListBox` accesses its contents, and it will not store copies of any of the widgets or position objects beyond the current rendering or input handling operation.

The `SimpleListWalker` stores a list of widgets, and uses integer indexes into this list as its position objects. It stores the focus position as an integer, so if you insert a widget into the list above the focus position then you need to remember to increment the focus position in the `SimpleListWalker` object or the contents of the `ListBox` will shift.

A custom List Walker object may be passed to the `ListBox` constructor instead of a plain list of widgets. List Walker objects must implement the *List Walker Interface*.

The fib.py example program demonstrates a custom list walker that doesn't store any widgets. It uses a tuple of two successive Fibonacci numbers as its position objects and it generates Text widgets to display the numbers on the fly. The result is a `ListBox` that can scroll through an unending list of widgets.

The edit.py example program demonstrates a custom list walker that loads lines from a text file only as the user scrolls them into view. This allows even huge files to be opened almost instantly.

The browse.py example program demonstrates a custom list walker that uses a tuple of strings as position objects, one for the parent directory and one for the file selected. The widgets are cached in a separate class that is accessed using a dictionary indexed by parent directory names. This allows the directories to be read only as required. The custom list walker also allows directories to be hidden from view when they are "collapsed".

Setting the Focus

The easiest way to change the current `ListBox` focus is to call the `ListBox.set_focus()` method. This method doesn't require that you know the `ListBox`'s current dimensions `(maxcol, maxrow)`. It will wait until the next call to either keypress or render to complete setting the offset and inset values using the dimensions passed to that method.

The position object passed to `set_focus()` must be compatible with the List Walker object that the `ListBox` is using. For `SimpleListWalker` the position is the integer index of the widget within the list.

The *coming_from* parameter should be set if you know that the old position is "above" or "below" the previous position. When the `ListBox` completes setting the offset and inset values it tries to find the old widget among the visible widgets. If the old widget is still visible, if will try to avoid causing the `ListBox` contents to scroll up or down from its previous position. If the widget is not visible, then the `ListBox` will:

- Display the new focus at the bottom of the `ListBox` if *coming_from* is "above".
- Display the new focus at the top of the `ListBox` if *coming_from* is "below".
- Display the new focus in the middle of the `ListBox` if coming_from is `None`.

If you know exactly where you want to display the new focus widget within the `ListBox` you may call `ListBox.set_focus_valign()`. This method lets you specify the *top*, *bottom*, *middle*, a relative position or the exact number of rows from the top or bottom of the `ListBox`.

List Walkers

`ListBox` does not manage the widgets it displays directly, instead it passes that task to a class called a "list walker". List walkers keep track of the widget in focus and provide an opaque position object that the `ListBox` may use to iterate through widgets above and below the focus widget.

A `SimpleFocusListWalker` is a list walker that behaves like a normal Python list. It may be used any time you will be displaying a moderate number of widgets.

If you need to display a large number of widgets you should implement your own list walker that manages creating widgets as they are requested and destroying them later to avoid excessive memory use.

List walkers may also be used to display tree or other structures within a `ListBox`. A number of the example programs demonstrate the use of custom list walker classes.

See also:

`ListWalker base class reference`

List Walker Interface

List Walker API Version 1

This API will remain available and is still the least restrictive option for the programmer. Your class should subclass `ListWalker`. Whenever the focus or content changes you are responsible for calling `ListWalker._modified()`.

`MyV1ListWalker.`**`get_focus`**`()`
: return a `(widget, position)` tuple or `(None, None)` if empty

`MyV1ListWalker.`**`set_focus`**`(`*position*`)`
: set the focus and call `self._modified()` or raise an `IndexError`.

`MyV1ListWalker.`**`get_next`**`(`*position*`)`
: return the `(widget, position)` tuple below *position* passed or `(None, None)` if there is none.

`MyV1ListWalker.`**`get_prev`**`(`*position*`)`
: return the `(widget, position)` tuple above *position* passed or `(None, None)` if there is none.

List Walker API Version 2

This API is an attempt to remove some of the duplicate code that V1 requires for many users. List walker API V1 will be implemented automatically by subclassing `ListWalker` and implementing the V2 methods. Whenever the focus or content changes you are responsible for calling `ListWalker._modified()`.

`MyV2ListWalker.`**`__getitem__`**`(`*position*`)`
: return widget at *position* or raise an `IndexError` or `KeyError`

`MyV2ListWalker.`**`next_position`**`(`*position*`)`
: return the position below passed *position* or raise an `IndexError` or `KeyError`

`MyV2ListWalker.`**`prev_position`**`(`*position*`)`
: return the position above passed *position* or raise an `IndexError` or `KeyError`

`MyV2ListWalker.`**`set_focus`**`(`*position*`)`
: set the focus and call `self._modified()` or raise an `IndexError`.

`MyV2ListWalker.`**`focus`**
: attribute or property containing the focus position, or define `MyV1ListWalker.get_focus()` as above

List Walker Iteration

There is an optional iteration helper method that may be defined in any list walker. When this is defined it will be used by `ListBox.__iter__()` and `ListBox.__reversed__()`:

`MyV2ListWalker.`**`positions`**`(`*reverse=False*`)`
: return a forward or reverse iterable of positions

3.4.7 Custom Widgets

Widgets in Urwid are easiest to create by extending other widgets. If you are making a new type of widget that can use other widgets to display its content, like a new type of button or control, then you should start by extending `WidgetWrap` and passing the display widget to its constructor.

The `Widget` interface is described in detail in the `Widget base class reference` and is useful if you're looking to modify the behavior of an existing widget, build a new widget class from scratch or just want a better understanding of the library.

One Urwid design choice that stands out is that widgets typically have no size. Widgets don't store their size on screen, and instead are passed that information when they need it.

This choice has some advantages:

- widgets may be reused in different locations
- reused widgets only need to be rendered once per size displayed
- widgets don't need to know their parents
- less data to store and update
- no worrying about widgets that haven't received their size yet
- same widgets could be displayed at different sizes to different users simultaneously

It also has disadvantages:

- difficult to determine a widget's size on screen
- more parameters to parse
- duplicated size calculations across methods

For determining a widget's size on screen it is possible to look up the size(s) it was rendered at in the `CanvasCache`. There are plans to address some of the duplicated size handling code in the container widgets in a future Urwid release.

The same holds true for a widget's focus state, so that too is passed in to functions that need it.

Modifying Existing Widgets

The easiest way to create a custom widget is to modify an existing widget. This can be done by either subclassing the original widget or by wrapping it. Subclassing is appropriate when you need to interact at a very low level with the original widget, such as if you are creating a custom edit widget with different behavior than the usual Edit widgets. If you are creating a custom widget that doesn't need tight coupling with the original widget then wrapping is more appropriate.

The `WidgetWrap` class simplifies wrapping existing widgets. You can create a custom widget simply by creating a subclass of WidgetWrap and passing a widget into WidgetWrap's constructor.

This is an example of a custom widget that uses WidgetWrap:

```python
import urwid

class QuestionnaireItem(urwid.WidgetWrap):
    def __init__(self):
        self.options = []
        unsure = urwid.RadioButton(self.options, u"Unsure")
        yes = urwid.RadioButton(self.options, u"Yes")
        no = urwid.RadioButton(self.options, u"No")
        display_widget = urwid.GridFlow([unsure, yes, no], 15, 3, 1, 'left')
        urwid.WidgetWrap.__init__(self, display_widget)

    def get_state(self):
        for o in self.options:
            if o.get_state() is True:
                return o.get_label()
```

The above code creates a group of RadioButtons and provides a method to query the state of the buttons.

Widgets from Scratch

Widgets must inherit from `Widget`. Box widgets must implement `Widget.selectable()` and `Widget.render()` methods, and flow widgets must implement `Widget.selectable()`, `Widget.render()` and `Widget.rows()` methods.

The default `Widget.sizing()` method returns a set of sizing modes supported from `self._sizing`, so we define `_sizing` attributes for our flow and box widgets below.

```
import urwid

class Pudding(urwid.Widget):
    _sizing = frozenset(['flow'])

    def rows(self, size, focus=False):
        return 1

    def render(self, size, focus=False):
        (maxcol,) = size
        num_pudding = maxcol / len("Pudding")
        return urwid.TextCanvas(["Pudding" * num_pudding], maxcol=maxcol)


class BoxPudding(urwid.Widget):
    _sizing = frozenset(['box'])

    def render(self, size, focus=False):
        (maxcol, maxrow) = size
        num_pudding = maxcol / len("Pudding")
        return urwid.TextCanvas(["Pudding" * num_pudding] * maxrow,
                                maxcol=maxcol)
```

The above code implements two widget classes. Pudding is a flow widget and BoxPudding is a box widget. Pudding will render as much “Pudding” as will fit in a single row, and BoxPudding will render as much “Pudding” as will fit into the entire area given.

Note that the rows and render methods’ focus parameter must have a default value of False. Also note that for flow widgets the number of rows returned by the rows method must match the number of rows rendered by the render method.

To improve the efficiency of your Urwid application you should be careful of how long your `rows()` methods take to execute. The `rows()` methods may be called many times as part of input handling and rendering operations. If you are using a display widget that is time consuming to create you should consider caching it to reduce its impact on performance.

It is possible to create a widget that will behave as either a flow widget or box widget depending on what is required:

```
import urwid

class MultiPudding(urwid.Widget):
    _sizing = frozenset(['flow', 'box'])

    def rows(self, size, focus=False):
        return 1

    def render(self, size, focus=False):
        if len(size) == 1:
            (maxcol,) = size
            maxrow = 1
        else:
            (maxcol, maxrow) = size
        num_pudding = maxcol / len("Pudding")
        return urwid.TextCanvas(["Pudding" * num_pudding] * maxrow,
                                maxcol=maxcol)
```

MultiPudding will work in place of either Pudding or BoxPudding above. The number of elements in the size tuple determines whether the containing widget is expecting a flow widget or a box widget.

Selectable Widgets

Selectable widgets such as Edit and Button widgets allow the user to interact with the application. A widget is selectable if its selectable method returns True. Selectable widgets must implement the `Widget.keypress()` method to handle keyboard input.

```
import urwid

class SelectablePudding(urwid.Widget):
    _sizing = frozenset(['flow'])
    _selectable = True

    def __init__(self):
        self.pudding = "pudding"

    def rows(self, size, focus=False):
        return 1

    def render(self, size, focus=False):
        (maxcol,) = size
        num_pudding = maxcol / len(self.pudding)
        pudding = self.pudding
        if focus:
            pudding = pudding.upper()
        return urwid.TextCanvas([pudding * num_pudding],
            maxcol=maxcol)

    def keypress(self, size, key):
        (maxcol,) = size
        if len(key) > 1:
            return key
        if key.lower() in self.pudding:
            # remove letter from pudding
            n = self.pudding.index(key.lower())
            self.pudding = self.pudding[:n] + self.pudding[n+1:]
            if not self.pudding:
                self.pudding = "pudding"
            self._invalidate()
        else:
            return key
```

The SelectablePudding widget will display its contents in uppercase when it is in focus, and it allows the user to “eat” the pudding by pressing each of the letters *P*, *U*, *D*, *D*, *I*, *N* and *G* on the keyboard. When the user has “eaten” all the pudding the widget will reset to its initial state.

Note that keys that are unhandled in the keypress method are returned so that another widget may be able to handle them. This is a good convention to follow unless you have a very good reason not to. In this case the *UP* and *DOWN* keys are returned so that if this widget is in a `ListBox` the `ListBox` will behave as the user expects and change the focus or scroll the `ListBox`.

Widget Displaying the Cursor

Widgets that display the cursor must implement the `Widget.get_cursor_coords()` method. Similar to the rows method for flow widgets, this method lets other widgets make layout decisions without rendering the entire widget. The `ListBox` widget in particular uses get_cursor_coords to make sure that the cursor is visible within its focus widget.

```
import urwid

class CursorPudding(urwid.Widget):
    _sizing = frozenset(['flow'])
    _selectable = True

    def __init__(self):
        self.cursor_col = 0

    def rows(self, size, focus=False):
        return 1

    def render(self, size, focus=False):
        (maxcol,) = size
        num_pudding = maxcol / len("Pudding")
        cursor = None
        if focus:
            cursor = self.get_cursor_coords(size)
        return urwid.TextCanvas(["Pudding" * num_pudding], [], cursor, maxcol)

    def get_cursor_coords(self, size):
        (maxcol,) = size
        col = min(self.cursor_col, maxcol - 1)
        return col, 0

    def keypress(self, size, key):
        (maxcol, ) = size
        if key == 'left':
            col = self.cursor_col - 1
        elif key == 'right':
            col = self.cursor_col + 1
        else:
            return key
        self.cursor_x = max(0, min(maxcol - 1, col))
        self._invalidate()
```

CursorPudding will let the user move the cursor through the widget by pressing *LEFT* and *RIGHT*. The cursor must only be added to the canvas when the widget is in focus. The get_cursor_coords method must always return the same cursor coordinates that render does.

A widget displaying a cursor may choose to implement `Widget.get_pref_col()`. This method returns the preferred column for the cursor, and is called when the focus is moving up or down off this widget.

Another optional method is `Widget.move_cursor_to_coords()`. This method allows other widgets to try to position the cursor within this widget. The `ListBox` widget uses `Widget.move_cursor_to_coords()` when changing focus and when the user pressed *PAGE UP* or *PAGE DOWN*. This method must return `True` on success and `False` on failure. If the cursor may be placed at any position within the row specified (not only at the exact column specified) then this method must move the cursor to that position and return `True`.

```
    def get_pref_col(self, (maxcol,)):
        return self.cursor_x

    def move_cursor_to_coords(self, (maxcol,), col, row):
        assert row == 0
        self.cursor_x = col
        return True
```

3.4.8 Widget Metaclass

The `Widget` base class has a metaclass defined that creates a `__super` attribute for calling your superclass: `self.__super` is the same as the usual `super(MyClassName, self)`. This shortcut is of little use with Python 3's new `super()` syntax, but will likely be retained for backwards compatibility in future versions.

This metaclass also uses `MetaSignal` to allow signals to be defined as a list of signal names in a `signals` class attribute. This is equivalent to calling `register_signal()` with the class name and list of signals and all those defined in superclasses after the class definition.

See also:

`Widget metaclass WidgetMeta`

3.5 User Input

All input from the user is parsed by a display module, and returned from either the `get_input()` or `get_input_nonblocking()` methods as a list. Window resize events are also included in the user input.

The `MainLoop` class will take this input and pass each item to the widget methods `keypress()` or `mouse_event()`. You may filter input (possibly removing or altering it) before it is passed to the widgets, or can catch unhandled input by passing functions into the `MainLoop` constructor. If the window was resized `MainLoop` will query the new display size and update the screen.

There may be more than one keystroke or mouse event processed at a time, and each is sent as a separate item in the list.

3.5.1 Keyboard Input

Not all keystrokes are sent by a user's terminal to the program, and which keys are sent varies from terminal to terminal, but Urwid will report any keys that are sent.

Key pressed	Input returned
H	`'h'`
SHIFT+H	`'H'`
SPACE	`' '`
ENTER	`'enter'`
UP	`'up'`
PAGE DOWN	`'page down'`
F5	`'f5'`
SHIFT+F5	`'shift f5'`
CTRL+SHIFT+F5	`'shift ctrl f5'`
ALT+J	`'meta j'`

With Unicode *text encoding* you will also receive Unicode strings for any non-ASCII characters:

Key pressed	Input returned

é	`u'é'`
	`u''`
	`u''`

With non-Unicode *text encoding* characters will be sent as-is in the original encoding.

Key pressed	Input returned (each in its own encoding)

é	`'é'`
	`''`
	`''` (two bytes)

Urwid does not try to convert this text to Unicode to avoid losing any information. If you want the input converted to Unicode in all cases you may create an input filter to do so.

3.5.2 Mouse Input

Mouse input is sent as a (*event*, *button*, *x*, *y*) tuple. *event* is a string describing the event. If the *SHIFT*, *ALT* or *CTRL* keys are held when a mouse event is sent then *event* may be prefixed by `'shift '`, `'meta '` or `'ctrl'`. *button* is a number from 1 to 5. *x* and *y* are character coordinates starting from `(0, 0)` at the top-left of the screen.

Support for the right-mouse button and use of modifier keys is poor in many terminals and some users don't have a middle mouse button, so these shouldn't be relied on.

`'mouse press'` Events

A mouse button was pressed.

button number	Mouse button
1	Left button
2	Middle button
3	Right button
4	Scroll wheel up [4]
5	Scroll wheel down [1]

`'mouse release'` Events

Mouse release events will often not have information about which button was released. In this case *button* will be set to 0.

[4]typically no corresponding release event is sent

`'mouse drag'` Events

In the rare event that your user is using a terminal that can send these events you can use them to track their mouse dragging from one character cell to the next across the screen. Be aware that you might see *x* and/or *y* coordinates one position off the screen if the user drags their mouse to the edge.

3.6 Text Layout

Mapping a text string to screen coordinates within a widget is called text layout. The `Text` widget's default layout class supports aligning text to the left, center or right, and can wrap text on space characters, at any location, or clip text that is off the edge.

```
Text("Showing some different alignment modes", align=...)

align='left' (default)
+-----------------+   +------------------------+
|Showing some     |   |Showing some different  |
|different        |   |alignment modes         |
|alignment modes  |   +------------------------+
+-----------------+

align='center'
+-----------------+   +------------------------+
|  Showing some   |   | Showing some different |
|   different     |   |    alignment modes     |
|alignment modes  |   +------------------------+
+-----------------+

align='right'
+-----------------+   +------------------------+
|     Showing some|   |  Showing some different|
|        different|   |         alignment modes|
| alignment modes|   +------------------------+
+-----------------+

Text("Showing some different wrapping modes\nnewline", wrap=...)

wrap='space' (default)
+-----------------+   +------------------------+
|Showing some     |   |Showing some different  |
|different        |   |wrapping modes          |
|wrapping modes   |   |newline                 |
|newline          |   +------------------------+
+-----------------+

wrap='any'
+-----------------+   +------------------------+
|Showing some dif|   |Showing some different w|
|ferent wrapping  |   |rapping modes           |
|modes            |   |newline                 |
|newline          |   +------------------------+
+-----------------+

wrap='clip'
+-----------------+   +------------------------+
|Showing some dif|   |Showing some different w|
|newline         |   |newline                 |
+----------------+   +------------------------+
```

If this is good enough for your application feel free to skip the rest of this section.

See also:

`Text widget reference`

3.6.1 Custom Text Layouts

The `StandardTextLayout` is set as the class variable `Text.layout`. Individual `Text` widgets may use a different layout class, or you can change the default by setting the `Text.layout` class variable itself.

A custom text layout class should extend the `TextLayout` base class and return text layout structures from its `layout()` method.

See also:

`TextLayout reference`

3.6.2 Text Layout Structures

```
"This is how a string of text might be displayed"
0----5---10---15---20---25---30---35---40---45--

0----5---10---15---+   right_aligned_text_layout = [
|     This is how a|     [(5, 0), (13, 0, 13)],
|    string of text|     [(4, 13), (14, 14, 28)],
|might be displayed|     [(18, 29, 47)]
+------------------+   ]
```

The mapping from a text string to where that text will be displayed in the widget is expressed as a text layout structure.

Text layout structures are used both for rendering `Text` widgets and for mapping `(x, y)` positions within a widget back to the corresponding offsets in the text. The latter is used when moving the cursor in `Edit` widgets up and down or by clicking with the mouse.

A text layout structure is a list of one or more line layouts. Each line layout corresponds to a row of text in the widget, starting from its top.

A line layout is a list zero or more of the following tuples, each expressing text to be displayed from left to right:

1. (*column width*, *starting text offset*, *ending text offset*)
2. (*column width of space characters to insert*, *text offset* or `None`)
3. (*column width*, *text offset*, *new text to insert*)‘‘

Tuple A displays a segment of text from the `Text` widget. Column width is explicitly specified because some characters within the text may be zero width or double width.

