urwidtrees Documentation

Release 1.0.2

Patrick Totzke

March 30, 2016

Contents

This is a Widget Container API for the `urwid` toolkit. It uses a MVC approach and allows to build trees of widgets. Its design goals are

- clear separation classes that define, decorate and display trees of widgets
- representation of trees by local operations on node positions
- easy to use default implementation for simple trees
- Collapses are considered decoration

Generally, tree structures are defined by subclassing *Tree* and overwriting local position movements. For most purposes however, using a *SimpleTree* will do. The choice to define trees by overwriting local position movements allows to easily define potentially infinite tree structures. See *example4* for how to walk local file systems.

Trees of widgets are rendered by `TreeBox` widgets. These are based on urwids `ListBox` widget and display trees such that siblings grow vertically and children horizontally. TreeBoxes handle key presses to move in the tree and collapse/expand subtrees if possible.

CHAPTER 1

Structure

Tree objects define a tree structure by implementing the local movement methods

- *parent_position()*
- *first_child_position()*
- *last_child_position()*
- *next_sibling_position()*
- *prev_sibling_position()*

Each of which takes and returns a *position* object of arbitrary type (fixed for the Tree) as done in urwids ListWalker API. Apart from this, a Tree is assumed to define a dedicated position *tree.root* that is used as fallback initially focussed element, and define the `__getitem__()` method to return its content (usually a Widget) for a given position.

Note that *Tree* only defines a tree structure, it does not necessarily have any decoration around its contained Widgets.

There is a ready made subclass called *SimpleTree* that offers the tree API for a given nested tuple structure. If you write your own classes its a good idea to subclass *Tree* and just overwrite the above mentioned methods as the base class already offers a number of derivative methods.

1.1 API

class `urwidtrees.tree.`**`Tree`**

Base class for a tree strucures that can be displayed by `TreeBox` widgets. An instance defines a structure by defining local transformations on positions. That is, by overwriting

- *next_sibling_position*
- *prev_sibling_position*
- *parent_position*
- *first_child_position*
- *last_child_position*

that compute the next position in the respective direction. Also, they need to implement method *__getitem__* that returns a `Widget` for a given position.

The type of objects used as positions may vary in subclasses and is deliberately unspecified for the base class.

This base class already implements methods based on the local transformations above. These include *depth()*, *last_decendant()* and *[next|prev]_position* that computes next/previous positions in depth-first order.

depth(*pos*)
determine depth of node at pos

first_ancestor(*pos*)
position of pos's ancestor with depth 0. Usually, this should return the root node, but a `Tree` might represent a forrest - have multiple nodes without parent.

first_child_position(*pos*)
returns the position of the first child of the node at *pos*, or *None* if none exists.

first_sibling_position(*pos*)
position of first sibling of pos

is_leaf(*pos*)
checks if given position has no children

last_child_position(*pos*)
returns the position of the last child of the node at *pos*, or *None* if none exists.

last_decendant(*pos*)
position of last (in DFO) decendant of pos

last_sibling_position(*pos*)
position of last sibling of pos

next_position(*pos*)
returns the next position in depth-first order

next_sibling_position(*pos*)
returns the position of the next sibling of the node at *pos*, or *None* if none exists.

parent_position(*pos*)
returns the position of the parent node of the node at *pos* or *None* if none exists.

positions(*reverse=False*)
returns a generator that walks the positions of this tree in DFO

prev_position(*pos*)
returns the previous position in depth-first order

prev_sibling_position(*pos*)
returns the position of the previous sibling of the node at *pos*, or *None* if none exists.

class urwidtrees.tree.**SimpleTree**(*treelist*)
Walks on a given fixed acyclic structure given as a list of nodes; every node is a tuple *(content, children)*, where *content* is a *urwid.Widget* to be displayed at that position and *children* is either *None* or a list of nodes.

Positions are lists of integers determining a path from the root node with position *(0,)*.

depth(*pos*)
more performant implementation due to specific structure of pos

CHAPTER 2

Containers

TreeBox is essentially a urwid.ListBox that displays a given *Tree*. Per default no decoration is used and the widgets of the tree are simply displayed line by line in depth first order. *TreeBox*'s constructor accepts a *focus* parameter to specify the initially focussed position. Internally, it uses a *TreeListWalker* to linearize the tree to a list.