Tuple B inserts any number of space characters, and if those characters correspond to an offset within the text, that may be specified.

Tuple C allows insertion of arbitrary text. This could be used for hyphenating split words or any other effect not covered by A or B. The `StandardTextLayout` does not currently use this tuple in its line layouts.

See also:

`TextLayout reference`, `StandardTextLayout reference`

3.7 Encodings Supported

Urwid has a single global setting for text encoding that is set on start-up based on the configured locale. You may change that setting with the `set_encoding()` method. eg.

```
urwid.set_encoding("UTF-8")
```

There are two distinct modes of handling encodings with Urwid: Unicode or Pass-through. The mode corresponds to using Unicode strings or normal strings in your widgets.

```
txt_a = urwid.Text(u"El Niño")
txt_b = urwid.Text("El Niño")
```

`txt_a` will be automatically encoded when it is displayed (Unicode mode).

`txt_b` is **assumed** to be in the encoding the user is expecting and passed through as-is (Pass-through mode). If the encodings are different then the user will see "mojibake" (garbage) on their screen.

The only time it makes sense to use pass-through mode is if you're handling an encoding that does not round-trip to Unicode properly, or if you're absolutely sure you know what you're doing.

3.7.1 Unicode Support

Urwid has a basic understanding of character widths so that the text layout code can properly wrap and display most text. There is currently no support for right-to-left text.

You should be able to use any valid Unicode characters that are present in the global encoding setting in your widgets, with the addition of some common DEC graphic characters:

```
\u00A3 (£), \u00B0 (°), \u00B1 (±), \u00B7 (·), \u03C0 (π),
\u2260 (), \u2264 (), \u2265 (), \u23ba (), \u23bb (),
\u23bc (), \u23bd (_), \u2500 (−), \u2502 (|), \u250c (),
\u2510 (), \u2514 (), \u2518 (), \u251c (), \u2524 (),
\u252c (), \u2534 (), \u253c (), \u2592 (), \u25c6 ()
```

If you use these characters with a non-UTF-8 encoding they will be sent using the alternate character set sequences supported by some terminals.

3.7.2 Pass-through Support

Supported encodings for pass-through mode:

- UTF-8 (narrow and wide characters)
- ISO-8859-*
- EUC-JP (JISX 0208 only)
- EUC-KR
- EUC-CN (aka CN-GB)
- EUC-TW (CNS 11643 plain 1 only)
- GB2312
- GBK
- BIG5

- UHC

In pass-through mode Urwid must still calculate character widths. For UTF-8 mode the widths are specified in the Unicode standard. For ISO-8859-* all bytes are assumed to be 1 column wide characters. For the remaining supported encodings any byte with the high-bit set is considered to be half of a 2-column wide character.

The additional plains in EUC are not currently supported.

3.7.3 Future Work

Text encoding should be a per-screen (display module) setting, not a global setting. It should be possible to simultaneously support different encodings on different screens with Urwid. Making this work involves possibly changing the function signature of many widget methods, because encoding needs to be specified along with size and focus.

Device-specific encodings should also be possible for Unicode mode. The LCD display module in development drives a device with a non-standard mapping of Unicode code points to 8-bit values, but it should still be possible to use a Unicode text to display the characters it supports.

3.8 Display Attributes

Urwid supports a number of common display attributes in monochrome, 16-color, 88-color and 256-color modes.

You are encouraged to provide support for as many of these modes as you like, while allowing your interface to degrade gracefully by providing command line arguments or other interfaces to switch modes.

When setting up a palette with `MainLoop` (or directly on your screen instance), you may specify attributes for 16-color, monochrome and high color modes. You can then switch between these modes with `screen.set_terminal_properties()`, where `screen` is your screen instance or `MainLoop.screen`.

See also:

`register_palette() reference`,

3.8.1 Using Display Attributes

Once you have defined a palette you may use the its display attribute names anywhere that expects a display attribute. When no display attribute is defined `None` is used as a default display attribute.

`None` will typically be rendered with the terminal's default foreground and background colors.

You can also specify an exact foreground and background using an `AttrSpec` instance instead of a display attribute name. Using `AttrSpec` instances in your code may be trickier than using your screen's palette because you must know which mode (number of colors) the screen is in.

Text Markup

A `Text` widget can specify which display attributes each part of the text will use with the format defined in `Text class reference`. Some examples:

```
Text(u"a simple string with default attribute")
```

The string and space around will use the `None` default display attribute which usually appears in the terminal's default foreground and background.

```
Text(('attr1', u"a string in display attribute attr1"))
```

The string will appear with foreground and backgrounds specified in the display module's palette for `'attr1'`, but the space around (before/after) the text will appear with the default display attribute.

```
Text([u"a simple string ", ('attr1', u"ending with attr1")])
```

The first three words have the default display attribute and the last three words have display attribute `'attr1'`.

```
Text([('attr1', u"start in attr1 "), ('attr2', u"end in attr2")])
```

The first three words have display attribute `'attr1'` and the last three words have display attribute `'attr2'`.

```
Text(('attr1', [u"nesting example ", ('attr2', u"inside"), u" outside"]))
```

When markup is nested only the innermost attribute applies. Here `"inside"` has attribute `'attr2'` and all the rest of the text has attribute `'attr1'`.

Assigning Display Attributes with AttrMap

If you want a whole widget to be assigned a display attribute, or if you want to change one or more display attributes to other display attributes, you can wrap your widget in an `AttrMap` widget. `Text` widgets have no way to specify a display attribute for the whitespace around the text caused by alignment and wrapping so `AttrMap` may be used. Some examples:

```
AttrMap(Text(u"hello"), 'attr1')
```

The whole `Text` widget will have display attribute `'attr1'` including whitespace around the `"hello"` text.

```
AttrMap(Text(('attr1', u"hello")), 'attr2')
```

The `u"hello"` text will appear with display attribute `'attr1'` and all surrounding whitespace will appear with display attribute `'attr2'`.

```
AttrMap(Text([('attr1', u"hello"), u" world"]), {'attr1': 'attr2'})
```

The `AttrMap` widget will apply display attribute `'attr2'` to all parts of the `Text` widget that are using `'attr1'`. The result is the `"hello"` text appearing with display attribute `'attr2'` and all other text and whitespace appearing in the default display attribute.

`AttrMap` can also change display attributes differently when they are in focus. This can be used to "highlight" one or more widgets to make your interface more user friendly. To use this feature set the `focus_map` parameter when creating the `AttrMap` widget.

3.8.2 Foreground and Background Settings

Supported by Terminal	xterm / gnome-term	rxvt	linux console	others
16 standard foreground colors	YES	YES	YES	very widely supported
8 standard background colors	YES	YES	YES	very widely supported
default foreground/background	YES	YES	YES	widely supported
bold, underline, standout	YES	YES	standout	widely supported
"bright" background colors	YES	urxvt		some support
256-color foreground/background	YES			some support
88-color foreground/background	w/palette setting	urxvt		limited support
RGB palette setting	YES			limited support

16 Standard Foreground Colors

- `'black'`
- `'dark red'`
- `'dark green'`
- `'brown'`
- `'dark blue'`
- `'dark magenta'`
- `'dark cyan'`
- `'light gray'`
- `'dark gray'`
- `'light red'`
- `'light green'`
- `'yellow'`
- `'light blue'`
- `'light magenta'`
- `'light cyan'`
- `'white'`

8 Standard Background Colors

- `'black'`
- `'dark red'`
- `'dark green'`
- `'brown'`
- `'dark blue'`
- `'dark magenta'`
- `'dark cyan'`
- `'light gray'`

Default Foreground and Background

- `'default'` (or simply `''`)

`'default'` may be specified as a foreground or background to use a terminal's default color. For terminals with transparent backgrounds `'default'` is the only way to show the transparent background. There is no way to tell what the default colors are, so it is best to use default foregrounds and backgrounds together (not with other colors) to ensure good contrast.

Bold, Underline, Standout

- `'bold'`
- `'underline'`
- `'standout'`

These settings may be tagged on to foreground colors using commas, eg: `'light gray,underline,bold'`

For monochrome mode combinations of these are the only values that may be used.

Many terminals will turn foreground colors into their bright versions when you use bold, eg: `'dark blue,bold'` might look the same as `'light blue'`. Some terminals also will display bright colors in a bold font even if you don't specify bold. To inhibit this you can try setting `bright_is_bold=False` with `BaseScreen.set_terminal_properties()`, but it is not always supported.

`'standout'` is usually displayed as the foreground and background colors reversed.

"Bright" Background Colors

Warning: Terminal support for bright background colors is spotty, and they generally should be avoided. If you are in a high-color mode you might have better luck using the high-color versions `'h8'`, `'h9'`, `'h10'`, ..., `'h15'`.

- `'dark gray'`
- `'light red'`
- `'light green'`
- `'yellow'`
- `'light blue'`
- `'light magenta'`
- `'light cyan'`
- `'white'`

256-Color Foreground and Background Colors

In 256-color mode you have the 16 basic colors, a 6 * 6 * 6 color cube and a gray scale with 24 entries (white and black not included).

The color cube is weighted towards the brighter colors, with RGB points at `0`, `0x5f`, `0x87`, `0xaf`, `0xd7` and `0xff`. The hex characters `'0'`, `'6'`, `'8'`, `'a'`, `'d'` and `'f'` are used as short-forms for these values.

High colors may be specified by their index `'h0'`, ..., `'h255'` or with the shortcuts for the color cube `'#000'`, `'#006'`, `'#008'`, ..., `'#fff'` or gray scale entries `'g0'` (black from color cube) , `'g3'`, `'g7'`, ... `'g100'` (white from color cube).

See also:

The palette_test.py example program

88-Color Foreground and Background Colors

In 88-color mode you have the 16 basic colors, a 4 * 4 * 4 color cube and a gray scale with 8 entries (white and black not included).

The color cube is weighted towards the brighter colors, with RGB points at `0`, `0x8b`, `0xcd`, and `0xff`. The hex characters `'0'`, `'8'`, `'c'` and `'f'` are used as short-forms for these values.

High colors may be specified by their index `'h0'`, ..., `'h87'` or with the shortcuts for the color cube `'#000'`, `'#008'`, `'#00c'`, ..., `'#fff'` or gray scale entries `'g0'` (black from color cube), `'g19'`, `'g35'`, ... `'g100'` (white from color cube).

See also:

The palette_test.py example program

RGB Palette Setting

A few terminals have the ability to customize the terminal palette's RGB values with `raw_display.Screen.modify_terminal_palette()`. There is no automatic way to tell if this is supported by a user's terminal, so this feature shouldn't be relied on.

`raw_display.Screen.reset_default_terminal_palette()` is used to reset the palette in the `palette_test.py` example program when switching modes.

3.9 Canvas Cache

In an Urwid application each time the screen is redrawn typically only part of the screen actually needs to be updated. A canvas cache is used to store visible, unchanged canvases so that not all of the visible widgets need to be rendered for each update.

The `Widget` base class uses some metaclass magic to capture the canvas objects returned `Widget.render()` is called and return them the next time `Widget.render()` is called again with the same parameters. The `Widget._invalidate()` method is provided as a way to remove cached widgets so that changes to the widget are visible the next time the screen is redrawn.

Similar metaclass magic is used for flow widgets' `Widget.rows()` method. If a canvas for that widget with the same parameters is cached then the rows of that canvas are returned instead of calling the widget's actual `Widget.rows()` method.

3.9.1 Composite Canvases

When container and decoration widgets are rendered, they collect the canvases returned by their children and arrange them into a composite canvas. Composite canvases may are nested to form a tree with the topmost widget's `Widget.render()` method returning the root of the tree. That canvas is sent to the display module to be rendered on the screen.

Composite canvases reference the content and layout from their children, reducing the number of copies required to build them. When a canvas is removed from the cache by a call to `Widget._invalidate()` all the direct parents of that canvas are removed from the cache as well, forcing those widgets to be re-drawn on the next screen update. This cascade-removal happens only once per update (the canvas is then no longer in the cache) so batched changes to visible widgets may be made efficiently. This is important when a user's input gets ahead of the screen updating – Urwid handles all the pending input first then updates the screen with the final result, instead of falling further and further behind.

3.9.2 Cache Lifetime

The canvases "stored" in the canvas cache are actually weak references to the canvases. The canvases must have a real reference somewhere for the cache to function properly. Urwid's display modules store the currently displayed topmost canvas for this reason. All canvases that are visible on the screen will remain in the cache, and others will be garbage collected.

3.9.3 Future Work

A updating method that invalidates regions of the display without redrawing parent widgets would be more efficient for the common case of a single change on the screen that does not affect the screen layout. Send an email to the mailing list if you're interested in helping with this or other display optimizations.

CHAPTER 4

Urwid Reference

4.1 MainLoop and Event Loops

4.1.1 MainLoop

class `urwid.`**`MainLoop`**(*widget*, *palette=()*, *screen=None*, *handle_mouse=True*, *input_filter=None*, *unhandled_input=None*, *event_loop=None*, *pop_ups=False*)

This is the standard main loop implementation for a single interactive session.

Parameters

- **widget** (*widget instance*) – the topmost widget used for painting the screen, stored as `widget` and may be modified. Must be a box widget.
- **palette** (*iterable of palette entries*) – initial palette for screen
- **screen** (*display module screen instance*) – screen to use, default is a new `raw_display.Screen` instance; stored as `screen`
- **handle_mouse** (*bool*) – `True` to ask `screen` to process mouse events
- **input_filter** (*callable*) – a function to filter input before sending it to `widget`, called from `input_filter()`
- **unhandled_input** (*callable*) – a function called when input is not handled by `widget`, called from `unhandled_input()`
- **event_loop** (*event loop instance*) – if `screen` supports external an event loop it may be given here, default is a new `SelectEventLoop` instance; stored as `event_loop`
- **pop_ups** (*boolean*) – *True* to wrap `widget` with a `PopUpTarget` instance to allow any widget to open a pop-up anywhere on the screen

`screen`

The screen object this main loop uses for screen updates and reading input

`event_loop`

The event loop object this main loop uses for waiting on alarms and IO

`draw_screen``()`

Render the widgets and paint the screen. This method is called automatically from `entering_idle()`.

If you modify the widgets displayed outside of handling input or responding to an alarm you will need to call this method yourself to repaint the screen.

`entering_idle`()
This method is called whenever the event loop is about to enter the idle state. `draw_screen()` is called here to update the screen when anything has changed.

`input_filter`(*keys*, *raw*)
This function is passed each all the input events and raw keystroke values. These values are passed to the *input_filter* function passed to the constructor. That function must return a list of keys to be passed to the widgets to handle. If no *input_filter* was defined this implementation will return all the input events.

`process_input`(*keys*)
This method will pass keyboard input and mouse events to `widget`. This method is called automatically from the `run()` method when there is input, but may also be called to simulate input from the user.

keys is a list of input returned from `screen`'s get_input() or get_input_nonblocking() methods.

Returns `True` if any key was handled by a widget or the `unhandled_input()` method.

`remove_alarm`(*handle*)
Remove an alarm. Return `True` if *handle* was found, `False` otherwise.

`remove_watch_file`(*handle*)
Remove a watch file. Returns `True` if the watch file exists, `False` otherwise.

`remove_watch_pipe`(*write_fd*)
Close the read end of the pipe and remove the watch created by `watch_pipe()`. You are responsible for closing the write end of the pipe.

Returns `True` if the watch pipe exists, `False` otherwise

`run`()
Start the main loop handling input events and updating the screen. The loop will continue until an `ExitMainLoop` exception is raised.

This method will use `screen`'s run_wrapper() method if `screen`'s start() method has not already been called.

`set_alarm_at`(*tm*, *callback*, *user_data=None*)
Schedule an alarm at *tm* time that will call *callback* from the within the :meth'run' function. Returns a handle that may be passed to `remove_alarm()`.

Parameters

- **tm** (*float*) – time to call callback e.g. `time.time() + 5`
- **callback** (*callable*) – function to call with two parameters: this main loop object and *user_data*

`set_alarm_in`(*sec*, *callback*, *user_data=None*)
Schedule an alarm in *sec* seconds that will call *callback* from the within the `run()` method.

Parameters

- **sec** (*float*) – seconds until alarm
- **callback** (*callable*) – function to call with two parameters: this main loop object and *user_data*

`unhandled_input`(*input*)
This function is called with any input that was not handled by the widgets, and calls the *unhandled_input* function passed to the constructor. If no *unhandled_input* was defined then the input will be ignored.

input is the keyboard or mouse input.

The *unhandled_input* function should return `True` if it handled the input.

`watch_file`(*fd*, *callback*)
Call *callback* when *fd* has some data to read. No parameters are passed to callback.

Returns a handle that may be passed to `remove_watch_file()`.

`watch_pipe`(*callback*)
Create a pipe for use by a subprocess or thread to trigger a callback in the process/thread running the main loop.

Parameters callback (*callable*) – function taking one parameter to call from within the process/thread running the main loop

This method returns a file descriptor attached to the write end of a pipe. The read end of the pipe is added to the list of files `event_loop` is watching. When data is written to the pipe the callback function will be called and passed a single value containing data read from the pipe.

This method may be used any time you want to update widgets from another thread or subprocess.

Data may be written to the returned file descriptor with `os.write(fd, data)`. Ensure that data is less than 512 bytes (or 4K on Linux) so that the callback will be triggered just once with the complete value of data passed in.

If the callback returns `False` then the watch will be removed from `event_loop` and the read end of the pipe will be closed. You are responsible for closing the write end of the pipe with `os.close(fd)`.

`widget`
Property for the topmost widget used to draw the screen. This must be a box widget.

4.1.2 SelectEventLoop

class `urwid.`**`SelectEventLoop`**
Event loop based on `select.select()`

`alarm`(*seconds*, *callback*)
Call callback() given time from from now. No parameters are passed to callback.

Returns a handle that may be passed to remove_alarm()

seconds – floating point time to wait before calling callback callback – function to call from event loop

`enter_idle`(*callback*)
Add a callback for entering idle.

Returns a handle that may be passed to remove_idle()

`remove_alarm`(*handle*)
Remove an alarm.

Returns True if the alarm exists, False otherwise

`remove_enter_idle`(*handle*)
Remove an idle callback.

Returns True if the handle was removed.

`remove_watch_file`(*handle*)
Remove an input file.

Returns True if the input file exists, False otherwise

`run`()
Start the event loop. Exit the loop when any callback raises an exception. If ExitMainLoop is raised, exit cleanly.

watch_file (*fd*, *callback*)

Call callback() when fd has some data to read. No parameters are passed to callback.

Returns a handle that may be passed to remove_watch_file()

fd – file descriptor to watch for input callback – function to call when input is available

4.1.3 GLibEventLoop

class `urwid.`**`GLibEventLoop`**

Event loop based on gobject.MainLoop

alarm (*seconds*, *callback*)

Call callback() given time from from now. No parameters are passed to callback.

Returns a handle that may be passed to remove_alarm()

seconds – floating point time to wait before calling callback callback – function to call from event loop

enter_idle (*callback*)

Add a callback for entering idle.

Returns a handle that may be passed to remove_enter_idle()

handle_exit (*f*)

Decorator that cleanly exits the `GLibEventLoop` if `ExitMainLoop` is thrown inside of the wrapped function. Store the exception info if some other exception occurs, it will be reraised after the loop quits.

f – function to be wrapped

remove_alarm (*handle*)

Remove an alarm.

Returns True if the alarm exists, False otherwise

remove_enter_idle (*handle*)

Remove an idle callback.

Returns True if the handle was removed.

remove_watch_file (*handle*)

Remove an input file.

Returns True if the input file exists, False otherwise

run ()

Start the event loop. Exit the loop when any callback raises an exception. If ExitMainLoop is raised, exit cleanly.

watch_file (*fd*, *callback*)

Call callback() when fd has some data to read. No parameters are passed to callback.

Returns a handle that may be passed to remove_watch_file()

fd – file descriptor to watch for input callback – function to call when input is available

4.1.4 TwistedEventLoop

class `urwid.`**`TwistedEventLoop`** (*reactor=None*, *manage_reactor=True*)

Event loop based on Twisted

Parameters **reactor** (`twisted.internet.reactor.`) – reactor to use

Param manage_reactor: *True* if you want this event loop to run and stop the reactor.

Warning: Twisted's reactor doesn't like to be stopped and run again. If you need to stop and run your `MainLoop`, consider setting `manage_reactor=False` and take care of running/stopping the reactor at the beginning/ending of your program yourself.

`alarm`(*seconds*, *callback*)
Call callback() given time from from now. No parameters are passed to callback.

Returns a handle that may be passed to remove_alarm()

seconds – floating point time to wait before calling callback callback – function to call from event loop

`enter_idle`(*callback*)
Add a callback for entering idle.

Returns a handle that may be passed to remove_enter_idle()

`handle_exit`(*f*, *enable_idle=True*)
Decorator that cleanly exits the `TwistedEventLoop` if `ExitMainLoop` is thrown inside of the wrapped function. Store the exception info if some other exception occurs, it will be reraised after the loop quits.

f – function to be wrapped

`remove_alarm`(*handle*)
Remove an alarm.

Returns True if the alarm exists, False otherwise

`remove_enter_idle`(*handle*)
Remove an idle callback.

Returns True if the handle was removed.

`remove_watch_file`(*handle*)
Remove an input file.

Returns True if the input file exists, False otherwise

`run`()
Start the event loop. Exit the loop when any callback raises an exception. If ExitMainLoop is raised, exit cleanly.

`watch_file`(*fd*, *callback*)
Call callback() when fd has some data to read. No parameters are passed to callback.

Returns a handle that may be passed to remove_watch_file()

fd – file descriptor to watch for input callback – function to call when input is available

4.2 Widget Classes

4.2.1 Widget Base Classes

Widget

class `urwid.`**`Widget`**
Widget base class

`__metaclass__` = urwid.WidgetMeta
See `urwid.WidgetMeta` definition

`_selectable` = False
The default `selectable()` method returns this value.

`_sizing` = frozenset(['flow', 'box', 'fixed'])
The default `sizing()` method returns this value.

`_command_map` = urwid.command_map
A shared `CommandMap` instance. May be redefined in subclasses or widget instances.

`render`(*size*, *focus=False*)

Note: This method is not implemented in `Widget` but must be implemented by any concrete subclass

Parameters

- **size** (*widget size*) – One of the following, *maxcol* and *maxrow* are integers > 0:

  (*maxcol*, *maxrow*) for box sizing – the parent chooses the exact size of this widget

  (*maxcol*,) for flow sizing – the parent chooses only the number of columns for this widget

  () for fixed sizing – this widget is a fixed size which can't be adjusted by the parent

- **focus** (*bool*) – set to `True` if this widget or one of its children is in focus

Returns A `Canvas` subclass instance containing the rendered content of this widget

`Text` widgets return a `TextCanvas` (arbitrary text and display attributes), `SolidFill` widgets return a `SolidCanvas` (a single character repeated across the whole surface) and container widgets return a `CompositeCanvas` (one or more other canvases arranged arbitrarily).

If *focus* is `False`, the returned canvas may not have a cursor position set.

There is some metaclass magic defined in the `Widget` metaclass `WidgetMeta` that causes the result of this method to be cached by `CanvasCache`. Later calls will automatically look up the value in the cache first.

As a small optimization the class variable `ignore_focus` may be defined and set to `True` if this widget renders the same canvas regardless of the value of the *focus* parameter.

Any time the content of a widget changes it should call `_invalidate()` to remove any cached canvases, or the widget may render the cached canvas instead of creating a new one.

`rows`(*size*, *focus=False*)

Note: This method is not implemented in `Widget` but must be implemented by any flow widget. See `sizing()`.

See `Widget.render()` for parameter details.

Returns The number of rows required for this widget given a number of columns in *size*

This is the method flow widgets use to communicate their size to other widgets without having to render a canvas. This should be a quick calculation as this function may be called a number of times in normal operation. If your implementation may take a long time you should add your own caching here.

There is some metaclass magic defined in the `Widget` metaclass `WidgetMeta` that causes the result of this function to be retrieved from any canvas cached by `CanvasCache`, so if your widget has been rendered you may not receive calls to this function. The class variable `ignore_focus` may be defined and set to `True` if this widget renders the same size regardless of the value of the *focus* parameter.

`keypress`(*size*, *key*)

Note: This method is not implemented in `Widget` but must be implemented by any selectable widget. See `selectable()`.

Parameters

- **size** (*widget size*) – See `Widget.render()` for details
- **key** (*bytes or unicode*) – a single keystroke value; see *Keyboard Input*

Returns `None` if *key* was handled by this widget or *key* (the same value passed) if *key* was not handled by this widget

Container widgets will typically call the `keypress()` method on whichever of their children is set as the focus.

The standard widgets use `_command_map` to determine what action should be performed for a given *key*. You may modify these values to your liking globally, at some level in the widget hierarchy or on individual widgets. See `CommandMap` for the defaults.

In your own widgets you may use whatever logic you like: filtering or translating keys, selectively passing along events etc.

`mouse_event`(*size*, *event*, *button*, *col*, *row*, *focus*)

Note: This method is not implemented in `Widget` but may be implemented by a subclass. Not implementing this method is equivalent to having a method that always returns `False`.

Parameters

- **size** (*widget size*) – See `Widget.render()` for details.
- **event** (*mouse event*) – Values such as `'mouse press'`, `'ctrl mouse press'`, `'mouse release'`, `'meta mouse release'`, `'mouse drag'`; see *Mouse Input*
- **button** (*int*) – 1 through 5 for press events, often 0 for release events (which button was released is often not known)
- **col** (*int*) – Column of the event, 0 is the left edge of this widget
- **row** (*int*) – Row of the event, 0 it the top row of this widget
- **focus** (*bool*) – Set to `True` if this widget or one of its children is in focus

Returns `True` if the event was handled by this widget, `False` otherwise

Container widgets will typically call the `mouse_event()` method on whichever of their children is at the position (*col*, *row*).

`get_cursor_coords`(*size*)

Note: This method is not implemented in `Widget` but must be implemented by any widget that may return cursor coordinates as part of the canvas that `render()` returns.

Parameters **size** (*widget size*) – See `Widget.render()` for details.

Returns (*col*, *row*) if this widget has a cursor, `None` otherwise

Return the cursor coordinates (*col*, *row*) of a cursor that will appear as part of the canvas rendered by this widget when in focus, or `None` if no cursor is displayed.

The `ListBox` widget uses this method to make sure a cursor in the focus widget is not scrolled out of view. It is a separate method to avoid having to render the whole widget while calculating layout.

Container widgets will typically call the `get_cursor_coords()` method on their focus widget.

`get_pref_col`(*size*)

Note: This method is not implemented in `Widget` but may be implemented by a subclass.

Parameters **size** (*widget size*) – See `Widget.render()` for details.

Returns a column number or `'left'` for the leftmost available column or `'right'` for the rightmost available column

Return the preferred column for the cursor to be displayed in this widget. This value might not be the same as the column returned from `get_cursor_coords()`.

The `ListBox` and `Pile` widgets call this method on a widget losing focus and use the value returned to call `move_cursor_to_coords()` on the widget becoming the focus. This allows the focus to move up and down through widgets while keeping the cursor in approximately the same column on screen.

`move_cursor_to_coords`(*size*, *col*, *row*)

Note: This method is not implemented in `Widget` but may be implemented by a subclass. Not implementing this method is equivalent to having a method that always returns `False`.

Parameters

- **size** (*widget size*) – See `Widget.render()` for details.
- **col** (*int*) – new column for the cursor, 0 is the left edge of this widget
- **row** (*int*) – new row for the cursor, 0 it the top row of this widget

Returns `True` if the position was set successfully anywhere on *row*, `False` otherwise

x.__init__(...) initializes x; see help(type(x)) for signature

`_emit`(*name*, **args*)

Convenience function to emit signals with self as first argument.

`_invalidate`()

Mark cached canvases rendered by this widget as dirty so that they will not be used again.

base_widget
Read-only property that steps through decoration widgets and returns the one at the base. This default implementation returns self.

focus
Read-only property returning the child widget in focus for container widgets. This default implementation always returns `None`, indicating that this widget has no children.

focus_position
Property for reading and setting the focus position for container widgets. This default implementation raises `IndexError`, making normal widgets fail the same way accessing `focus_position` on an empty container widget would.

pack (*size*, *focus=False*)
See `Widget.render()` for parameter details.

Returns A "packed" size (*maxcol*, *maxrow*) for this widget

Calculate and return a minimum size where all content could still be displayed. Fixed widgets must implement this method and return their size when `()` is passed as the *size* parameter.

This default implementation returns the *size* passed, or the *maxcol* passed and the value of `rows()` as the *maxrow* when (*maxcol*,) is passed as the *size* parameter.

Note: This is a new method that hasn't been fully implemented across the standard widget types. In particular it has not yet been implemented for container widgets.

`Text` widgets have implemented this method. You can use `Text.pack()` to calculate the minumum columns and rows required to display a text widget without wrapping, or call it iteratively to calculate the minimum number of columns required to display the text wrapped into a target number of rows.

selectable()

Returns `True` if this is a widget that is designed to take the focus, i.e. it contains something the user might want to interact with, `False` otherwise,

This default implementation returns `_selectable`. Subclasses may leave these is if the are not selectable, or if they are always selectable they may set the `_selectable` class variable to `True`.

If this method returns `True` then the `keypress()` method must be implemented.

Returning `False` does not guarantee that this widget will never be in focus, only that this widget will usually be skipped over when changing focus. It is still possible for non selectable widgets to have the focus (typically when there are no other selectable widgets visible).

sizing()

Returns A frozenset including one or more of `'box'`, `'flow'` and `'fixed'`. Default implementation returns the value of `_sizing`, which for this class includes all three.

The sizing modes returned indicate the modes that may be supported by this widget, but is not sufficient to know that using that sizing mode will work. Subclasses should make an effort to remove sizing modes they know will not work given the state of the widget, but many do not yet do this.

If a sizing mode is missing from the set then the widget should fail when used in that mode.

If `'flow'` is among the values returned then the other methods in this widget must be able to accept a single-element tuple (*maxcol*,) to their `size` parameter, and the `rows()` method must be defined.

If `'box'` is among the values returned then the other methods must be able to accept a two-element tuple (*maxcol*, *maxrow*) to their size paramter.

If `'fixed'` is among the values returned then the other methods must be able to accept an empty tuple () to their size parameter, and the `pack()` method must be defined.

WidgetWrap

class `urwid.`**`WidgetWrap`**(*w*)

w – widget to wrap, stored as self._w

This object will pass the functions defined in Widget interface definition to self._w.

The purpose of this widget is to provide a base class for widgets that compose other widgets for their display and behaviour. The details of that composition should not affect users of the subclass. The subclass may decide to expose some of the wrapped widgets by behaving like a ContainerWidget or WidgetDecoration, or it may hide them from outside access.

WidgetDecoration

class `urwid.`**`WidgetDecoration`**(*original_widget*)

original_widget – the widget being decorated

This is a base class for decoration widgets, widgets that contain one or more widgets and only ever have a single focus. This type of widget will affect the display or behaviour of the original_widget but it is not part of determining a chain of focus.

Don't actually do this – use a WidgetDecoration subclass instead, these are not real widgets:

```
>>> WidgetDecoration(Text(u"hi"))
<WidgetDecoration flow widget <Text flow widget 'hi'>>
```

`base_widget`

Return the widget without decorations. If there is only one Decoration then this is the same as original_widget.

```
>>> t = Text('hello')
>>> wd1 = WidgetDecoration(t)
>>> wd2 = WidgetDecoration(wd1)
>>> wd3 = WidgetDecoration(wd2)
>>> wd3.original_widget is wd2
True
>>> wd3.base_widget is t
True
```

WidgetContainerMixin

class `urwid.`**`WidgetContainerMixin`**

Mixin class for widget containers implementing common container methods

x.__init__(...) initializes x; see help(type(x)) for signature

`get_focus_path`()

Return the .focus_position values starting from this container and proceeding along each child widget until reaching a leaf (non-container) widget.

`set_focus_path`(*positions*)

Set the .focus_position property starting from this container widget and proceeding along newly focused child widgets. Any failed assignment due do incompatible position types or invalid positions will raise an IndexError.

This method may be used to restore a particular widget to the focus by passing in the value returned from an earlier call to get_focus_path().

positions – sequence of positions

4.2.2 Basic Widget Classes

Text

class `urwid.`**`Text`**`(`*markup*, *align='left'*, *wrap='space'*, *layout=None*`)`
a horizontally resizeable text widget

Parameters

- **markup** (*Text Markup*) – content of text widget, one of:

  bytes or unicode text to be displayed

  (*display attribute*, *text markup*) *text markup* with *display attribute* applied to all parts of *text markup* with no display attribute already applied

  [*text markup*, *text markup*, ...] all *text markup* in the list joined together

- **align** (*text alignment mode*) – typically `'left'`, `'center'` or `'right'`
- **wrap** (*text wrapping mode*) – typically `'space'`, `'any'` or `'clip'`
- **layout** (*text layout instance*) – defaults to a shared `StandardTextLayout` instance

```
>>> Text(u"Hello")
<Text flow widget 'Hello'>
>>> t = Text(('bold', u"stuff"), 'right', 'any')
>>> t
<Text flow widget 'stuff' align='right' wrap='any'>
>>> print t.text
stuff
>>> t.attrib
[('bold', 5)]
```

`attrib`
Read-only property returning the run-length encoded display attributes of this widget

`get_line_translation``(`*maxcol*, *ta=None*`)`
Return layout structure used to map self.text to a canvas. This method is used internally, but may be useful for debugging custom layout classes.

Parameters

- **maxcol** (*int*) – columns available for display
- **ta** (*text and display attributes*) – `None` or the (*text*, *display attributes*) tuple returned from `get_text()`

`get_text``()`

Returns

(*text*, *display attributes*)

text complete bytes/unicode content of text widget

display attributes run length encoded display attributes for *text*, eg. `[('attr1', 10), ('attr2', 5)]`

```
>>> Text(u"Hello").get_text() # ... = u in Python 2
(...'Hello', [])
>>> Text(('bright', u"Headline")).get_text()
(...'Headline', [('bright', 8)])
>>> Text([('a', u"one"), u"two", ('b', u"three")]).get_text()
(...'onetwothree', [('a', 3), (None, 3), ('b', 5)])
```

pack (*size=None, focus=False*)
Return the number of screen columns and rows required for this Text widget to be displayed without wrapping or clipping, as a single element tuple.

Parameters **size** (*widget size*) – `None` for unlimited screen columns or (*maxcol*,) to specify a maximum column size

```
>>> Text(u"important things").pack()
(16, 1)
>>> Text(u"important things").pack((15,))
(9, 2)
>>> Text(u"important things").pack((8,))
(8, 2)
```

render (*size, focus=False*)
Render contents with wrapping and alignment. Return canvas.

See `Widget.render()` for parameter details.

```
>>> Text(u"important things").render((18,)).text # ... = b in Python 3
[...'important things  ']
>>> Text(u"important things").render((11,)).text
[...'important  ', ...'things     ']
```

rows (*size, focus=False*)
Return the number of rows the rendered text requires.

See `Widget.rows()` for parameter details.

```
>>> Text(u"important things").rows((18,))
1
>>> Text(u"important things").rows((11,))
2
```

set_align_mode (*mode*)
Set text alignment mode. Supported modes depend on text layout object in use but defaults to a `StandardTextLayout` instance

Parameters **mode** (*text alignment mode*) – typically `'left'`, `'center'` or `'right'`

```
>>> t = Text(u"word")
>>> t.set_align_mode('right')
>>> t.align
'right'
>>> t.render((10,)).text # ... = b in Python 3
[...'      word']
>>> t.align = 'center'
>>> t.render((10,)).text
[...'   word   ']
>>> t.align = 'somewhere'
Traceback (most recent call last):
TextError: Alignment mode 'somewhere' not supported.
```

set_layout (*align*, *wrap*, *layout=None*)
Set the text layout object, alignment and wrapping modes at the same time.

Parameters

- **wrap** (*text wrapping mode*) – typically 'space', 'any' or 'clip'
- **layout** (*text layout instance*) – defaults to a shared `StandardTextLayout` instance

```
>>> t = Text(u"hi")
>>> t.set_layout('right', 'clip')
>>> t
<Text flow widget 'hi' align='right' wrap='clip'>
```

set_text (*markup*)
Set content of text widget.

Parameters **markup** (*text markup*) – see `Text` for description.

```
>>> t = Text(u"foo")
>>> print t.text
foo
>>> t.set_text(u"bar")
>>> print t.text
bar
>>> t.text = u"baz"  # not supported because text stores text but set_text() takes markup
Traceback (most recent call last):
AttributeError: can't set attribute
```

set_wrap_mode (*mode*)
Set text wrapping mode. Supported modes depend on text layout object in use but defaults to a `StandardTextLayout` instance

Parameters **mode** (*text wrapping mode*) – typically `'space'`, `'any'` or `'clip'`

```
>>> t = Text(u"some words")
>>> t.render((6,)).text # ... = b in Python 3
[...'some  ', ...'words ']
>>> t.set_wrap_mode('clip')
>>> t.wrap
'clip'
>>> t.render((6,)).text
[...'some w']
>>> t.wrap = 'any'  # Urwid 0.9.9 or later
>>> t.render((6,)).text
[...'some w', ...'ords  ']
>>> t.wrap = 'somehow'
Traceback (most recent call last):
TextError: Wrap mode 'somehow' not supported.
```

text
Read-only property returning the complete bytes/unicode content of this widget

Edit

class urwid.**Edit** (*caption=u''*, *edit_text=u''*, *multiline=False*, *align='left'*, *wrap='space'*, *allow_tab=False*, *edit_pos=None*, *layout=None*, *mask=None*)
Text editing widget implements cursor movement, text insertion and deletion. A caption may prefix the editing area. Uses text class for text layout.

Users of this class to listen for `"change"` events sent when the value of edit_text changes. See :func:`connect_signal`.

Parameters

- **caption** (*text markup*) – markup for caption preceeding edit_text, see `Text` for description of text markup.
- **edit_text** (*bytes or unicode*) – initial text for editing, type (bytes or unicode) must match the text in the caption
- **multiline** (*bool*) – True: 'enter' inserts newline False: return it
- **align** (*text alignment mode*) – typically 'left', 'center' or 'right'
- **wrap** (*text wrapping mode*) – typically 'space', 'any' or 'clip'
- **allow_tab** (*bool*) – True: 'tab' inserts 1-8 spaces False: return it
- **edit_pos** (*int*) – initial position for cursor, None:end of edit_text
- **layout** (*text layout instance*) – defaults to a shared `StandardTextLayout` instance
- **mask** (*bytes or unicode*) – hide text entered with this character, None:disable mask

```
>>> Edit()
<Edit selectable flow widget '' edit_pos=0>
>>> Edit(u"Y/n? ", u"yes")
<Edit selectable flow widget 'yes' caption='Y/n? ' edit_pos=3>
>>> Edit(u"Name ", u"Smith", edit_pos=1)
<Edit selectable flow widget 'Smith' caption='Name ' edit_pos=1>
>>> Edit(u"", u"3.14", align='right')
<Edit selectable flow widget '3.14' align='right' edit_pos=4>
```

`edit_text`

Read-only property returning the edit text for this widget.