TreeListWalker serve as adapter between *Tree* and urwid.ListWalker APIs: They implement the ListWalker API using the data from a given *Tree* in depth-first order. As such, one can directly pass on a *TreeListWalker* to an urwid.ListBox if one doesn't want to use tree-based focus movement or key bindings for collapsing subtrees.

2.1 API

class urwidtrees.widgets.**TreeBox**(*tree*, *focus=None*)

A widget that displays a given Tree. This is essentially a ListBox with the ability to move the focus based on directions in the Tree and to collapse/expand subtrees if possible.

TreeBox interprets *left/right* as well as *page up/'page down* to move the focus to parent/first child and next/previous sibling respectively. All other keys are passed to the underlying ListBox.

collapse_all()

Collapse all positions; works only if the underlying tree allows it.

collapse_focussed()

Collapse currently focussed position; works only if the underlying tree allows it.

expand_all()

Expand all positions; works only if the underlying tree allows it.

expand_focussed()

Expand currently focussed position; works only if the underlying tree allows it.

focus_first_child()

move focus to first child of currently focussed one

focus_last_child()

move focus to last child of currently focussed one

focus_next()

move focus to next position (DFO)

focus_next_sibling()

move focus to next sibling of currently focussed one

focus_parent()
move focus to parent node of currently focussed one

focus_prev()
move focus to previous position (DFO)

focus_prev_sibling()
move focus to previous sibling of currently focussed one

class urwidtrees.widgets.**TreeListWalker**(*tree*, *focus=None*)
ListWalker to walk through a class:*Tree*.

This translates a `Tree` into a `urwid.ListWalker` that is digestible by `urwid.ListBox`. It uses `Tree.[next|prev]_position` to determine the next/previous position in depth first order.

clear_cache()
removes all cached lines

positions(*reverse=False*)
returns a generator that walks the tree's positions

urwidtrees.widgets.**implementsCollapseAPI**(*tree*)
determines if given tree can collapse positions

urwidtrees.widgets.**implementsDecorateAPI**(*tree*)
determines if given tree offers line decoration

CHAPTER 3

Decoration

Is done by using (subclasses of) *DecoratedTree*. Objects of this type wrap around a given *Tree* and themselves behave like a (possibly altered) tree. Per default, *DecoratedTree* just passes every method on to its underlying tree. Decoration is done *not* by overwriting *__getitem__*, but by offering two additional methods

- *DecoratedTree.get_decorated()*
- *DecoratedTree.decorate()*.

get_decorated(pos) returns the (decorated) content of the original tree at the given position. *decorate(pos, widget,..)* decorates the given widget assuming its placed at a given position. The former is trivially based on the latter, Containers that display *Tree*'s use *get_decorated* instead of `__getitem__()` when working on *DecoratedTree*'s.

The reason for this slightly odd design choice is that first it makes it easy to read the original content of a decorated tree: You simply use *dtree[pos]*. Secondly, this makes it possible to recursively add line decoration when nesting (decorated) Trees.

The module *decoration* offers a few readily usable *DecoratedTree* subclasses that implement decoration by indentation, arrow shapes and subtree collapsing: *CollapsibleTree*, *IndentedTree*, *CollapsibleIndentedTree*, *ArrowTree* and *CollapsibleArrowTree*. Each can be further customized by constructor parameters.

3.1 API

class `urwidtrees.decoration.`**`ArrowTree`**(*walker*, *indent=3*, *childbar_offset=0*, *arrow_hbar_char=u'u2500'*, *arrow_hbar_att=None*, *arrow_vbar_char=u'u2502'*, *arrow_vbar_att=None*, *arrow_tip_char=u'u27a4'*, *arrow_tip_att=None*, *arrow_att=None*, *arrow_connector_tchar=u'u251c'*, *arrow_connector_lchar=u'u2514'*, *arrow_connector_att=None*, ***kwargs*)

Decorates the tree by indenting nodes according to their depth and using the gaps to draw arrows indicate the tree structure.