`get_cursor_coords`(*size*)

Return the (*x*, *y*) coordinates of cursor within widget.

```
>>> Edit("? ","yes").get_cursor_coords((10,))
(5, 0)
```

`get_edit_text`()

Return the edit text for this widget.

```
>>> e = Edit(u"What? ", u"oh, nothing.")
>>> print e.get_edit_text()
oh, nothing.
>>> print e.edit_text
oh, nothing.
```

`get_pref_col`(*size*)

Return the preferred column for the cursor, or the current cursor x value. May also return `'left'` or `'right'` to indicate the leftmost or rightmost column available.

This method is used internally and by other widgets when moving the cursor up or down between widgets so that the column selected is one that the user would expect.

```
>>> size = (10,)
>>> Edit().get_pref_col(size)
0
>>> e = Edit(u"", u"word")
>>> e.get_pref_col(size)
4
>>> e.keypress(size, 'left')
>>> e.get_pref_col(size)
3
>>> e.keypress(size, 'end')
>>> e.get_pref_col(size)
'right'
>>> e = Edit(u"", u"2\nwords")
>>> e.keypress(size, 'left')
>>> e.keypress(size, 'up')
>>> e.get_pref_col(size)
4
>>> e.keypress(size, 'left')
>>> e.get_pref_col(size)
0
```

`get_text`()

Returns `(text, display attributes)`. See `Text.get_text()` for details.

Text returned includes the caption and edit_text, possibly masked.

```
>>> Edit(u"What? ","oh, nothing.").get_text() # ... = u in Python 2
(...'What? oh, nothing.', [])
>>> Edit(('bright',u"user@host:~$ "),"ls").get_text()
(...'user@host:~$ ls', [('bright', 13)])
>>> Edit(u"password:", u"seekrit", mask=u"*").get_text()
(...'password:*******', [])
```

`insert_text`(*text*)

Insert text at the cursor position and update cursor. This method is used by the keypress() method when inserting one or more characters into edit_text.

Parameters **text** (*bytes or unicode*) – text for inserting, type (bytes or unicode) must match the text in the caption

```
>>> e = Edit(u"", u"42")
>>> e.insert_text(u".5")
>>> e
<Edit selectable flow widget '42.5' edit_pos=4>
>>> e.set_edit_pos(2)
>>> e.insert_text(u"a")
>>> print e.edit_text
42a.5
```

`insert_text_result`(*text*)

Return result of insert_text(text) without actually performing the insertion. Handy for pre-validation.

Parameters **text** (*bytes or unicode*) – text for inserting, type (bytes or unicode) must match the text in the caption

`keypress`(*size*, *key*)

Handle editing keystrokes, return others.

```
>>> e, size = Edit(), (20,)
>>> e.keypress(size, 'x')
>>> e.keypress(size, 'left')
>>> e.keypress(size, '1')
>>> print e.edit_text
1x
```

```
>>> e.keypress(size, 'backspace')
>>> e.keypress(size, 'end')
>>> e.keypress(size, '2')
>>> print e.edit_text
x2
>>> e.keypress(size, 'shift f1')
'shift f1'
```

mouse_event (*size*, *event*, *button*, *x*, *y*, *focus*)
Move the cursor to the location clicked for button 1.

```
>>> size = (20,)
>>> e = Edit("","words here")
>>> e.mouse_event(size, 'mouse press', 1, 2, 0, True)
True
>>> e.edit_pos
2
```

move_cursor_to_coords (*size*, *x*, *y*)
Set the cursor position with (x,y) coordinates. Returns True if move succeeded, False otherwise.

```
>>> size = (10,)
>>> e = Edit("","edit\ntext")
>>> e.move_cursor_to_coords(size, 5, 0)
True
>>> e.edit_pos
4
>>> e.move_cursor_to_coords(size, 5, 3)
False
>>> e.move_cursor_to_coords(size, 0, 1)
True
>>> e.edit_pos
5
```

position_coords (*maxcol*, *pos*)
Return (*x*, *y*) coordinates for an offset into self.edit_text.

render (*size*, *focus=False*)
Render edit widget and return canvas. Include cursor when in focus.

```
>>> c = Edit("? ","yes").render((10,), focus=True)
>>> c.text # ... = b in Python 3
[...'? yes     ']
>>> c.cursor
(5, 0)
```

set_caption (*caption*)
Set the caption markup for this widget.

Parameters caption – markup for caption preceeding edit_text, see `Text.__init__()` for description of text markup.

```
>>> e = Edit("")
>>> e.set_caption("cap1")
>>> print e.caption
cap1
>>> e.set_caption(('bold', "cap2"))
>>> print e.caption
cap2
>>> e.attrib
```

```
[('bold', 4)]
>>> e.caption = "cap3"  # not supported because caption stores text but set_caption() takes
Traceback (most recent call last):
AttributeError: can't set attribute
```

set_edit_pos (*pos*)
Set the cursor position with a self.edit_text offset. Clips pos to [0, len(edit_text)].

Parameters **pos** (*int*) – cursor position

```
>>> e = Edit(u"", u"word")
>>> e.edit_pos
4
>>> e.set_edit_pos(2)
>>> e.edit_pos
2
>>> e.edit_pos = -1  # Urwid 0.9.9 or later
>>> e.edit_pos
0
>>> e.edit_pos = 20
>>> e.edit_pos
4
```

set_edit_text (*text*)
Set the edit text for this widget.

Parameters **text** (*bytes or unicode*) – text for editing, type (bytes or unicode) must match the text in the caption

```
>>> e = Edit()
>>> e.set_edit_text(u"yes")
>>> print e.edit_text
yes
>>> e
<Edit selectable flow widget 'yes' edit_pos=0>
>>> e.edit_text = u"no"  # Urwid 0.9.9 or later
>>> print e.edit_text
no
```

set_mask (*mask*)
Set the character for masking text away.

Parameters **mask** (*bytes or unicode*) – hide text entered with this character, None:disable mask

set_text (*markup*)
Not supported by Edit widget.

```
>>> Edit().set_text("test")
Traceback (most recent call last):
EditError: set_text() not supported.  Use set_caption() or set_edit_text() instead.
```

update_text ()
No longer supported.

```
>>> Edit().update_text()
Traceback (most recent call last):
EditError: update_text() has been removed.  Use set_caption() or set_edit_text() instead.
```

valid_char (*ch*)
Filter for text that may be entered into this widget by the user

Parameters **ch** (*bytes or unicode*) – character to be inserted

This implementation returns True for all printable characters.

IntEdit

class `urwid.`**`IntEdit`**(*caption=''*, *default=None*)

Edit widget for integer values

caption – caption markup default – default edit value

```
>>> IntEdit(u"", 42)
<IntEdit selectable flow widget '42' edit_pos=2>
```

`keypress`(*size*, *key*)

Handle editing keystrokes. Remove leading zeros.

```
>>> e, size = IntEdit(u"", 5002), (10,)
>>> e.keypress(size, 'home')
>>> e.keypress(size, 'delete')
>>> print e.edit_text
002
>>> e.keypress(size, 'end')
>>> print e.edit_text
2
```

`valid_char`(*ch*)

Return true for decimal digits.

`value`()

Return the numeric value of self.edit_text.

```
>>> e, size = IntEdit(), (10,)
>>> e.keypress(size, '5')
>>> e.keypress(size, '1')
>>> e.value() == 51
True
```

Button

class `urwid.`**`Button`**(*label*, *on_press=None*, *user_data=None*)

Parameters

- **label** – markup for button label
- **on_press** – shorthand for connect_signal() function call for a single callback
- **user_data** – user_data for on_press

Signals supported: `'click'`

Register signal handler with:

```
urwid.connect_signal(button, 'click', callback, user_data)
```

where callback is callback(button [,user_data]) Unregister signal handlers with:

```
urwid.disconnect_signal(button, 'click', callback, user_data)
```

```
>>> Button(u"Ok")
<Button selectable flow widget 'Ok'>
>>> b = Button("Cancel")
>>> b.render((15,), focus=True).text # ... = b in Python 3
[...'< Cancel      >']
```

get_label()

Return label text.

```
>>> b = Button(u"Ok")
>>> print b.get_label()
Ok
>>> print b.label
Ok
```

keypress(*size*, *key*)

Send 'click' signal on 'activate' command.

```
>>> assert Button._command_map[' '] == 'activate'
>>> assert Button._command_map['enter'] == 'activate'
>>> size = (15,)
>>> b = Button(u"Cancel")
>>> clicked_buttons = []
>>> def handle_click(button):
...     clicked_buttons.append(button.label)
>>> connect_signal(b, 'click', handle_click)
>>> b.keypress(size, 'enter')
>>> b.keypress(size, ' ')
>>> clicked_buttons # ... = u in Python 2
[...'Cancel', ...'Cancel']
```

label

Return label text.

```
>>> b = Button(u"Ok")
>>> print b.get_label()
Ok
>>> print b.label
Ok
```

mouse_event(*size*, *event*, *button*, *x*, *y*, *focus*)

Send 'click' signal on button 1 press.

```
>>> size = (15,)
>>> b = Button(u"Ok")
>>> clicked_buttons = []
>>> def handle_click(button):
...     clicked_buttons.append(button.label)
>>> connect_signal(b, 'click', handle_click)
>>> b.mouse_event(size, 'mouse press', 1, 4, 0, True)
True
>>> b.mouse_event(size, 'mouse press', 2, 4, 0, True) # ignored
False
>>> clicked_buttons # ... = u in Python 2
[...'Ok']
```

set_label(*label*)

Change the button label.

label – markup for button label

```
>>> b = Button("Ok")
>>> b.set_label(u"Yup yup")
>>> b
<Button selectable flow widget 'Yup yup'>
```

CheckBox

class urwid.**CheckBox**(*label*, *state=False*, *has_mixed=False*, *on_state_change=None*, *user_data=None*)

Parameters

- **label** – markup for check box label
- **state** – False, True or "mixed"
- **has_mixed** – True if "mixed" is a state to cycle through
- **on_state_change** – shorthand for connect_signal() function call for a single callback
- **user_data** – user_data for on_state_change

Signals supported: `'change'`

Register signal handler with:

```
urwid.connect_signal(check_box, 'change', callback, user_data)
```

where callback is callback(check_box, new_state [,user_data]) Unregister signal handlers with:

```
urwid.disconnect_signal(check_box, 'change', callback, user_data)
```

```
>>> CheckBox(u"Confirm")
<CheckBox selectable flow widget 'Confirm' state=False>
>>> CheckBox(u"Yogourt", "mixed", True)
<CheckBox selectable flow widget 'Yogourt' state='mixed'>
>>> cb = CheckBox(u"Extra onions", True)
>>> cb
<CheckBox selectable flow widget 'Extra onions' state=True>
>>> cb.render((20,), focus=True).text # ... = b in Python 3
[...'[X] Extra onions    ']
```

get_label()

Return label text.

```
>>> cb = CheckBox(u"Seriously")
>>> print cb.get_label()
Seriously
>>> print cb.label
Seriously
>>> cb.set_label([('bright_attr', u"flashy"), u" normal"])
>>> print cb.label  #  only text is returned
flashy normal
```

get_state()

Return the state of the checkbox.

keypress(*size*, *key*)

Toggle state on 'activate' command.

```
>>> assert CheckBox._command_map[' '] == 'activate'
>>> assert CheckBox._command_map['enter'] == 'activate'
>>> size = (10,)
>>> cb = CheckBox('press me')
>>> cb.state
False
>>> cb.keypress(size, ' ')
>>> cb.state
True
>>> cb.keypress(size, ' ')
>>> cb.state
False
```

label

Return label text.

```
>>> cb = CheckBox(u"Seriously")
>>> print cb.get_label()
Seriously
>>> print cb.label
Seriously
>>> cb.set_label([('bright_attr', u"flashy"), u" normal"])
>>> print cb.label  #  only text is returned
flashy normal
```

mouse_event (*size*, *event*, *button*, *x*, *y*, *focus*)

Toggle state on button 1 press.

```
>>> size = (20,)
>>> cb = CheckBox("clickme")
>>> cb.state
False
>>> cb.mouse_event(size, 'mouse press', 1, 2, 0, True)
True
>>> cb.state
True
```

set_label (*label*)

Change the check box label.

label – markup for label. See Text widget for description of text markup.

```
>>> cb = CheckBox(u"foo")
>>> cb
<CheckBox selectable flow widget 'foo' state=False>
>>> cb.set_label(('bright_attr', u"bar"))
>>> cb
<CheckBox selectable flow widget 'bar' state=False>
```

set_state (*state*, *do_callback=True*)

Set the CheckBox state.

state – True, False or “mixed” do_callback – False to supress signal from this change

```
>>> changes = []
>>> def callback_a(cb, state, user_data):
...     changes.append("A %r %r" % (state, user_data))
>>> def callback_b(cb, state):
...     changes.append("B %r" % state)
>>> cb = CheckBox('test', False, False)
>>> connect_signal(cb, 'change', callback_a, "user_a")
>>> connect_signal(cb, 'change', callback_b)
>>> cb.set_state(True) # both callbacks will be triggered
>>> cb.state
True
>>> disconnect_signal(cb, 'change', callback_a, "user_a")
>>> cb.state = False
>>> cb.state
False
>>> cb.set_state(True)
>>> cb.state
True
>>> cb.set_state(False, False) # don't send signal
>>> changes
["A True 'user_a'", 'B True', 'B False', 'B True']
```

state
Return the state of the checkbox.

toggle_state()
Cycle to the next valid state.

```
>>> cb = CheckBox("3-state", has_mixed=True)
>>> cb.state
False
>>> cb.toggle_state()
>>> cb.state
True
>>> cb.toggle_state()
>>> cb.state
'mixed'
>>> cb.toggle_state()
>>> cb.state
False
```

RadioButton

class urwid.**RadioButton**(*group*, *label*, *state='first True'*, *on_state_change=None*, *user_data=None*)

Parameters

- **group** – list for radio buttons in same group
- **label** – markup for radio button label
- **state** – False, True, “mixed” or “first True”
- **on_state_change** – shorthand for connect_signal() function call for a single ‘change’ callback
- **user_data** – user_data for on_state_change

This function will append the new radio button to group. “first True” will set to True if group is empty.

Signals supported: `'change'`

Register signal handler with:

```
urwid.connect_signal(radio_button, 'change', callback, user_data)
```

where callback is callback(radio_button, new_state [,user_data]) Unregister signal handlers with:

```
urwid.disconnect_signal(radio_button, 'change', callback, user_data)
```

```
>>> bgroup = [] # button group
>>> b1 = RadioButton(bgroup, u"Agree")
>>> b2 = RadioButton(bgroup, u"Disagree")
>>> len(bgroup)
2
>>> b1
<RadioButton selectable flow widget 'Agree' state=True>
>>> b2
<RadioButton selectable flow widget 'Disagree' state=False>
>>> b2.render((15,), focus=True).text # ... = b in Python 3
[...'( ) Disagree   ']
```

set_state(*state*, *do_callback=True*)

Set the RadioButton state.

state – True, False or "mixed"

do_callback – False to supress signal from this change

If state is True all other radio buttons in the same button group will be set to False.

```
>>> bgroup = [] # button group
>>> b1 = RadioButton(bgroup, u"Agree")
>>> b2 = RadioButton(bgroup, u"Disagree")
>>> b3 = RadioButton(bgroup, u"Unsure")
>>> b1.state, b2.state, b3.state
(True, False, False)
>>> b2.set_state(True)
>>> b1.state, b2.state, b3.state
(False, True, False)
>>> def relabel_button(radio_button, new_state):
...     radio_button.set_label(u"Think Harder!")
>>> connect_signal(b3, 'change', relabel_button)
>>> b3
<RadioButton selectable flow widget 'Unsure' state=False>
>>> b3.set_state(True) # this will trigger the callback
>>> b3
<RadioButton selectable flow widget 'Think Harder!' state=True>
```

toggle_state()

Set state to True.

```
>>> bgroup = [] # button group
>>> b1 = RadioButton(bgroup, "Agree")
>>> b2 = RadioButton(bgroup, "Disagree")
>>> b1.state, b2.state
(True, False)
>>> b2.toggle_state()
>>> b1.state, b2.state
(False, True)
>>> b2.toggle_state()
>>> b1.state, b2.state
(False, True)
```

TreeWidget

class `urwid.`**`TreeWidget`**(*node*)

A widget representing something in a nested tree display.

`first_child`()

Return first child if expanded.

`keypress`(*size*, *key*)

Handle expand & collapse requests (non-leaf nodes)

`last_child`()

Return last child if expanded.

`next_inorder`()

Return the next TreeWidget depth first from this one.

`prev_inorder`()

Return the previous TreeWidget depth first from this one.

`selectable`()

Allow selection of non-leaf nodes so children may be (un)expanded

`update_expanded_icon`()

Update display widget text for parent widgets

SelectableIcon

class `urwid.`**`SelectableIcon`**(*text*, *cursor_position=1*)

Parameters

- **text** – markup for this widget; see `Text` for description of text markup
- **cursor_position** – position the cursor will appear in the text when this widget is in focus

This is a text widget that is selectable. A cursor displayed at a fixed location in the text when in focus. This widget has no special handling of keyboard or mouse input.

`get_cursor_coords`(*size*)

Return the position of the cursor if visible. This method is required for widgets that display a cursor.

`keypress`(*size*, *key*)

No keys are handled by this widget. This method is required for selectable widgets.

`render`(*size*, *focus=False*)

Render the text content of this widget with a cursor when in focus.

```
>>> si = SelectableIcon(u"[!]")
>>> si
<SelectableIcon selectable flow widget '[!]'>
>>> si.render((4,), focus=True).cursor
(1, 0)
>>> si = SelectableIcon("((*))", 2)
>>> si.render((8,), focus=True).cursor
(2, 0)
>>> si.render((2,), focus=True).cursor
(0, 1)
```

4.2.3 Decoration Widget Classes

AttrMap

class `urwid.AttrMap`(*w*, *attr_map*, *focus_map=None*)

AttrMap is a decoration that maps one set of attributes to another. This object will pass all function calls and variable references to the wrapped widget.

Parameters

- **w** (*widget*) – widget to wrap (stored as self.original_widget)
- **attr_map** (*display attribute or dict*) – attribute to apply to *w*, or dict of old display attribute: new display attribute mappings
- **focus_map** (*display attribute or dict*) – attribute to apply when in focus or dict of old display attribute: new display attribute mappings; if `None` use *attr*

```
>>> AttrMap(Divider(u"!"), 'bright')
<AttrMap flow widget <Divider flow widget '!'> attr_map={None: 'bright'}>
>>> AttrMap(Edit(), 'notfocus', 'focus')
<AttrMap selectable flow widget <Edit selectable flow widget '' edit_pos=0> attr_map={None: 'not
>>> size = (5,)
>>> am = AttrMap(Text(u"hi"), 'greeting', 'fgreet')
>>> am.render(size, focus=False).content().next() # ... = b in Python 3
[('greeting', None, ...'hi   ')]
>>> am.render(size, focus=True).content().next()
[('fgreet', None, ...'hi   ')]
>>> am2 = AttrMap(Text(('word', u"hi")), {'word':'greeting', None:'bg'})
>>> am2
<AttrMap flow widget <Text flow widget 'hi'> attr_map={'word': 'greeting', None: 'bg'}>
>>> am2.render(size).content().next()
[('greeting', None, ...'hi'), ('bg', None, ...'   ')]
```

`render`(*size*, *focus=False*)

Render wrapped widget and apply attribute. Return canvas.

`set_attr_map`(*attr_map*)

Set the attribute mapping dictionary {from_attr: to_attr, ...}

Note this function does not accept a single attribute the way the constructor does. You must specify {None: attribute} instead.

```
>>> w = AttrMap(Text(u"hi"), None)
>>> w.set_attr_map({'a':'b'})
>>> w
<AttrMap flow widget <Text flow widget 'hi'> attr_map={'a': 'b'}>
```

`set_focus_map`(*focus_map*)

Set the focus attribute mapping dictionary {from_attr: to_attr, ...}

If None this widget will use the attr mapping instead (no change when in focus).

Note this function does not accept a single attribute the way the constructor does. You must specify {None: attribute} instead.

```
>>> w = AttrMap(Text(u"hi"), {})
>>> w.set_focus_map({'a':'b'})
>>> w
<AttrMap flow widget <Text flow widget 'hi'> attr_map={} focus_map={'a': 'b'}>
>>> w.set_focus_map(None)
>>> w
<AttrMap flow widget <Text flow widget 'hi'> attr_map={}>
```

Padding

class `urwid.`**`Padding`**(*w*, *align='left'*, *width=('relative', 100)*, *min_width=None*, *left=0*, *right=0*)

Parameters

- **w** (*Widget*) – a box, flow or fixed widget to pad on the left and/or right this widget is stored as self.original_widget
- **align** – one of: `'left'`, `'center'`, `'right'` (`'relative'`, *percentage* 0=left 100=right)
- **width** – one of:

  given width integer number of columns for self.original_widget

  `'pack'` try to pack self.original_widget to its ideal size

  (`'relative'`, *percentage of total width*) make width depend on the container's width

  `'clip'` to enable clipping mode for a fixed widget

- **min_width** (*int*) – the minimum number of columns for self.original_widget or `None`
- **left** (*int*) – a fixed number of columns to pad on the left
- **right** (*int*) – a fixed number of columns to pad on thr right

Clipping Mode: (width= `'clip'`) In clipping mode this padding widget will behave as a flow widget and self.original_widget will be treated as a fixed widget. self.original_widget will will be clipped to fit the available number of columns. For example if align is `'left'` then self.original_widget may be clipped on the right.

```
>>> size = (7,)
>>> def pr(w):
...     for t in w.render(size).text:
...         print "|%s|" % (t.decode('ascii'),)
>>> pr(Padding(Text(u"Head"), ('relative', 20), 'pack'))
| Head  |
>>> pr(Padding(Divider(u"-"), left=2, right=1))
|  ---- |
>>> pr(Padding(Divider(u"*"), 'center', 3))
|  ***  |
>>> p=Padding(Text(u"1234"), 'left', 2, None, 1, 1)
>>> p
<Padding flow widget <Text flow widget '1234'> left=1 right=1 width=2>
>>> pr(p)   # align against left
| 12    |
| 34    |
>>> p.align = 'right'
>>> pr(p)   # align against right
|    12 |
|    34 |
>>> pr(Padding(Text(u"hi\nthere"), 'right', 'pack')) # pack text first
|  hi   |
|  there|
```

`align`
    Return the padding alignment setting.

`get_cursor_coords` (*size*)
Return the (x,y) coordinates of cursor within self._original_widget.

`get_pref_col` (*size*)
Return the preferred column from self._original_widget, or None.

`keypress` (*size*, *key*)
Pass keypress to self._original_widget.

`mouse_event` (*size*, *event*, *button*, *x*, *y*, *focus*)
Send mouse event if position is within self._original_widget.

`move_cursor_to_coords` (*size*, *x*, *y*)
Set the cursor position with (x,y) coordinates of self._original_widget.

Returns True if move succeeded, False otherwise.

`padding_values` (*size*, *focus*)
Return the number of columns to pad on the left and right.

Override this method to define custom padding behaviour.

`rows` (*size*, *focus=False*)
Return the rows needed for self.original_widget.

`width`
Return the padding widthment setting.

Filler

class `urwid.`**`Filler`** (*body*, *valign='middle'*, *height='pack'*, *min_height=None*, *top=0*, *bottom=0*)

Parameters

- **body** (*Widget*) – a flow widget or box widget to be filled around (stored as self.original_widget)
- **valign** – one of: `'top'`, `'middle'`, `'bottom'`, (`'relative'`, *percentage* 0=top 100=bottom)
- **height** – one of:

  `'pack'` if body is a flow widget

  given height integer number of rows for self.original_widget

  (`'relative'`, *percentage of total height*) make height depend on container's height
- **min_height** – one of:

  `None` if no minimum or if body is a flow widget

  minimum height integer number of rows for the widget when height not fixed
- **top** (*int*) – a fixed number of rows to fill at the top
- **bottom** (*int*) – a fixed number of rows to fill at the bottom

If body is a flow widget then height must be `'flow'` and *min_height* will be ignored.

Filler widgets will try to satisfy height argument first by reducing the valign amount when necessary. If height still cannot be satisfied it will also be reduced.

filler_values (*size*, *focus*)
Return the number of rows to pad on the top and bottom.

Override this method to define custom padding behaviour.

get_cursor_coords (*size*)
Return cursor coords from self.original_widget if any.

get_pref_col (*size*)
Return pref_col from self.original_widget if any.

keypress (*size*, *key*)
Pass keypress to self.original_widget.

mouse_event (*size*, *event*, *button*, *col*, *row*, *focus*)
Pass to self.original_widget.

move_cursor_to_coords (*size*, *col*, *row*)
Pass to self.original_widget.

render (*size*, *focus=False*)
Render self.original_widget with space above and/or below.

selectable ()
Return selectable from body.

Divider

class urwid.**Divider** (*div_char=u' '*, *top=0*, *bottom=0*)
Horizontal divider widget

Parameters

- **div_char** (*bytes or unicode*) – character to repeat across line
- **top** (*int*) – number of blank lines above
- **bottom** (*int*) – number of blank lines below

```
>>> Divider()
<Divider flow widget>
>>> Divider(u'-')
<Divider flow widget '-'>
>>> Divider(u'x', 1, 2)
<Divider flow widget 'x' bottom=2 top=1>
```

render (*size*, *focus=False*)
Render the divider as a canvas and return it.

```
>>> Divider().render((10,)).text # ... = b in Python 3
[...'          ']
>>> Divider(u'-', top=1).render((10,)).text
[...'          ', ...'----------']
>>> Divider(u'x', bottom=2).render((5,)).text
[...'xxxxx', ...'     ', ...'     ']
```

rows (*size*, *focus=False*)
Return the number of lines that will be rendered.

```
>>> Divider().rows((10,))
1
```

```
>>> Divider(u'x', 1, 2).rows((10,))
4
```

LineBox

class `urwid.`**`LineBox`**(*original_widget*, *title=''*, *tlcorner=u'u250c'*, *tline=u'u2500'*, *lline=u'u2502'*, *trcorner=u'u2510'*, *blcorner=u'u2514'*, *rline=u'u2502'*, *bline=u'u2500'*, *brcorner=u'u2518'*)

Draw a line around original_widget.

Use 'title' to set an initial title text with will be centered on top of the box.

You can also override the widgets used for the lines/corners: tline: top line bline: bottom line lline: left line rline: right line tlcorner: top left corner trcorner: top right corner blcorner: bottom left corner brcorner: bottom right corner

SolidFill

class `urwid.`**`SolidFill`**(*fill_char=' '*)

A box widget that fills an area with a single character

Parameters fill_char (*bytes or unicode*) – character to fill area with

```
>>> SolidFill(u'8')
<SolidFill box widget '8'>
```

`render`(*size*, *focus=False*)

Render the Fill as a canvas and return it.

```
>>> SolidFill().render((4,2)).text # ... = b in Python 3
[...'    ', ...'    ']
>>> SolidFill('#').render((5,3)).text
[...'#####', ...'#####', ...'#####']
```

PopUpLauncher

class `urwid.`**`PopUpLauncher`**(*original_widget*)

`create_pop_up`()

Subclass must override this method and have is return a widget to be used for the pop-up. This method is called once each time the pop-up is opened.

`get_pop_up_parameters`()

Subclass must override this method and have it return a dict, eg:

{'left':0, 'top':1, 'overlay_width':30, 'overlay_height':4}

This method is called each time this widget is rendered.

PopUpTarget

class `urwid.`**`PopUpTarget`**(*original_widget*)

WidgetPlaceholder

class `urwid.`**`WidgetPlaceholder`**`(`*original_widget*`)`

This is a do-nothing decoration widget that can be used for swapping between widgets without modifying the container of this widget.

This can be useful for making an interface with a number of distinct pages or for showing and hiding menu or status bars.

The widget displayed is stored as the self.original_widget property and can be changed by assigning a new widget to it.

WidgetDisable

class `urwid.`**`WidgetDisable`**`(`*original_widget*`)`

A decoration widget that disables interaction with the widget it wraps. This widget always passes focus=False to the wrapped widget, even if it somehow does become the focus.

4.2.4 Container Widget Classes

Frame

class `urwid.`**`Frame`**`(`*body*, *header=None*, *footer=None*, *focus_part='body'*`)`

Frame widget is a box widget with optional header and footer flow widgets placed above and below the box widget.

Note: The main difference between a Frame and a `Pile` widget defined as: *Pile([('pack', header), body, ('pack', footer)])* is that the Frame will not automatically change focus up and down in response to keystrokes.

Parameters

- **body** (*Widget*) – a box widget for the body of the frame
- **header** (*Widget*) – a flow widget for above the body (or None)
- **footer** (*Widget*) – a flow widget for below the body (or None)
- **focus_part** (*str*) – 'header', 'footer' or 'body'

`contents`

a dict-like object similar to:

```
{
    'body': (body_widget, None),
    'header': (header_widget, None),  # if frame has a header
    'footer': (footer_widget, None) # if frame has a footer
}
```

This object may be used to read or update the contents of the Frame.

The values are similar to the the list-like .contents objects used in other containers with (`Widget`, options) tuples, but are constrained to keys for each of the three usual parts of a Frame. When other keys are used a `KeyError` will be raised.

Currently all options are *None*, but using the `options()` method to create the options value is recommended for forwards compatibility.

`focus`
child `Widget` in focus: the body, header or footer widget. This is a read-only property.

`focus_position`
writeable property containing an indicator which part of the frame that is in focus: *'body', 'header'* or *'footer'*.

`frame_top_bottom`(*size*, *focus*)
Calculate the number of rows for the header and footer.

Parameters

- **size** (*widget size*) – See `Widget.render()` for details
- **focus** (*bool*) – `True` if this widget is in focus

Returns *(head rows, foot rows),(orig head, orig foot)* orig head/foot are from rows() calls.

Return type (int, int), (int, int)

`get_focus`()
Return an indicator which part of the frame is in focus

Note: included for backwards compatibility. You should rather use the container property `focus_position` to get this value.

Returns one of 'header', 'footer' or 'body'.

Return type str

`keypress`(*size*, *key*)
Pass keypress to widget in focus.

`mouse_event`(*size*, *event*, *button*, *col*, *row*, *focus*)
Pass mouse event to appropriate part of frame. Focus may be changed on button 1 press.

`options`()
There are currently no options for Frame contents.

Return None as a placeholder for future options.

`set_focus`(*part*)
Determine which part of the frame is in focus.

Note: included for backwards compatibility. You should rather use the container property `focus_position` to set this value.

Parameters **part** (*str*) – 'header', 'footer' or 'body'

ListBox

class `urwid.`**`ListBox`**(*body*)
a horizontally stacked list of widgets

Parameters **body** (*ListWalker*) – a ListWalker subclass such as `SimpleFocusListWalker` that contains widgets to be displayed inside the list box

calculate_visible(*size*, *focus=False*)

Returns the widgets that would be displayed in the ListBox given the current *size* and *focus*.

see `Widget.render()` for parameter details

Returns (*middle*, *top*, *bottom*) or (`None`, `None`, `None`)

middle (*row offset*(when +ve) or *inset*(when -ve), *focus widget*, *focus position*, *focus rows*, *cursor coords* or `None`)

top (*# lines to trim off top*, list of (*widget*, *position*, *rows*) tuples above focus in order from bottom to top)

bottom (*# lines to trim off bottom*, list of (*widget*, *position*, *rows*) tuples below focus in order from top to bottom)

change_focus(*size*, *position*, *offset_inset=0*, *coming_from=None*, *cursor_coords=None*, *snap_rows=None*)

Change the current focus widget. This is used internally by methods that know the widget's *size*.

See also `set_focus()`.

Parameters

- **size** – see `Widget.render()` for details
- **position** – a position compatible with `self.body.set_focus()`
- **offset_inset** (*int*) – either the number of rows between the top of the listbox and the start of the focus widget (+ve value) or the number of lines of the focus widget hidden off the top edge of the listbox (-ve value) or 0 if the top edge of the focus widget is aligned with the top edge of the listbox (default if unspecified)
- **coming_from** (*str*) – eiter 'above', 'below' or unspecified *None*
- **cursor_coords** (*(int, int)*) – (x, y) tuple indicating the desired column and row for the cursor, a (x,) tuple indicating only the column for the cursor, or unspecified
- **snap_rows** (*int*) – the maximum number of extra rows to scroll when trying to "snap" a selectable focus into the view

contents

An object that allows reading widgets from the ListBox's list walker as a *(widget, options)* tuple. *None* is currently the only value for options.

Warning: This object may not be used to set or iterate over contents.
You must use the list walker stored as `body` to perform manipulation and iteration, if supported.

ends_visible(*size*, *focus=False*)

Return a list that may contain `'top'` and/or `'bottom'`.

i.e. this function will return one of: [], [`'top'`], [`'bottom'`] or [`'top'`, `'bottom'`].

convenience function for checking whether the top and bottom of the list are visible

focus

the child widget in focus or None when ListBox is empty

focus_position

the position of child widget in focus. The valid values for this position depend on the list walker in use. `IndexError` will be raised by reading this property when the ListBox is empty or setting this property to an invalid position.

get_cursor_coords(*size*)
See `Widget.get_cursor_coords()` for details

get_focus()
Return a *(focus widget, focus position)* tuple, for backwards compatibility. You may also use the new standard container properties `focus` and `focus_position` to read these values.

get_focus_offset_inset(*size*)
Return (offset rows, inset rows) for focus widget.

keypress(*size*, *key*)
Move selection through the list elements scrolling when necessary. 'up' and 'down' are first passed to widget in focus in case that widget can handle them. 'page up' and 'page down' are always handled by the ListBox.

Keystrokes handled by this widget are: 'up' up one line (or widget) 'down' down one line (or widget) 'page up' move cursor up one listbox length 'page down' move cursor down one listbox length

make_cursor_visible(*size*)
Shift the focus widget so that its cursor is visible.

mouse_event(*size*, *event*, *button*, *col*, *row*, *focus*)
Pass the event to the contained widgets. May change focus on button 1 press.

options()
There are currently no options for ListBox contents.

Return None as a placeholder for future options.

render(*size*, *focus=False*)
Render ListBox and return canvas.

see `Widget.render()` for details

set_focus(*position*, *coming_from=None*)
Set the focus position and try to keep the old focus in view.

Parameters

- **position** – a position compatible with `self.body.set_focus()`
- **coming_from** (*str*) – set to 'above' or 'below' if you know that old position is above or below the new position.

set_focus_valign(*valign*)
Set the focus widget's display offset and inset.

Parameters **valign** – one of: 'top', 'middle', 'bottom' ('fixed top', rows) ('fixed bottom', rows) ('relative', percentage 0=top 100=bottom)

shift_focus(*size*, *offset_inset*)
Move the location of the current focus relative to the top. This is used internally by methods that know the widget's *size*.

See also `set_focus_valign()`.

Parameters

- **size** – see `Widget.render()` for details
- **offset_inset** (*int*) – either the number of rows between the top of the listbox and the start of the focus widget (+ve value) or the number of lines of the focus widget hidden off the top edge of the listbox (-ve value) or `0` if the top edge of the focus widget is aligned with the top edge of the listbox.

update_pref_col_from_focus (*size*)
Update self.pref_col from the focus widget.

TreeListBox

class urwid.**TreeListBox** (*body*)
A ListBox with special handling for navigation and collapsing of TreeWidgets

Parameters **body** (*ListWalker*) – a ListWalker subclass such as `SimpleFocusListWalker` that contains widgets to be displayed inside the list box

collapse_focus_parent (*size*)
Collapse parent directory.

focus_end (*size*)
Move focus to far bottom.

focus_home (*size*)
Move focus to very top.

move_focus_to_parent (*size*)
Move focus to parent of widget in focus.

unhandled_input (*size*, *input*)
Handle macro-navigation keys

Columns

class urwid.**Columns** (*widget_list*, *dividechars=0*, *focus_column=None*, *min_width=1*, *box_columns=None*)
Widgets arranged horizontally in columns from left to right

Parameters

- **widget_list** – iterable of flow or box widgets
- **dividechars** – number of blank characters between columns
- **focus_column** – index into widget_list of column in focus, if `None` the first selectable widget will be chosen.
- **min_width** – minimum width for each column which is not calling widget.pack() in *widget_list*.
- **box_columns** – a list of column indexes containing box widgets whose height is set to the maximum of the rows required by columns not listed in *box_columns*.

widget_list may also contain tuples such as:

(*given_width*, *widget*) make this column *given_width* screen columns wide, where *given_width* is an int

('pack', *widget*) call `pack()` to calculate the width of this column

('weight', *weight*, *widget*)‘ give this column a relative *weight* (number) to calculate its width from the screen columns remaining

Widgets not in a tuple are the same as (`'weight'`, `1`, *widget*)

If the Columns widget is treated as a box widget then all children are treated as box widgets, and *box_columns* is ignored.

If the Columns widget is treated as a flow widget then the rows are calcualated as the largest rows() returned from all columns except the ones listed in *box_columns*. The box widgets in *box_columns* will be displayed with this calculated number of rows, filling the full height.

`box_columns`
A list of the indexes of the columns that are to be treated as box widgets when the Columns is treated as a flow widget.

Note: only for backwards compatibility. You should use the new standard container property `contents`.

`column_types`
A list of the old partial options values for widgets in this Pile, for backwards compatibility only. You should use the new standard container property .contents to modify Pile contents.

`column_widths`(*size*, *focus=False*)
Return a list of column widths.

0 values in the list mean hide corresponding column completely

`contents`
The contents of this Columns as a list of *(widget, options)* tuples. This list may be modified like a normal list and the Columns widget will update automatically.

See also:

Create new options tuples with the `options()` method

`focus`
the child widget in focus or None when Columns is empty

`focus_col`
A property for reading and setting the index of the column in focus.

Note: only for backwards compatibility. You may also use the new standard container property `focus_position` to get the focus.

`focus_position`
index of child widget in focus. Raises IndexError if read when Columns is empty, or when set to an invalid index.

`get_cursor_coords`(*size*)
Return the cursor coordinates from the focus widget.

`get_focus`()
Return the widget in focus, for backwards compatibility. You may also use the new standard container property .focus to get the child widget in focus.

`get_focus_column`()
Return the focus column index.

Note: only for backwards compatibility. You may also use the new standard container property `focus_position` to get the focus.

`get_pref_col`(*size*)
Return the pref col from the column in focus.

`has_flow_type`
Deprecated since version 1.0: Read values from `contents` instead.

keypress (*size*, *key*)
Pass keypress to the focus column.

Parameters size (*int, int*) – *(maxcol,)* if `widget_list` contains flow widgets or *(maxcol, maxrow)* if it contains box widgets.

mouse_event (*size*, *event*, *button*, *col*, *row*, *focus*)
Send event to appropriate column. May change focus on button 1 press.

move_cursor_to_coords (*size*, *col*, *row*)
Choose a selectable column to focus based on the coords.

see `Widget.move_cursor_coords()` for details

options (*width_type='weight'*, *width_amount=1*, *box_widget=False*)
Return a new options tuple for use in a Pile's .contents list.

This sets an entry's width type: one of the following:

'pack' Call the widget's `Widget.pack()` method to determine how wide this column should be. *width_amount* is ignored.

'given' Make column exactly width_amount screen-columns wide.

'weight' Allocate the remaining space to this column by using *width_amount* as a weight value.

Parameters

- **width_type** – `'pack'`, `'given'` or `'weight'`
- **width_amount** – `None` for `'pack'`, a number of screen columns for `'given'` or a weight value (number) for `'weight'`
- **box_widget** (*bool*) – set to *True* if this widget is to be treated as a box widget when the Columns widget itself is treated as a flow widget.

render (*size*, *focus=False*)
Render columns and return canvas.