`decorate`(*pos*, *widget*, *is_first=True*)

builds a list element for given position in the tree. It consists of the original widget taken from the Tree and some decoration columns depending on the existence of parent and sibling positions. The result is a urwid.Columns widget.

class urwidtrees.decoration.**CollapseIconMixin**(*is_collapsed=<function <lambda>>*, *icon_collapsed_char='+'*, *icon_expanded_char='-'*, *icon_collapsed_att=None*, *icon_expanded_att=None*, *icon_frame_left_char='['*, *icon_frame_right_char=']'*, *icon_frame_att=None*, *icon_focussed_att=None*, ***kwargs*)

Mixin for `Tree` that allows to collapse subtrees and use an indicator icon in line decorations. This Mixin adds the ability to construct collapse-icon for a position, indicating its collapse status to `CollapseMixin`.

class urwidtrees.decoration.**CollapseMixin**(*is_collapsed=<function <lambda>>*, ***kwargs*)

Mixin for `Tree` that allows to collapse subtrees.

This works by overwriting `[first|last]_child_position`, forcing them to return *None* if the given position is considered collapsed. We use a (given) callable *is_collapsed* that accepts positions and returns a boolean to determine which node is considered collapsed.

is_collapsed(*pos*)

checks if given position is currently collapsed

class urwidtrees.decoration.**CollapsibleArrowTree**(*treelistwalker*, *icon_offset=0*, *indent=5*, ***kwargs*)

Arrow-decoration that allows collapsing subtrees

class urwidtrees.decoration.**CollapsibleIndentedTree**(*walker*, *icon_offset=1*, *indent=4*, ***kwargs*)

Indent collapsible tree nodes according to their depth in the tree and display icons indicating collapse-status in the gaps.

decorate(*pos*, *widget*, *is_first=True*)

builds a list element for given position in the tree. It consists of the original widget taken from the Tree and some decoration columns depending on the existence of parent and sibling positions. The result is a urwid.Culumns widget.

class urwidtrees.decoration.**CollapsibleTree**(*tree*, ***kwargs*)

Undecorated Tree that allows to collapse subtrees

class urwidtrees.decoration.**DecoratedTree**(*content*)

`Tree` that wraps around another `Tree` and allows to read original content as well as decorated versions thereof.

decorate(*pos*, *widget*, *is_first=True*)

decorate *widget* according to a position *pos* in the original tree. setting *is_first* to False indicates that we are decorating a line that is *part* of the (multi-line) content at this position, but not the first part. This allows to omit incoming arrow heads for example.

get_decorated(*pos*)

return widget that consists of the content of original tree at given position plus its decoration.

class urwidtrees.decoration.**IndentedTree**(*tree*, *indent=2*)

Indent tree nodes according to their depth in the tree

CHAPTER 4

Examples

4.1 Minimal example

Simplest example rendering:

```
[-] item 1
        sub item 1
        sub item 2
    item 2
```

```
import urwid
import urwidtrees


tree_widget = urwidtrees.widgets.TreeBox(
    urwidtrees.decoration.CollapsibleIndentedTree(
        urwidtrees.tree.SimpleTree([
            (urwid.SelectableIcon('item 1'), (
                (urwid.SelectableIcon('sub item 1'), None),
                (urwid.SelectableIcon('sub item 2'), None),
            )),
            (urwid.SelectableIcon('item 2'), None),
        ])
    )
)

urwid.MainLoop(tree_widget).run()
```