Parameters

- **size** – see `Widget.render()` for details
- **focus** (*bool*) – `True` if this widget is in focus

rows (*size*, *focus=False*)
Return the number of rows required by the columns. This only makes sense if `widget_list` contains flow widgets.

see `Widget.rows()` for details

selectable ()
Return the selectable value of the focus column.

set_focus (*item*)
Set the item in focus

Note: only for backwards compatibility. You may also use the new standard container property `focus_position` to get the focus.

Parameters item – widget or integer index

set_focus_column(*num*)

Set the column in focus by its index in `widget_list`.

Parameters **num** (*int*) – index of focus-to-be entry

Note: only for backwards compatibility. You may also use the new standard container property `focus_position` to set the focus.

widget_list

A list of the widgets in this Columns

Note: only for backwards compatibility. You should use the new standard container property `contents`.

Pile

class `urwid.`**Pile**(*widget_list*, *focus_item=None*)

A pile of widgets stacked vertically from top to bottom

Parameters

- **widget_list** (*iterable*) – child widgets
- **focus_item** (*Widget or int*) – child widget that gets the focus initially. Chooses the first selectable widget if unset.

widget_list may also contain tuples such as:

(*given_height*, *widget*) always treat *widget* as a box widget and give it *given_height* rows, where given_height is an int

(`'pack'`, *widget*) allow *widget* to calculate its own height by calling its `rows()` method, ie. treat it as a flow widget.

(`'weight'`, *weight*, *widget*) if the pile is treated as a box widget then treat widget as a box widget with a height based on its relative weight value, otherwise treat the same as (`'pack'`, *widget*).

Widgets not in a tuple are the same as (`'weight'`, `1`, *widget*)‘

Note: If the Pile is treated as a box widget there must be at least one `'weight'` tuple in `widget_list`.

contents

The contents of this Pile as a list of (widget, options) tuples.

options currently may be one of

(`'pack'`, `None`) allow widget to calculate its own height by calling its `rows` method, i.e. treat it as a flow widget.

(`'given'`, *n*) Always treat widget as a box widget with a given height of *n* rows.

(`'weight'`, *w*) If the Pile itself is treated as a box widget then the value *w* will be used as a relative weight for assigning rows to this box widget. If the Pile is being treated as a flow widget then this is the same as (`'pack'`, `None`) and the *w* value is ignored.

If the Pile itself is treated as a box widget then at least one widget must have a (`'weight'`, *w*) options value, or the Pile will not be able to grow to fill the required number of rows.

This list may be modified like a normal list and the Pile widget will updated automatically.

See also:

Create new options tuples with the `options()` method

`focus`
the child widget in focus or None when Pile is empty

`focus_item`
A property for reading and setting the widget in focus.

Note: only for backwards compatibility. You should use the new standard container properties `focus` and `focus_position` to get the child widget in focus or modify the focus position.

`focus_position`
index of child widget in focus. Raises `IndexError` if read when Pile is empty, or when set to an invalid index.

`get_cursor_coords`(*size*)
Return the cursor coordinates of the focus widget.

`get_focus`()
Return the widget in focus, for backwards compatibility. You may also use the new standard container property .focus to get the child widget in focus.

`get_item_rows`(*size*, *focus*)
Return a list of the number of rows used by each widget in self.contents

`get_item_size`(*size*, *i*, *focus*, *item_rows=None*)
Return a size appropriate for passing to self.contents[i][0].render

`get_pref_col`(*size*)
Return the preferred column for the cursor, or None.

`item_types`
A list of the options values for widgets in this Pile.

Note: only for backwards compatibility. You should use the new standard container property `contents`.

`keypress`(*size*, *key*)
Pass the keypress to the widget in focus. Unhandled 'up' and 'down' keys may cause a focus change.

`mouse_event`(*size*, *event*, *button*, *col*, *row*, *focus*)
Pass the event to the contained widget. May change focus on button 1 press.

`move_cursor_to_coords`(*size*, *col*, *row*)
Capture pref col and set new focus.

`options`(*height_type='weight'*, *height_amount=1*)
Return a new options tuple for use in a Pile's `contents` list.

Parameters

- **height_type** – `'pack'`, `'given'` or `'weight'`
- **height_amount** – `None` for `'pack'`, a number of rows for `'fixed'` or a weight value (number) for `'weight'`

`selectable`()
Return True if the focus item is selectable.

`set_focus`(*item*)
Set the item in focus, for backwards compatibility.

Note: only for backwards compatibility. You should use the new standard container property `focus_position`. to set the position by integer index instead.

Parameters **item** (*Widget or int*) – element to focus

`widget_list`
A list of the widgets in this Pile

Note: only for backwards compatibility. You should use the new standard container property `contents`.

GridFlow

class `urwid.`**`GridFlow`**(*cells*, *cell_width*, *h_sep*, *v_sep*, *align*)
The GridFlow widget is a flow widget that renders all the widgets it contains the same width and it arranges them from left to right and top to bottom.

Parameters

- **cells** – list of flow widgets to display
- **cell_width** – column width for each cell
- **h_sep** – blank columns between each cell horizontally
- **v_sep** – blank rows between cells vertically (if more than one row is required to display all the cells)
- **align** – horizontal alignment of cells, one of: 'left', 'center', 'right', ('relative', percentage 0=left 100=right)

`cell_width`
The width of each cell in the GridFlow. Setting this value affects all cells.

`cells`
A list of the widgets in this GridFlow

Note: only for backwards compatibility. You should use the new use the new standard container property `contents` to modify GridFlow contents.

`contents`
The contents of this GridFlow as a list of (widget, options) tuples.

options is currently a tuple in the form *('fixed', number)*. number is the number of screen columns to allocate to this cell. 'fixed' is the only type accepted at this time.

This list may be modified like a normal list and the GridFlow widget will update automatically.

See also:

Create new options tuples with the `options()` method.

`focus`
the child widget in focus or None when GridFlow is empty

`focus_cell`
The widget in focus, for backwards compatibility.

Note: only for backwards compatibility. You should use the new use the new standard container property `focus` to get the widget in focus and `focus_position` to get/set the cell in focus by index.

---

focus_position
index of child widget in focus. Raises `IndexError` if read when GridFlow is empty, or when set to an invalid index.

generate_display_widget (*size*)
Actually generate display widget (ignoring cache)

get_cursor_coords (*size*)
Get cursor from display widget.

get_display_widget (*size*)
Arrange the cells into columns (and possibly a pile) for display, input or to calculate rows, and update the display widget.

get_focus ()
Return the widget in focus, for backwards compatibility.

Note: only for backwards compatibility. You may also use the new standard container property `focus` to get the focus.

get_pref_col (*size*)
Return pref col from display widget.

keypress (*size*, *key*)
Pass keypress to display widget for handling. Captures focus changes.

move_cursor_to_coords (*size*, *col*, *row*)
Set the widget in focus based on the col + row.

options (*width_type='given'*, *width_amount=None*)
Return a new options tuple for use in a GridFlow's .contents list.

width_type – 'given' is the only value accepted width_amount – None to use the default cell_width for this GridFlow

set_focus (*cell*)
Set the cell in focus, for backwards compatibility.

Note: only for backwards compatibility. You may also use the new standard container property `focus_position` to get the focus.

Parameters **cell** (*Widget or int*) – contained element to focus

BoxAdapter

class urwid.**BoxAdapter** (*box_widget*, *height*)
Adapter for using a box widget where a flow widget would usually go

Create a flow widget that contains a box widget

Parameters

- **box_widget** (*Widget*) – box widget to wrap
- **height** (*int*) – number of rows for box widget

```
>>> BoxAdapter(SolidFill(u"x"), 5) # 5-rows of x's
<BoxAdapter flow widget <SolidFill box widget 'x'> height=5>
```

rows (*size*, *focus=False*)
Return the predetermined height (behave like a flow widget)

```
>>> BoxAdapter(SolidFill(u"x"), 5).rows((20,))
5
```

Overlay

class urwid.**Overlay** (*top_w*, *bottom_w*, *align*, *width*, *valign*, *height*, *min_width=None*, *min_height=None*, *left=0*, *right=0*, *top=0*, *bottom=0*)
Overlay contains two box widgets and renders one on top of the other

Parameters

- **top_w** (*Widget*) – a flow, box or fixed widget to overlay "on top"
- **bottom_w** (*Widget*) – a box widget to appear "below" previous widget
- **align** (*str*) – alignment, one of `'left'`, `'center'`, `'right'` or (`'relative'`, *percentage* 0=left 100=right)
- **width** – width type, one of:

  `'pack'` if *top_w* is a fixed widget

  given width integer number of columns wide

  (`'relative'`, *percentage of total width*) make *top_w* width related to container width
- **valign** – alignment mode, one of `'top'`, `'middle'`, `'bottom'` or (`'relative'`, *percentage* 0=top 100=bottom)
- **height** – one of:

  `'pack'` if *top_w* is a flow or fixed widget

  given height integer number of rows high

  (`'relative'`, *percentage of total height*) make *top_w* height related to container height
- **min_width** (*int*) – the minimum number of columns for *top_w* when width is not fixed
- **min_height** (*int*) – minimum number of rows for *top_w* when height is not fixed
- **left** (*int*) – a fixed number of columns to add on the left
- **right** (*int*) – a fixed number of columns to add on the right
- **top** (*int*) – a fixed number of rows to add on the top
- **bottom** (*int*) – a fixed number of rows to add on the bottom

Overlay widgets behave similarly to `Padding` and `Filler` widgets when determining the size and position of *top_w*. *bottom_w* is always rendered the full size available "below" *top_w*.

calculate_padding_filler (*size*, *focus*)
Return (padding left, right, filler top, bottom).

contents
a list-like object similar to:

```
[(bottom_w, bottom_options)),
 (top_w, top_options)]
```

This object may be used to read or update top and bottom widgets and top widgets's options, but no widgets may be added or removed.

top_options takes the form *(align_type, align_amount, width_type, width_amount, min_width, left, right, valign_type, valign_amount, height_type, height_amount, min_height, top, bottom)*

bottom_options is always *('left', None, 'relative', 100, None, 0, 0, 'top', None, 'relative', 100, None, 0, 0)* which means that bottom widget always covers the full area of the Overlay. writing a different value for *bottom_options* raises an `OverlayError`.

focus
the top widget in this overlay is always in focus

focus_position
index of child widget in focus, currently always 1

get_cursor_coords(*size*)
Return cursor coords from top_w, if any.

keypress(*size*, *key*)
Pass keypress to top_w.

mouse_event(*size*, *event*, *button*, *col*, *row*, *focus*)
Pass event to top_w, ignore if outside of top_w.

options(*align_type*, *align_amount*, *width_type*, *width_amount*, *valign_type*, *valign_amount*, *height_type*, *height_amount*, *min_width=None*, *min_height=None*, *left=0*, *right=0*, *top=0*, *bottom=0*)
Return a new options tuple for use in this Overlay's .contents mapping.

This is the common container API to create options for replacing the top widget of this Overlay. It is provided for completeness but is not necessarily the easiest way to change the overlay parameters. See also `set_overlay_parameters()`

render(*size*, *focus=False*)
Render top_w overlayed on bottom_w.

selectable()
Return selectable from top_w.

set_overlay_parameters(*align*, *width*, *valign*, *height*, *min_width=None*, *min_height=None*, *left=0*, *right=0*, *top=0*, *bottom=0*)
Adjust the overlay size and position parameters.

See `__init__()` for a description of the parameters.

top_w_size(*size*, *left*, *right*, *top*, *bottom*)
Return the size to pass to top_w.

4.2.5 Graphic Widget Classes

BarGraph

class `urwid.`**BarGraph**(*attlist*, *hatt=None*, *satt=None*)
Create a bar graph with the passed display characteristics. see set_segment_attributes for a description of the parameters.

calculate_bar_widths(*size*, *bardata*)
Return a list of bar widths, one for each bar in data.

If self.bar_width is None this implementation will stretch the bars across the available space specified by maxcol.

`calculate_display`(*size*)
Calculate display data.

`hlines_display`(*disp*, *top*, *hlines*, *maxrow*)
Add hlines to display structure represented as bar_type tuple values: (bg, 0-5) bg is the segment that has the hline on it 0-5 is the hline graphic to use where 0 is a regular underscore and 1-5 are the UTF-8 horizontal scan line characters.

`render`(*size*, *focus=False*)
Render BarGraph.

`selectable`()
Return False.

`set_bar_width`(*width*)
Set a preferred bar width for calculate_bar_widths to use.

width – width of bar or None for automatic width adjustment

`set_data`(*bardata*, *top*, *hlines=None*)
Store bar data, bargraph top and horizontal line positions.

bardata – a list of bar values. top – maximum value for segments within bardata hlines – None or a bar value marking horizontal line positions

bar values are [segment1, segment2, ...] lists where top is the maximal value corresponding to the top of the bar graph and segment1, segment2, ... are the values for the top of each segment of this bar. Simple bar graphs will only have one segment in each bar value.

Eg: if top is 100 and there is a bar value of [80, 30] then the top of this bar will be at 80% of full height of the graph and it will have a second segment that starts at 30%.

`set_segment_attributes`(*attlist*, *hatt=None*, *satt=None*)

Parameters

- **attlist** – list containing display attribute or (display attribute, character) tuple for background, first segment, and optionally following segments. ie. len(attlist) == num segments+1 character defaults to ' ' if not specified.
- **hatt** – list containing attributes for horizontal lines. First element is for lines on background, second is for lines on first segment, third is for lines on second segment etc.
- **satt** – dictionary containing attributes for smoothed transitions of bars in UTF-8 display mode. The values are in the form:

  (fg,bg) : attr

  fg and bg are integers where 0 is the graph background, 1 is the first segment, 2 is the second, ... fg > bg in all values. attr is an attribute with a foreground corresponding to fg and a background corresponding to bg.

If satt is not None and the bar graph is being displayed in a terminal using the UTF-8 encoding then the character cell that is shared between the segments specified will be smoothed with using the UTF-8 vertical eighth characters.

eg: set_segment_attributes(['no', ('unsure',"?"), 'yes']) will use the attribute 'no' for the background (the area from the top of the graph to the top of the bar), question marks with the attribute 'unsure' will be used for the topmost segment of the bar, and the attribute 'yes' will be used for the bottom segment of the bar.

smooth_display (*disp*)
smooth (col, row*8) display into (col, row) display using UTF vertical eighth characters represented as bar_type tuple values: (fg, bg, 1-7) where fg is the lower segment, bg is the upper segment and 1-7 is the vertical eighth character to use.

GraphVScale

class `urwid.`**GraphVScale** (*labels*, *top*)
GraphVScale([(label1 position, label1 markup),...], top) label position – 0 < position < top for the y position label markup – text markup for this label top – top y position

This widget is a vertical scale for the BarGraph widget that can correspond to the BarGraph's horizontal lines

render (*size*, *focus=False*)
Render GraphVScale.

selectable ()
Return False.

set_scale ([*(label1 position*, *label1 markup)*, *...*], *top*)
label position – 0 < position < top for the y position label markup – text markup for this label top – top y position

ProgressBar

class `urwid.`**ProgressBar** (*normal*, *complete*, *current=0*, *done=100*, *satt=None*)

Parameters

- **normal** – display attribute for uncomplete part of progress bar
- **complete** – display attribute for complete part of progress bar
- **current** – current progress
- **done** – progress amount at 100%
- **satt** – display attribute for smoothed part of bar where the foreground of satt corresponds to the normal part and the background corresponds to the complete part. If satt is `None` then no smoothing will be done.

get_text ()
Return the progress bar percentage text.

render (*size*, *focus=False*)
Render the progress bar.

set_completion (*current*)
current – current progress

BigText

class `urwid.`**BigText** (*markup*, *font*)
markup – same as Text widget markup font – instance of a Font class

get_text ()
Returns (text, attributes).

urwid.**get_all_fonts**()

Return a list of (font name, font class) tuples.

Terminal

class urwid.**Terminal**(*command*, *env=None*, *main_loop=None*, *escape_sequence=None*)

A terminal emulator within a widget.

'command' is the command to execute inside the terminal, provided as a list of the command followed by its arguments. If 'command' is None, the command is the current user's shell. You can also provide a callable instead of a command, which will be executed in the subprocess.

'env' can be used to pass custom environment variables. If omitted, os.environ is used.

'main_loop' should be provided, because the canvas state machine needs to act on input from the PTY master device. This object must have watch_file and remove_watch_file methods.

'escape_sequence' is the urwid key symbol which should be used to break out of the terminal widget. If it's not specified, "ctrl a" is used.

change_focus(*has_focus*)

Ignore SIGINT if this widget has focus.

respond(*string*)

Respond to the underlying application with 'string'.

4.3 Display Modules

class urwid.**BaseScreen**

Base class for Screen classes (raw_display.Screen, .. etc)

register_palette(*palette*)

Register a set of palette entries.

palette – a list of (name, like_other_name) or (name, foreground, background, mono, foreground_high, background_high) tuples

The (name, like_other_name) format will copy the settings from the palette entry like_other_name, which must appear before this tuple in the list.

The mono and foreground/background_high values are optional ie. the second tuple format may have 3, 4 or 6 values. See register_palette_entry() for a description of the tuple values.

register_palette_entry(*name*, *foreground*, *background*, *mono=None*, *foreground_high=None*, *background_high=None*)

Register a single palette entry.

name – new entry/attribute name

foreground – a string containing a comma-separated foreground color and settings

Color values: 'default' (use the terminal's default foreground), 'black', 'dark red', 'dark green', 'brown', 'dark blue', 'dark magenta', 'dark cyan', 'light gray', 'dark gray', 'light red', 'light green', 'yellow', 'light blue', 'light magenta', 'light cyan', 'white'

Settings: 'bold', 'underline', 'blink', 'standout'

Some terminals use 'bold' for bright colors. Most terminals ignore the 'blink' setting. If the color is not given then 'default' will be assumed.

background – a string containing the background color

> Background color values: 'default' (use the terminal's default background), 'black', 'dark red', 'dark green', 'brown', 'dark blue', 'dark magenta', 'dark cyan', 'light gray'

mono – a comma-separated string containing monochrome terminal settings (see "Settings" above.)

> None = no terminal settings (same as 'default')

foreground_high – a string containing a comma-separated foreground color and settings, standard foreground colors (see "Color values" above) or high-colors may be used

> High-color example values: '#009' (0% red, 0% green, 60% red, like HTML colors) '#fcc' (100% red, 80% green, 80% blue) 'g40' (40% gray, decimal), 'g#cc' (80% gray, hex), '#000', 'g0', 'g#00' (black), '#fff', 'g100', 'g#ff' (white) 'h8' (color number 8), 'h255' (color number 255)
>
> None = use foreground parameter value

background_high – a string containing the background color, standard background colors (see "Background colors" above) or high-colors (see "High-color example values" above) may be used

> None = use background parameter value

4.3.1 raw_display

class `urwid.raw_display.`**`Screen`**

Initialize a screen that directly prints escape codes to an output terminal.

`clear``()`

Force the screen to be completely repainted on the next call to draw_screen().

`draw_screen``(`*(maxcol, maxrow), r*`)`

Paint screen with rendered canvas.

`get_cols_rows``()`

Return the terminal dimensions (num columns, num rows).

`get_input``(`*raw_keys=False*`)`

Return pending input as a list.

raw_keys – return raw keycodes as well as translated versions

This function will immediately return all the input since the last time it was called. If there is no input pending it will wait before returning an empty list. The wait time may be configured with the set_input_timeouts function.

If raw_keys is False (default) this function will return a list of keys pressed. If raw_keys is True this function will return a (keys pressed, raw keycodes) tuple instead.

Examples of keys returned:

- ASCII printable characters: " ", "a", "0", "A", "-", "/"
- ASCII control characters: "tab", "enter"
- Escape sequences: "up", "page up", "home", "insert", "f1"
- Key combinations: "shift f1", "meta a", "ctrl b"
- Window events: "window resize"

When a narrow encoding is not enabled:

- "Extended ASCII" characters: "xa1", "xb2", "xfe"

When a wide encoding is enabled:

- Double-byte characters: “xa1xea”, “xb2xd4”

When utf8 encoding is enabled:

- Unicode characters: u”u00a5”, u’u253c”

Examples of mouse events returned:

- **Mouse button press: (‘mouse press’, 1, 15, 13),** (‘meta mouse press’, 2, 17, 23)
- **Mouse drag: (‘mouse drag’, 1, 16, 13),** (‘mouse drag’, 1, 17, 13), (‘ctrl mouse drag’, 1, 18, 13)
- **Mouse button release: (‘mouse release’, 0, 18, 13),** (‘ctrl mouse release’, 0, 17, 23)

`get_input_descriptors`()

Return a list of integer file descriptors that should be polled in external event loops to check for user input.

Use this method if you are implementing yout own event loop.

`get_input_nonblocking`()

Return a (next_input_timeout, keys_pressed, raw_keycodes) tuple.

Use this method if you are implementing your own event loop.

When there is input waiting on one of the descriptors returned by get_input_descriptors() this method should be called to read and process the input.

This method expects to be called in next_input_timeout seconds (a floating point number) if there is no input waiting.

`modify_terminal_palette`(*entries*)

entries - list of (index, red, green, blue) tuples.

Attempt to set part of the terminal pallette (this does not work on all terminals.) The changes are sent as a single escape sequence so they should all take effect at the same time.

0 <= index < 256 (some terminals will only have 16 or 88 colors) 0 <= red, green, blue < 256

`reset_default_terminal_palette`()

Attempt to set the terminal palette to default values as taken from xterm. Uses number of colors from current set_terminal_properties() screen setting.

`run_wrapper`(*fn*, *alternate_buffer=True*)

Call start to initialize screen, then call fn. When fn exits call stop to restore the screen to normal.

alternate_buffer – use alternate screen buffer and restore normal screen buffer on exit

`set_input_timeouts`(*max_wait=None*, *complete_wait=0.125*, *resize_wait=0.125*)

Set the get_input timeout values. All values are in floating point numbers of seconds.

max_wait – amount of time in seconds to wait for input when there is no input pending, wait forever if None

complete_wait – amount of time in seconds to wait when get_input detects an incomplete escape sequence at the end of the available input

resize_wait – amount of time in seconds to wait for more input after receiving two screen resize requests in a row to stop Urwid from consuming 100% cpu during a gradual window resize operation

`set_mouse_tracking`()

Enable mouse tracking.

After calling this function get_input will include mouse click events along with keystrokes.

`set_terminal_properties`(*colors=None*, *bright_is_bold=None*, *has_underline=None*)

colors – number of colors terminal supports (1, 16, 88 or 256) or None to leave unchanged

bright_is_bold – set to True if this terminal uses the bold setting to create bright colors (numbers 8-15), set to False if this Terminal can create bright colors without bold or None to leave unchanged

has_underline – set to True if this terminal can use the underline setting, False if it cannot or None to leave unchanged

`signal_init`()
Called in the startup of run wrapper to set the SIGWINCH and SIGCONT signal handlers.

Override this function to call from main thread in threaded applications.

`signal_restore`()
Called in the finally block of run wrapper to restore the SIGWINCH and SIGCONT signal handlers.

Override this function to call from main thread in threaded applications.

`start`(*alternate_buffer=True*)
Initialize the screen and input mode.

alternate_buffer – use alternate screen buffer

`stop`()
Restore the screen.

4.3.2 curses_display

class `urwid.curses_display.`**`Screen`**

`clear`()
Force the screen to be completely repainted on the next call to draw_screen().

`draw_screen`(*(cols, rows), r*)
Paint screen with rendered canvas.

`get_cols_rows`()
Return the terminal dimensions (num columns, num rows).

`get_input`(*raw_keys=False*)
Return pending input as a list.

raw_keys – return raw keycodes as well as translated versions

This function will immediately return all the input since the last time it was called. If there is no input pending it will wait before returning an empty list. The wait time may be configured with the set_input_timeouts function.

If raw_keys is False (default) this function will return a list of keys pressed. If raw_keys is True this function will return a (keys pressed, raw keycodes) tuple instead.

Examples of keys returned:

- ASCII printable characters: ” ”, “a”, “0”, “A”, “-”, “/”
- ASCII control characters: “tab”, “enter”
- Escape sequences: “up”, “page up”, “home”, “insert”, “f1”
- Key combinations: “shift f1”, “meta a”, “ctrl b”
- Window events: “window resize”

When a narrow encoding is not enabled:

- "Extended ASCII" characters: "xa1", "xb2", "xfe"

When a wide encoding is enabled:

- Double-byte characters: "xa1xea", "xb2xd4"

When utf8 encoding is enabled:

- Unicode characters: u"u00a5", u'u253c"

Examples of mouse events returned:

- **Mouse button press: ('mouse press', 1, 15, 13),** ('meta mouse press', 2, 17, 23)
- **Mouse button release: ('mouse release', 0, 18, 13),** ('ctrl mouse release', 0, 17, 23)

`run_wrapper`(*fn*)
Call fn in fullscreen mode. Return to normal on exit.

This function should be called to wrap your main program loop. Exception tracebacks will be displayed in normal mode.

`set_input_timeouts`(*max_wait=None*, *complete_wait=0.1*, *resize_wait=0.1*)
Set the get_input timeout values. All values have a granularity of 0.1s, ie. any value between 0.15 and 0.05 will be treated as 0.1 and any value less than 0.05 will be treated as 0. The maximum timeout value for this module is 25.5 seconds.

max_wait – amount of time in seconds to wait for input when there is no input pending, wait forever if None

complete_wait – amount of time in seconds to wait when get_input detects an incomplete escape sequence at the end of the available input

resize_wait – amount of time in seconds to wait for more input after receiving two screen resize requests in a row to stop urwid from consuming 100% cpu during a gradual window resize operation

`set_mouse_tracking`()
Enable mouse tracking.

After calling this function get_input will include mouse click events along with keystrokes.

`start`()
Initialize the screen and input mode.

`stop`()
Restore the screen.

4.3.3 web_display

class `urwid.web_display.`**`Screen`**

`clear`()
Force the screen to be completely repainted on the next call to draw_screen().

(does nothing for web_display)

`draw_screen`(*(cols*, *rows)*, *r*)
Send a screen update to the client.

`get_cols_rows`()
Return the screen size.

get_input(*raw_keys=False*)
Return pending input as a list.

register_palette(*l*)
Register a list of palette entries.

l – list of (name, foreground, background) or (name, same_as_other_name) palette entries.

calls self.register_palette_entry for each item in l

register_palette_entry(*name*, *foreground*, *background*, *mono=None*)
Register a single palette entry.

name – new entry/attribute name foreground – foreground colour background – background colour mono – monochrome terminal attribute

See curses_display.register_palette_entry for more info.

run_wrapper(*fn*)
Run the application main loop, calling start() first and stop() on exit.

set_mouse_tracking()
Not yet implemented

start()
This function reads the initial screen size, generates a unique id and handles cleanup when fn exits.

web_display.set_preferences(..) must be called before calling this function for the preferences to take effect

stop()
Restore settings and clean up.

tty_signal_keys(**args*, ***vargs*)
Do nothing.

4.3.4 html_fragment

class urwid.html_fragment.**HtmlGenerator**

clear()
Force the screen to be completely repainted on the next call to draw_screen().

(does nothing for html_fragment)

draw_screen(*(cols*, *rows)*, *r*)
Create an html fragment from the render object. Append it to HtmlGenerator.fragments list.

get_cols_rows()
Return the next screen size in HtmlGenerator.sizes.

get_input(*raw_keys=False*)
Return the next list of keypresses in HtmlGenerator.keys.

run_wrapper(*fn*)
Call fn.

set_mouse_tracking()
Not yet implemented

`urwid.html_fragment.`**`screenshot_init`**(*sizes*, *keys*)

Replace curses_display.Screen and raw_display.Screen class with HtmlGenerator.

Call this function before executing an application that uses curses_display.Screen to have that code use HtmlGenerator instead.

sizes – list of (columns, rows) tuples to be returned by each call to HtmlGenerator.get_cols_rows()

keys – list of lists of keys to be returned by each call to HtmlGenerator.get_input()

Lists of keys may include "window resize" to force the application to call get_cols_rows and read a new screen size.

For example, the following call will prepare an application to:

1. start in 80x25 with its first call to get_cols_rows()
2. take a screenshot when it calls draw_screen(..)
3. simulate 5 "down" keys from get_input()
4. take a screenshot when it calls draw_screen(..)
5. simulate keys "a", "b", "c" and a "window resize"
6. resize to 20x10 on its second call to get_cols_rows()
7. take a screenshot when it calls draw_screen(..)
8. simulate a "Q" keypress to quit the application

screenshot_init([(80,25), (20,10)], [["down"]*5, ["a","b","c","window resize"], ["Q"]])

`urwid.html_fragment.`**`screenshot_collect`**()

Return screenshots as a list of HTML fragments.

4.3.5 lcd_display

class `urwid.lcd_display.`**`LCDScreen`**

class `urwid.lcd_display.`**`CFLCDScreen`**(*device_path*, *baud*)

Common methods for Crystal Fontz LCD displays

device_path – eg. '/dev/ttyUSB0' baud – baud rate

class `urwid.lcd_display.`**`CF635Screen`**(*device_path*, *baud=115200*, *repeat_delay=0.5*, *repeat_next=0.125*, *key_map=['up', 'down', 'left', 'right', 'enter', 'esc']*)

Crystal Fontz 635 display

20x4 character display + cursor no foreground/background colors or settings supported

see CGROM for list of close unicode matches to characters available

6 button input up, down, left, right, enter (check mark), exit (cross)

device_path – eg. '/dev/ttyUSB0' baud – baud rate repeat_delay – seconds to wait before starting to repeat keys repeat_next – time between each repeated key key_map – the keys to send for this device's buttons

`get_input_descriptors`()

return the fd from our serial device so we get called on input and responses

`get_input_nonblocking`()

Return a (next_input_timeout, keys_pressed, raw_keycodes) tuple.

The protocol for our device requires waiting for acks between each command, so this method responds to those as well as key press and release events.

Key repeat events are simulated here as the device doesn't send any for us.

raw_keycodes are the bytes of messages we received, which might not seem to have any correspondence to keys_pressed.

program_cgram (*index*, *data*)
Program character data. Characters available as chr(0) through chr(7), and repeated as chr(8) through chr(15).

index – 0 to 7 index of character to program

data – list of 8, 6-bit integer values top to bottom with MSB on the left side of the character.

set_backlight (*value*)
Set backlight brightness

value – 0 to 100

set_cursor_style (*style*)

style – CURSOR_BLINKING_BLOCK, CURSOR_UNDERSCORE,
CURSOR_BLINKING_BLOCK_UNDERSCORE or CURSOR_INVERTING_BLINKING_BLOCK

set_lcd_contrast (*value*)
value – 0 to 255

set_led_pin (*led*, *rg*, *value*)
led – 0 to 3 rg – 0 for red, 1 for green value – 0 to 100

class urwid.lcd_display.**KeyRepeatSimulator** (*repeat_delay*, *repeat_next*)
Provide simulated repeat key events when given press and release events.

If two or more keys are pressed disable repeating until all keys are released.

repeat_delay – seconds to wait before starting to repeat keys repeat_next – time between each repeated key

next_event ()
Return (remaining, key) where remaining is the number of seconds (float) until the key repeat event should be sent, or None if no events are pending.

sent_event ()
Cakk this method when you have sent a key repeat event so the timer will be reset for the next event

4.4 List Walker Classes

4.4.1 ListWalker

class urwid.**ListWalker**
x.__init__(...) initializes x; see help(type(x)) for signature

get_focus ()
This default implementation relies on a focus attribute and a __getitem__() method defined in a subclass.

Override and don't call this method if these are not defined.

get_next (*position*)
This default implementation relies on a next_position() method and a __getitem__() method defined in a subclass.

Override and don't call this method if these are not defined.

get_prev (*position*)
This default implementation relies on a prev_position() method and a __getitem__() method defined in a subclass.

Override and don't call this method if these are not defined.

4.4.2 List-like List Walkers

class `urwid.`**SimpleFocusListWalker** (*contents*)
contents – list to copy into this object

Changes made to this object (when it is treated as a list) are detected automatically and will cause ListBox objects using this list walker to be updated.

Also, items added or removed before the widget in focus with normal list methods will cause the focus to be updated intelligently.

next_position (*position*)
Return position after start_from.

positions (*reverse=False*)
Optional method for returning an iterable of positions.

prev_position (*position*)
Return position before start_from.

set_focus (*position*)
Set focus position.

set_modified_callback (*callback*)
This function inherited from MonitoredList is not implemented in SimpleFocusListWalker.

Use connect_signal(list_walker, "modified", ...) instead.

class `urwid.`**SimpleListWalker** (*contents*)
contents – list to copy into this object

Changes made to this object (when it is treated as a list) are detected automatically and will cause ListBox objects using this list walker to be updated.

contents
Return self.

Provides compatibility with old SimpleListWalker class.

next_position (*position*)
Return position after start_from.

positions (*reverse=False*)
Optional method for returning an iterable of positions.

prev_position (*position*)
Return position before start_from.

set_focus (*position*)
Set focus position.

set_modified_callback (*callback*)
This function inherited from MonitoredList is not implemented in SimpleListWalker.

Use connect_signal(list_walker, "modified", ...) instead.

4.4.3 TreeWalker and Nodes

class `urwid.`**`TreeWalker`**`(`*start_from*`)`
ListWalker-compatible class for displaying TreeWidgets

positions are TreeNodes.

start_from: TreeNode with the initial focus.

class `urwid.`**`TreeNode`**`(`*value*, *parent=None*, *key=None*, *depth=None*`)`
Store tree contents and cache TreeWidget objects. A TreeNode consists of the following elements: * key: accessor token for parent nodes * value: subclass-specific data * parent: a TreeNode which contains a pointer back to this object * widget: The widget used to render the object

`get_widget``(`*reload=False*`)`
Return the widget for this node.

`load_parent``()`
Provide TreeNode with a parent for the current node. This function is only required if the tree was instantiated from a child node (virtual function)

class `urwid.`**`ParentNode`**`(`*value*, *parent=None*, *key=None*, *depth=None*`)`
Maintain sort order for TreeNodes.

`get_child_keys``(`*reload=False*`)`
Return a possibly ordered list of child keys

`get_child_node``(`*key*, *reload=False*`)`
Return the child node for a given key. Create if necessary.

`get_child_widget``(`*key*`)`
Return the widget for a given key. Create if necessary.

`get_first_child``()`
Return the first TreeNode in the directory.

`get_last_child``()`
Return the last TreeNode in the directory.

`has_children``()`
Does this node have any children?

`load_child_keys``()`
Provide ParentNode with an ordered list of child keys (virtual function)

`load_child_node``(`*key*`)`
Load the child node for a given key (virtual function)

`next_child``(`*key*`)`
Return the next child node in index order from the given key.

`prev_child``(`*key*`)`
Return the previous child node in index order from the given key.

`set_child_node``(`*key*, *node*`)`
Set the child node for a given key. Useful for bottom-up, lazy population of a tree..

4.5 Signal Functions

The `urwid.*_signal()` functions use a shared Signals object instance for tracking registered and connected signals. There is no reason to instantiate your own Signals object.

`urwid.`**`connect_signal`**(*obj*, *name*, *callback*, *user_arg=None*)

`Signals.`**`connect`**(*obj*, *name*, *callback*, *user_arg=None*)

Parameters

- **obj** (*object*) – the object sending a signal
- **name** (*signal name*) – the signal to listen for, typically a string
- **callback** (*function*) – the function to call when that signal is sent
- **user_arg** – optional additional argument to callback, if None no arguments will be added

When a matching signal is sent, callback will be called with all the positional parameters sent with the signal. If user_arg is not None it will be sent added to the end of the positional parameters sent to callback.

`urwid.`**`disconnect_signal`**(*obj*, *name*, *callback*, *user_arg=None*)

`Signals.`**`disconnect`**(*obj*, *name*, *callback*, *user_arg=None*)
This function will remove a callback from the list connected to a signal with connect_signal().

`urwid.`**`register_signal`**(*sig_cls*, *signals*)

`Signals.`**`register`**(*sig_cls*, *signals*)

Parameters

- **sig_class** (*class*) – the class of an object that will be sending signals
- **signals** (*signal names*) – a list of signals that may be sent, typically each signal is represented by a string

This function must be called for a class before connecting any signal callbacks or emiting any signals from that class' objects

`urwid.`**`emit_signal`**(*obj*, *name*, **args*)

`Signals.`**`emit`**(*obj*, *name*, **args*)

Parameters

- **obj** (*object*) – the object sending a signal
- **name** (*signal name*) – the signal to send, typically a string
- ***args** – zero or more positional arguments to pass to the signal callback functions

This function calls each of the callbacks connected to this signal with the args arguments as positional parameters.

This function returns True if any of the callbacks returned True.

4.6 Global Settings

`urwid.`**`set_encoding`**(*encoding*)
Set the byte encoding to assume when processing strings and the encoding to use when converting unicode strings.

`urwid.`**`get_encoding_mode`**()
Get the mode Urwid is using when processing text strings. Returns 'narrow' for 8-bit encodings, 'wide' for CJK encodings or 'utf8' for UTF-8 encodings.

`urwid.`**`supports_unicode`**()
Return True if python is able to convert non-ascii unicode strings to the current encoding.

4.7 Raw Display Attributes

class `urwid.AttrSpec` (*fg*, *bg*, *colors=256*)

fg – a string containing a comma-separated foreground color and settings

Color values: 'default' (use the terminal's default foreground), 'black', 'dark red', 'dark green', 'brown', 'dark blue', 'dark magenta', 'dark cyan', 'light gray', 'dark gray', 'light red', 'light green', 'yellow', 'light blue', 'light magenta', 'light cyan', 'white'

High-color example values: '#009' (0% red, 0% green, 60% red, like HTML colors) '#fcc' (100% red, 80% green, 80% blue) 'g40' (40% gray, decimal), 'g#cc' (80% gray, hex), '#000', 'g0', 'g#00' (black), '#fff', 'g100', 'g#ff' (white) 'h8' (color number 8), 'h255' (color number 255)

Setting: 'bold', 'underline', 'blink', 'standout'

Some terminals use 'bold' for bright colors. Most terminals ignore the 'blink' setting. If the color is not given then 'default' will be assumed.

bg – a string containing the background color

Color values: 'default' (use the terminal's default background), 'black', 'dark red', 'dark green', 'brown', 'dark blue', 'dark magenta', 'dark cyan', 'light gray'

High-color exaples: see fg examples above

An empty string will be treated the same as 'default'.

colors – the maximum colors available for the specification

Valid values include: 1, 16, 88 and 256. High-color values are only usable with 88 or 256 colors. With 1 color only the foreground settings may be used.

```
>>> AttrSpec('dark red', 'light gray', 16)
AttrSpec('dark red', 'light gray')
>>> AttrSpec('yellow, underline, bold', 'dark blue')
AttrSpec('yellow,bold,underline', 'dark blue')
>>> AttrSpec('#ddb', '#004', 256) # closest colors will be found
AttrSpec('#dda', '#006')
>>> AttrSpec('#ddb', '#004', 88)
AttrSpec('#ccc', '#000', colors=88)
```

`background`
Return the background color.

`colors`
Return the maximum colors required for this object.

Returns 256, 88, 16 or 1.

`get_rgb_values` ()
Return (fg_red, fg_green, fg_blue, bg_red, bg_green, bg_blue) color components. Each component is in the range 0-255. Values are taken from the XTerm defaults and may not exactly match the user's terminal.