4.2 Basic use

```
#!/usr/bin/python
# Copyright (C) 2013  Patrick Totzke <patricktotzke@gmail.com>
# This file is released under the GNU GPL, version 3 or a later revision.

import urwid
from urwidtrees.tree import SimpleTree
from urwidtrees.widgets import TreeBox


# define some colours
```

```
palette = [
    ('body', 'black', 'light gray'),
    ('focus', 'light gray', 'dark blue', 'standout'),
    ('bars', 'dark blue', 'light gray', ''),
    ('arrowtip', 'light blue', 'light gray', ''),
    ('connectors', 'light red', 'light gray', ''),
]

# We use selectable Text widgets for our example..


class FocusableText(urwid.WidgetWrap):
    """Selectable Text used for nodes in our example"""
    def __init__(self, txt):
        t = urwid.Text(txt)
        w = urwid.AttrMap(t, 'body', 'focus')
        urwid.WidgetWrap.__init__(self, w)

    def selectable(self):
        return True

    def keypress(self, size, key):
        return key

# define a test tree in the format accepted by SimpleTree. Essentially, a
# tree is given as (nodewidget, [list, of, subtrees]). SimpleTree accepts
# lists of such trees.


def construct_example_simpletree_structure(selectable_nodes=True, children=3):

    Text = FocusableText if selectable_nodes else urwid.Text

    # define root node
    tree = (Text('ROOT'), [])

    # define some children
    c = g = gg = 0  # counter
    for i in range(children):
        subtree = (Text('Child %d' % c), [])
        # and grandchildren..
        for j in range(children):
            subsubtree = (Text('Grandchild %d' % g), [])
            for k in range(children):
                leaf = (Text('Grand Grandchild %d' % gg), None)
                subsubtree[1].append(leaf)
                gg += 1  # inc grand-grandchild counter
            subtree[1].append(subsubtree)
            g += 1  # inc grandchild counter
        tree[1].append(subtree)
        c += 1
    return tree


def construct_example_tree(selectable_nodes=True, children=2):
    # define a list of tree structures to be passed on to SimpleTree
    forrest = [construct_example_simpletree_structure(selectable_nodes,
                                                       children)]
```

```
69
70      # stick out test tree into a SimpleTree and return
71      return SimpleTree(forrest)
72
73  def unhandled_input(k):
74      #exit on q
75      if k in ['q', 'Q']: raise urwid.ExitMainLoop()
76
77  if __name__ == "__main__":
78      # get example tree
79      stree = construct_example_tree()
80
81      # put the tree into a treebox
82      treebox = TreeBox(stree)
83
84      # add some decoration
85      rootwidget = urwid.AttrMap(treebox, 'body')
86      #add a text footer
87      footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
88      #enclose all in a frame
89      urwid.MainLoop(urwid.Frame(rootwidget, footer=footer), palette, unhandled_input = unhandled_input
```

4.3 Decoration

```
1   #!/usr/bin/python
2   # Copyright (C) 2013  Patrick Totzke <patricktotzke@gmail.com>
3   # This file is released under the GNU GPL, version 3 or a later revision.
4
5   from example1 import construct_example_tree, palette, unhandled_input # example data
6   from urwidtrees.decoration import ArrowTree  # for Decoration
7   from urwidtrees.widgets import TreeBox
8   import urwid
9
10  if __name__ == "__main__":
11      # get example tree
12      stree = construct_example_tree()
13      # Here, we add some decoration by wrapping the tree using ArrowTree.
14      atree = ArrowTree(stree,
15                        # customize at will..
16                        # arrow_hbar_char=u'\u2550',
17                        # arrow_vbar_char=u'\u2551',
18                        # arrow_tip_char=u'\u25B7',
19                        # arrow_connector_tchar=u'\u2560',
20                        # arrow_connector_lchar=u'\u255A',
21                        )
22
23      # put the into a treebox
24      treebox = TreeBox(atree)
25      rootwidget = urwid.AttrMap(treebox, 'body')
26      #add a text footer
27      footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
28      #enclose in a frame
29      urwid.MainLoop(urwid.Frame(rootwidget, footer=footer), palette, unhandled_input = unhandled_input
```

4.4 Collapsible subtrees

```
#!/usr/bin/python
# Copyright (C) 2013  Patrick Totzke <patricktotzke@gmail.com>
# This file is released under the GNU GPL, version 3 or a later revision.

from example1 import construct_example_tree, palette, unhandled_input # example data
from urwidtrees.decoration import CollapsibleIndentedTree  # for Decoration
from urwidtrees.widgets import TreeBox
import urwid

if __name__ == "__main__":
    # get some SimpleTree
    stree = construct_example_tree()

    # Use (subclasses of) the wrapper decoration.CollapsibleTree to construct a
    # tree where collapsible subtrees. Apart from the original tree, these take
    # a callable `is_collapsed` that defines initial collapsed-status if a
    # given position.

    # We want all grandchildren collapsed initially
    if_grandchild = lambda pos: stree.depth(pos) > 1

    # We use CollapsibleIndentedTree around the original example tree.
    # This uses Indentation to indicate the tree structure and squeezes in
    # text-icons to indicate the collapsed status.
    # Also try CollapsibleTree or CollapsibleArrowTree..
    tree = CollapsibleIndentedTree(stree,
                                   is_collapsed=if_grandchild,
                                   icon_focussed_att='focus',
                                   # indent=6,
                                   # childbar_offset=1,
                                   # icon_frame_left_char=None,
                                   # icon_frame_right_char=None,
                                   # icon_expanded_char='-',
                                   # icon_collapsed_char='+',
                                   )

    # put the tree into a treebox
    treebox = TreeBox(tree)
    rootwidget = urwid.AttrMap(treebox, 'body')
    #add a text footer
    footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
    #enclose all in a frame
    urwid.MainLoop(urwid.Frame(rootwidget, footer=footer), palette, unhandled_input = unhandled_input
```