If the foreground or background is 'default' then all their compenents will be returned as None.

```
>>> AttrSpec('yellow', '#ccf', colors=88).get_rgb_values()
(255, 255, 0, 205, 205, 255)
>>> AttrSpec('default', 'g92').get_rgb_values()
(None, None, None, 238, 238, 238)
```

4.8 Canvas Classes and Functions

4.8.1 Canvas Classes

class `urwid.`**`Canvas`**(*value1=None*, *value2=None*, *value3=None*)

base class for canvases

value1, value2, value3 – if not None, raise a helpful error: the old Canvas class is now called TextCanvas.

`finalize`(*widget*, *size*, *focus*)

Mark this canvas as finalized (should not be any future changes to its content). This is required before caching the canvas. This happens automatically after a widget's 'render call returns the canvas thanks to some metaclass magic.

widget – widget that rendered this canvas size – size parameter passed to widget's render method focus – focus parameter passed to widget's render method

`set_pop_up`(*w*, *left*, *top*, *overlay_width*, *overlay_height*)

This method adds pop-up information to the canvas. This information is intercepted by a PopUpTarget widget higher in the chain to display a pop-up at the given (left, top) position relative to the current canvas.

Parameters

- **w** (*widget*) – widget to use for the pop-up
- **left** (*int*) – x position for left edge of pop-up >= 0
- **top** (*int*) – y position for top edge of pop-up >= 0
- **overlay_width** (*int*) – width of overlay in screen columns > 0
- **overlay_height** (*int*) – height of overlay in screen rows > 0

`text`

Return the text content of the canvas as a list of strings, one for each row.

`translate_coords`(*dx*, *dy*)

Return coords shifted by (dx, dy).

class `urwid.`**`TextCanvas`**(*text=None*, *attr=None*, *cs=None*, *cursor=None*, *maxcol=None*, *check_width=True*)

class for storing rendered text and attributes

text – list of strings, one for each line attr – list of run length encoded attributes for text cs – list of run length encoded character set for text cursor – (x,y) of cursor or None maxcol – screen columns taken by this canvas check_width – check and fix width of all lines in text

`cols`()

Return the screen column width of this canvas.

`content`(*trim_left=0*, *trim_top=0*, *cols=None*, *rows=None*, *attr_map=None*)

Return the canvas content as a list of rows where each row is a list of (attr, cs, text) tuples.

trim_left, trim_top, cols, rows may be set by CompositeCanvas when rendering a partially obscured canvas.

`content_delta`(*other*)

Return the differences between other and this canvas.

If other is the same object as self this will return no differences, otherwise this is the same as calling content().

`rows`()

Return the number of rows in this canvas.

translated_coords (*dx*, *dy*)
Return cursor coords shifted by (dx, dy), or None if there is no cursor.

class urwid.**BlankCanvas**
a canvas with nothing on it, only works as part of a composite canvas since it doesn't know its own size

content (*trim_left*, *trim_top*, *cols*, *rows*, *attr*)
return (cols, rows) of spaces with default attributes.

class urwid.**SolidCanvas** (*fill_char*, *cols*, *rows*)
A canvas filled completely with a single character.

content_delta (*other*)
Return the differences between other and this canvas.

class urwid.**CompositeCanvas** (*canv=None*)
class for storing a combination of canvases

canv – a Canvas object to wrap this CompositeCanvas around.

if canv is a CompositeCanvas, make a copy of its contents

content ()
Return the canvas content as a list of rows where each row is a list of (attr, cs, text) tuples.

content_delta (*other*)
Return the differences between other and this canvas.

fill_attr (*a*)
Apply attribute a to all areas of this canvas with default attribute currently set to None, leaving other attributes intact.

fill_attr_apply (*mapping*)
Apply an attribute-mapping dictionary to the canvas.

mapping – dictionary of original-attribute:new-attribute items

overlay (*other*, *left*, *top*)
Overlay other onto this canvas.

pad_trim_left_right (*left*, *right*)
Pad or trim this canvas on the left and right

values > 0 indicate screen columns to pad values < 0 indicate screen columns to trim

pad_trim_top_bottom (*top*, *bottom*)
Pad or trim this canvas on the top and bottom.

set_depends (*widget_list*)
Explicitly specify the list of widgets that this canvas depends on. If any of these widgets change this canvas will have to be updated.

trim (*top*, *count=None*)
Trim lines from the top and/or bottom of canvas.

top – number of lines to remove from top count – number of lines to keep, or None for all the rest

trim_end (*end*)
Trim lines from the bottom of the canvas.

end – number of lines to remove from the end

4.8.2 CompositeCanvas Builders

`urwid.`**`CanvasCombine`**(*l*)

Stack canvases in l vertically and return resulting canvas.

Parameters l – list of (canvas, position, focus) tuples:

position a value that widget.set_focus will accept or None if not allowed

focus True if this canvas is the one that would be in focus if the whole widget is in focus

`urwid.`**`CanvasJoin`**(*l*)

Join canvases in l horizontally. Return result.

Parameters l – list of (canvas, position, focus, cols) tuples:

position value that widget.set_focus will accept or None if not allowed

focus True if this canvas is the one that would be in focus if the whole widget is in focus

cols is the number of screen columns that this widget will require, if larger than the actual canvas.cols() value then this widget will be padded on the right.

`urwid.`**`CanvasOverlay`**(*top_c*, *bottom_c*, *left*, *top*)

Overlay canvas top_c onto bottom_c at position (left, top).

4.8.3 CanvasCache

class `urwid.`**`CanvasCache`**

Cache for rendered canvases. Automatically populated and accessed by Widget render() MetaClass magic, cleared by Widget._invalidate().

Stores weakrefs to the canvas objects, so an external class must maintain a reference for this cache to be effective. At present the Screen classes store the last topmost canvas after redrawing the screen, keeping the canvases from being garbage collected.

_widgets[widget] = {(wcls, size, focus): weakref.ref(canvas), ...} _refs[weakref.ref(canvas)] = (widget, wcls, size, focus) _deps[widget} = [dependent_widget, ...]

x.__init__(...) initializes x; see help(type(x)) for signature

classmethod **`clear`**()

Empty the cache.

classmethod **`fetch`**(*widget*, *wcls*, *size*, *focus*)

Return the cached canvas or None.

widget – widget object requested wcls – widget class that contains render() function size, focus – render() parameters

classmethod **`invalidate`**(*widget*)

Remove all canvases cached for widget.

classmethod **`store`**(*wcls*, *canvas*)

Store a weakref to canvas in the cache.

wcls – widget class that contains render() function canvas – rendered canvas with widget_info (widget, size, focus)

4.9 Text Layout Classes

class `urwid.`**`TextLayout`**

`layout` (*text*, *width*, *align*, *wrap*)
Return a layout structure for text.

Parameters

- **text** – string in current encoding or unicode string
- **width** – number of screen columns available
- **align** – align mode for text
- **wrap** – wrap mode for text

Layout structure is a list of line layouts, one per output line. Line layouts are lists than may contain the following tuples:

•(column width of text segment, start offset, end offset)

•(number of space characters to insert, offset or None)

•(column width of insert text, offset, "insert text")

The offset in the last two tuples is used to determine the attribute used for the inserted spaces or text respectively. The attribute used will be the same as the attribute at that text offset. If the offset is None when inserting spaces then no attribute will be used.

`supports_align_mode` (*align*)
Return True if align is a supported align mode.

`supports_wrap_mode` (*wrap*)
Return True if wrap is a supported wrap mode.

class `urwid.`**`StandardTextLayout`**

`align_layout` (*text*, *width*, *segs*, *wrap*, *align*)
Convert the layout segs to an aligned layout.

`calculate_text_segments` (*text*, *width*, *wrap*)
Calculate the segments of text to display given width screen columns to display them.

text - unicode text or byte string to display width - number of available screen columns wrap - wrapping mode used

Returns a layout structure without alignment applied.

`layout` (*text*, *width*, *align*, *wrap*)
Return a layout structure for text.

`pack` (*maxcol*, *layout*)
Return a minimal maxcol value that would result in the same number of lines for layout. layout must be a layout structure returned by self.layout().

`supports_align_mode` (*align*)
Return True if align is 'left', 'center' or 'right'.

`supports_wrap_mode` (*wrap*)
Return True if wrap is 'any', 'space' or 'clip'.

4.10 Command Map

class `urwid.`**`CommandMap`**

dict-like object for looking up commands from keystrokes

Default values (key: command):

```
'tab':        'next selectable',
'ctrl n':     'next selectable',
'shift tab':  'prev selectable',
'ctrl p':     'prev selectable',
'ctrl l':     'redraw screen',
'esc':        'menu',
'up':         'cursor up',
'down':       'cursor down',
'left':       'cursor left',
'right':      'cursor right',
'page up':    'cursor page up',
'page down':  'cursor page down',
'home':       'cursor max left',
'end':        'cursor max right',
' ':          'activate',
'enter':      'activate',
```

`copy``()`

Return a new copy of this CommandMap, likely so we can modify it separate from a shared one.

4.11 Constants

Note: These constants may be used, but using the string values themselves in your program is equally supported. These constants are used internally by urwid just to avoid possible misspelling, but the example programs and tutorials tend to use the string values.

4.11.1 Widget Sizing Methods

One or more of these values returned by `Widget.sizing()` to indicate supported sizing methods.

`urwid.`**`FLOW`** **= 'flow'**

Widget that is given a number of columns by its parent widget and calculates the number of rows it requires for rendering e.g. `Text`

`urwid.`**`BOX`** **= 'box'**

Widget that is given a number of columns and rows by its parent widget and must render that size e.g. `ListBox`

`urwid.`**`FIXED`** **= 'fixed'**

Widget that knows the number of columns and rows it requires and will only render at that exact size e.g. `BigText`

4.11.2 Horizontal Alignment

Used to horizontally align text in `Text` widgets and child widgets of `Padding` and `Overlay`.

`urwid.`**`LEFT`** **= 'left'**

`urwid.`**`CENTER`** **= 'center'**

`urwid.`**`RIGHT`** **= 'right'**

4.11.3 Veritcal Alignment

Used to vertically align child widgets of `Filler` and `Overlay`.

`urwid.`**`TOP`** **= 'top'**

`urwid.`**`MIDDLE`** **= 'middle'**

`urwid.`**`BOTTOM`** **= 'bottom'**

4.11.4 Width and Height Settings

Used to distribute or set widths and heights of child widgets of `Padding`, `Filler`, `Columns`, `Pile` and `Overlay`.

`urwid.`**`PACK`** **= 'pack'**
Ask the child widget to calculate the number of columns or rows it needs

`urwid.`**`GIVEN`** **= 'given'**
A set number of columns or rows, e.g. ('given', 10) will have exactly 10 columns or rows given to the child widget

`urwid.`**`RELATIVE`** **= 'relative'**
A percentage of the total space, e.g. ('relative', 50) will give half of the total columns or rows to the child widget

`urwid.`**`RELATIVE_100`** **= ('relative', 100)**

`urwid.`**`WEIGHT`** **= 'weight'**
A weight value for distributing columns or rows, e.g. ('weight', 3) will give 3 times as many columns or rows as another widget in the same container with ('weight', 1).

4.11.5 Text Wrapping Modes

`urwid.`**`SPACE`** **= 'space'**
wrap text on space characters or at the boundaries of wide characters

`urwid.`**`ANY`** **= 'any'**
wrap before any wide or narrow character that would exceed the available screen columns

`urwid.`**`CLIP`** **= 'clip'**
clip before any wide or narrow character that would exceed the available screen columns ad don't display the remaining text on the line

4.11.6 Foreground and Background Colors

Standard background and foreground colors

`urwid.`**`BLACK`** **= 'black'**

`urwid.`**`DARK_RED`** **= 'dark red'**

`urwid.`**`DARK_GREEN`** **= 'dark green'** `urwid.`**`BROWN`** = **‘brown’**

`urwid.`**`DARK_BLUE`** = **‘dark blue’**

`urwid.`**`DARK_MAGENTA`** = **‘dark magenta’**

`urwid.`**`DARK_CYAN`** = **‘dark cyan’**

`urwid.`**`LIGHT_GRAY`** = **‘light gray’**

Standard foreground colors (not safe to use as background)

`urwid.`**`DARK_GRAY`** = **‘dark gray’**

`urwid.`**`LIGHT_RED`** = **‘light red’**

`urwid.`**`LIGHT_GREEN`** = **‘light green’**

`urwid.`**`YELLOW`** = **‘yellow’**

`urwid.`**`LIGHT_BLUE`** = **‘light blue’**

`urwid.`**`LIGHT_MAGENTA`** = **‘light magenta’**

`urwid.`**`LIGHT_CYAN`** = **‘light cyan’**

`urwid.`**`WHITE`** = **‘white’**

User’s terminal configuration default foreground or background

Note: There is no way to tell if the user’s terminal has a light or dark color as their default foreground or background, so it is highly recommended to use this setting for both foreground and background when you do use it.

`urwid.`**`DEFAULT`** = **‘default’**

256 and 88 Color Foregrounds and Backgrounds

Constants are not defined for these colors.

See also:

256-Color Foreground and Background Colors

4.11.7 Signal Names

`urwid.`**`UPDATE_PALETTE_ENTRY`** = **‘update palette entry’**
sent by `BaseScreen` (and subclasses like `raw_display.Screen`) when a palette entry is changed. `MainLoop` handles this signal by redrawing the whole screen.

`urwid.`**`INPUT_DESCRIPTORS_CHANGED`** = **‘input descriptors changed’**
sent by `BaseScreen` (and subclasses like `raw_display.Screen`) when the list of input file descriptors has changed. `MainLoop` handles this signal by updating the file descriptors being watched by its event loop.

4.11.8 Command Names

Command names are used as values in `CommandMap` instances. Widgets look up the command associated with keypresses in their `Widget.keypress()` methods.

You may define any new command names as you wish and look for them in your own widget code. These are the standard ones expected by code included in Urwid.

`urwid.`**`REDRAW_SCREEN`** **= 'redraw screen'**
Default associated keypress: 'ctrl l'

`MainLoop.process_input()` looks for this command to force a screen refresh. This is useful in case the screen becomes corrupted.

`urwid.`**`ACTIVATE`** **= 'activate'**
Default associated keypresses: ' ' (space), 'enter'

Activate a widget such as a `Button`, `CheckBox` or `RadioButton`.

`urwid.`**`CURSOR_UP`** **= 'cursor up'**
Default associated keypress: 'up'

Move the cursor or selection up one row.

`urwid.`**`CURSOR_DOWN`** **= 'cursor down'**
Default associated keypress: 'down'

Move the cursor or selection down one row.

`urwid.`**`CURSOR_LEFT`** **= 'cursor left'**
Default associated keypress: 'left'

Move the cursor or selection left one column.

`urwid.`**`CURSOR_RIGHT`** **= 'cursor right'**
Default associated keypress: 'right'

Move the cursor or selection right one column.

`urwid.`**`CURSOR_PAGE_UP`** **= 'cursor page up'**
Default associated keypress: 'page up'

Move the cursor or selection up one page.

`urwid.`**`CURSOR_PAGE_DOWN`** **= 'cursor page down'**
Default associated keypress: 'page down'

Move the cursor or selection down one page.

`urwid.`**`CURSOR_MAX_LEFT`** **= 'cursor max left'**
Default associated keypress: 'home'

Move the cursor or selection to the leftmost column.

`urwid.`**`CURSOR_MAX_RIGHT`** **= 'cursor max right'**
Default associated keypress: 'end'

Move the cursor or selection to the rightmost column.

4.12 Exceptions

exception `urwid.`**`ExitMainLoop`**
When this exception is raised within a main loop the main loop will exit cleanly.

x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`WidgetError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`ListBoxError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`ColumnsError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`PileError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`GridFlowError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`BoxAdapterError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`OverlayError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`TextError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`EditError`**
x.__init__(...) initializes x; see help(type(x)) for signature

exception `urwid.`**`CanvasError`**
x.__init__(...) initializes x; see help(type(x)) for signature

4.13 Metaclasses

class `urwid.`**`WidgetMeta`**(*name*, *bases*, *d*)
Bases: `MetaSuper`, `MetaSignals`

Automatic caching of render and rows methods.

Class variable *no_cache* is a list of names of methods to not cache automatically. Valid method names for *no_cache* are `'render'` and `'rows'`.

Class variable *ignore_focus* if defined and set to `True` indicates that the canvas this widget renders is not affected by the focus parameter, so it may be ignored when caching.

class `urwid.`**`MetaSuper`**(*name*, *bases*, *d*)
adding .__super

class `urwid.`**`MetaSignals`**(*name*, *bases*, *d*)
register the list of signals in the class varable signals, including signals in superclasses.

4.14 Deprecated Classes

class `urwid.`**`FlowWidget`**
Deprecated. Inherit from Widget and add:

_sizing = frozenset(['flow'])

at the top of your class definition instead.

Base class of widgets that determine their rows from the number of columns available.

x.__init__(...) initializes x; see help(type(x)) for signature

render (*size*, *focus=False*)
All widgets must implement this function.

rows (*size*, *focus=False*)
All flow widgets must implement this function.

class urwid.**BoxWidget**
Deprecated. Inherit from Widget and add:

_sizing = frozenset(['box']) _selectable = True

at the top of your class definition instead.

Base class of width and height constrained widgets such as the top level widget attached to the display object

x.__init__(...) initializes x; see help(type(x)) for signature

render (*size*, *focus=False*)
All widgets must implement this function.

class urwid.**FixedWidget**
Deprecated. Inherit from Widget and add:

_sizing = frozenset(['fixed'])

at the top of your class definition instead.

Base class of widgets that know their width and height and cannot be resized

x.__init__(...) initializes x; see help(type(x)) for signature

pack (*size=None*, *focus=False*)
All fixed widgets must implement this function.

render (*size*, *focus=False*)
All widgets must implement this function.

class urwid.**AttrWrap** (*w*, *attr*, *focus_attr=None*)
w – widget to wrap (stored as self.original_widget) attr – attribute to apply to w focus_attr – attribute to apply when in focus, if None use attr

This widget is a special case of the new AttrMap widget, and it will pass all function calls and variable references to the wrapped widget. This class is maintained for backwards compatibility only, new code should use AttrMap instead.

```
>>> AttrWrap(Divider(u"!"), 'bright')
<AttrWrap flow widget <Divider flow widget '!'> attr='bright'>
>>> AttrWrap(Edit(), 'notfocus', 'focus')
<AttrWrap selectable flow widget <Edit selectable flow widget '' edit_pos=0> attr='notfocus' foc
>>> size = (5,)
>>> aw = AttrWrap(Text(u"hi"), 'greeting', 'fgreet')
>>> aw.render(size, focus=False).content().next()
[('greeting', None, ...'hi   ')]
>>> aw.render(size, focus=True).content().next()
[('fgreet', None, ...'hi   ')]
```

set_attr (*attr*)
Set the attribute to apply to the wrapped widget

>> w = AttrWrap(Divider("-"), None) >> w.set_attr('new_attr') >> w <AttrWrap flow widget <Divider flow widget '-'> attr='new_attr'>

set_focus_attr (*focus_attr*)

Set the attribute to apply to the wapped widget when it is in focus

If None this widget will use the attr instead (no change when in focus).

>> w = AttrWrap(Divider("-"), 'old') >> w.set_focus_attr('new_attr') >> w <AttrWrap flow widget <Divider flow widget '-'> attr='old' focus_attr='new_attr'> >> w.set_focus_attr(None) >> w <AttrWrap flow widget <Divider flow widget '-'> attr='old'>

class urwid.**PollingListWalker** (*contents*)

contents – list to poll for changes

This class is deprecated. Use SimpleFocusListWalker instead.

get_focus ()

Return (focus widget, focus position).

get_next (*start_from*)

Return (widget after start_from, position after start_from).

get_prev (*start_from*)

Return (widget before start_from, position before start_from).

set_focus (*position*)

Set focus position.

Python Module Index