4.5 Custom Trees: Walking the filesystem

```
#!/usr/bin/python
# Copyright (C) 2013  Patrick Totzke <patricktotzke@gmail.com>
# This file is released under the GNU GPL, version 3 or a later revision.

import urwid
import os
```

```
from example1 import palette, unhandled_input  # example data
from urwidtrees.widgets import TreeBox
from urwidtrees.tree import Tree
from urwidtrees.decoration import CollapsibleArrowTree


# define selectable urwid.Text widgets to display paths
class FocusableText(urwid.WidgetWrap):
    """Widget to display paths lines"""
    def __init__(self, txt):
        t = urwid.Text(txt)
        w = urwid.AttrMap(t, 'body', 'focus')
        urwid.WidgetWrap.__init__(self, w)

    def selectable(self):
        return True

    def keypress(self, size, key):
        return key

# define Tree that can walk your filesystem


class DirectoryTree(Tree):
    """
    A custom Tree representing our filesystem structure.
    This implementation is rather inefficient: basically every position-lookup
    will call `os.listdir`.. This makes navigation in the tree quite slow.
    In real life you'd want to do some caching.

    As positions we use absolute path strings.
    """
    # determine dir separator and form of root node
    pathsep = os.path.sep
    drive, _ = os.path.splitdrive(pathsep)

    # define root node This is part of the Tree API!
    root = drive + pathsep

    def __getitem__(self, pos):
        return FocusableText(pos)

    # generic helper
    def _list_dir(self, path):
        """returns absolute paths for all entries in a directory"""
        try:
            elements = [os.path.join(
                path, x) for x in os.listdir(path) if os.path.isdir(path)]
            elements.sort()
        except OSError:
            elements = None
        return elements

    def _get_siblings(self, pos):
        """lists the parent directory of pos """
        parent = self.parent_position(pos)
        siblings = [pos]
        if parent is not None:
```

```
            siblings = self._list_dir(parent)
        return siblings

    # Tree API
    def parent_position(self, pos):
        parent = None
        if pos != '/':
            parent = os.path.split(pos)[0]
        return parent

    def first_child_position(self, pos):
        candidate = None
        if os.path.isdir(pos):
            children = self._list_dir(pos)
            if children:
                candidate = children[0]
        return candidate

    def last_child_position(self, pos):
        candidate = None
        if os.path.isdir(pos):
            children = self._list_dir(pos)
            if children:
                candidate = children[-1]
        return candidate

    def next_sibling_position(self, pos):
        candidate = None
        siblings = self._get_siblings(pos)
        myindex = siblings.index(pos)
        if myindex + 1 < len(siblings):  # pos is not the last entry
            candidate = siblings[myindex + 1]
        return candidate

    def prev_sibling_position(self, pos):
        candidate = None
        siblings = self._get_siblings(pos)
        myindex = siblings.index(pos)
        if myindex > 0:  # pos is not the first entry
            candidate = siblings[myindex - 1]
        return candidate


if __name__ == "__main__":
    cwd = os.getcwd()  # get current working directory
    dtree = DirectoryTree()  # get a directory walker

    # Use CollapsibleArrowTree for decoration.
    # define initial collapse:
    as_deep_as_cwd = lambda pos: dtree.depth(pos) >= dtree.depth(cwd)

    # We hide the usual arrow tip and use a customized collapse-icon.
    decorated_tree = CollapsibleArrowTree(dtree,
                                          is_collapsed=as_deep_as_cwd,
                                          arrow_tip_char=None,
                                          icon_frame_left_char=None,
                                          icon_frame_right_char=None,
                                          icon_collapsed_char=u'\u25B6',
```

```
                                        icon_expanded_char=u'\u25B7',)

    # stick it into a TreeBox and use 'body' color attribute for gaps
    tb = TreeBox(decorated_tree, focus=cwd)
    root_widget = urwid.AttrMap(tb, 'body')
    #add a text footer
    footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
    #enclose all in a frame
    urwid.MainLoop(urwid.Frame(root_widget, footer=footer), palette, unhandled_input = unhandled_inpu
```

4.6 Nesting Trees

```
#!/usr/bin/python
# Copyright (C) 2013  Patrick Totzke <patricktotzke@gmail.com>
# This file is released under the GNU GPL, version 3 or a later revision.

from example1 import palette, construct_example_tree  # example data
from example1 import FocusableText, unhandled_input  # Selectable Text used for nodes
from urwidtrees.widgets import TreeBox
from urwidtrees.tree import SimpleTree
from urwidtrees.nested import NestedTree
from urwidtrees.decoration import ArrowTree, CollapsibleArrowTree  # decoration
import urwid
import logging

if __name__ == "__main__":
    #logging.basicConfig(filename='example.log',level=logging.DEBUG)
    # Take some Arrow decorated Tree that we later stick inside another tree.
    innertree = ArrowTree(construct_example_tree())
    # Some collapsible, arrow decorated tree with extra indent
    anotherinnertree = CollapsibleArrowTree(construct_example_tree(),
                                            indent=10)

    # A SimpleTree, that contains the two above
    middletree = SimpleTree(
        [
            (FocusableText('Middle ROOT'),
             [
                 (FocusableText('Mid Child One'), None),
                 (FocusableText('Mid Child Two'), None),
                 (innertree, None),
                 (FocusableText('Mid Child Three'),
                  [
                      (FocusableText('Mid Grandchild One'), None),
                      (FocusableText('Mid Grandchild Two'), None),
                  ]
                  ),
                 (anotherinnertree,
                  # middletree defines a childnode here. This is usually
                  # covered by the tree 'anotherinnertree', unless the
                  # interepreting NestedTree's constructor gets parameter
                  # interpret_covered=True..
                  [
                      (FocusableText('XXX I\'m invisible!'), None),

                  ]),
```

```
            ]
            )
        ]
    )  # end SimpleTree constructor for middletree
    # use customized arrow decoration for middle tree
    middletree = ArrowTree(middletree,
                           arrow_hbar_char=u'\u2550',
                           arrow_vbar_char=u'\u2551',
                           arrow_tip_char=u'\u25B7',
                           arrow_connector_tchar=u'\u2560',
                           arrow_connector_lchar=u'\u255A')

    # define outmost tree
    outertree = SimpleTree(
        [
            (FocusableText('Outer ROOT'),
             [
                 (FocusableText('Child One'), None),
                 (middletree, None),
                 (FocusableText('last outer child'), None),
             ]
             )
        ]
    )  # end SimpleTree constructor

    # add some Arrow decoration
    outertree = ArrowTree(outertree)
    # wrap the whole thing into a Nested Tree
    outertree = NestedTree(outertree,
                           # show covered nodes like  XXX
                           interpret_covered=False
                           )

    # put it into a treebox and run
    treebox = TreeBox(outertree)
    rootwidget = urwid.AttrMap(treebox, 'body')
    #add a text footer
    footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
    #enclose all in a frame
    urwid.MainLoop(urwid.Frame(rootwidget, footer=footer), palette, unhandled_input = unhandled_input
```

4.7 Dynamic List

Update the tree after it's initially build.

Shows something like:

```
root
-PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
|  64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.039 ms
|
-64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.053 ms
|
-64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.064 ms
```

```
import subprocess
import urwid
import urwidtrees

root_node = [urwid.Text('root'), None]
tree_widget = urwidtrees.widgets.TreeBox(
    urwidtrees.decoration.ArrowTree(
        urwidtrees.tree.SimpleTree([root_node])
    )
)

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

loop = urwid.MainLoop(tree_widget,
    unhandled_input=exit_on_q)


def on_stdout(data):
    if not root_node[1]:
        root_node[1] = []
    root_node[1].append((urwid.Text(data), None))
    tree_widget.refresh()


proc = subprocess.Popen(
    ['ping', '127.0.0.1'],
    stdout=loop.watch_pipe(on_stdout),
    close_fds=True)

loop.run()
proc.kill()
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

u

T

U