
urwidtrees Documentation

Release 1.0.3.dev

Patrick Totzke

Oct 01, 2017

Contents

1	Structure	3
2	Containers	5
3	Decoration	7
4	Examples	9
5	Indices and tables	19
	Python Module Index	21

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 urwid's `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.

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 urwid's 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.

API

class `urwidtrees.tree.Tree`

Base class for a tree structures 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_descendant()` 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 forest - 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_descendant (*pos*)

position of last (in DFO) descendant 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,).

static depth (*pos*)

more performant implementation due to specific structure of *pos*

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.

API

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

A widget that displays a given *Tree*. This is essentially a `ListWalker` 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 `ListWalker`.

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

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.

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.Columns` 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

```
exception urwidtrees.decoration.NoSpaceError
```

too little space for requested decoration

Minimal example

Simplest example rendering:

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

```
1 import urwid
2 import urwidtrees
3
4
5 tree_widget = urwidtrees.widgets.TreeBox(
6     urwidtrees.decoration.CollapsibleIndentedTree(
7         urwidtrees.tree.SimpleTree([
8             (urwid.SelectableIcon('item 1'), (
9                 (urwid.SelectableIcon('sub item 1'), None),
10                (urwid.SelectableIcon('sub item 2'), None),
11            )),
12            (urwid.SelectableIcon('item 2'), None),
13        ])
14     )
15 )
16
17 urwid.MainLoop(tree_widget).run()
```

Basic use

```

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  import urwid
6  from urwidtrees.tree import SimpleTree
7  from urwidtrees.widgets import TreeBox
8
9
10 # define some colours
11 palette = [
12     ('body', 'black', 'light gray'),
13     ('focus', 'light gray', 'dark blue', 'standout'),
14     ('bars', 'dark blue', 'light gray', ''),
15     ('arrowtip', 'light blue', 'light gray', ''),
16     ('connectors', 'light red', 'light gray', ''),
17 ]
18
19 # We use selectable Text widgets for our example..
20
21
22 class FocusableText(urwid.WidgetWrap):
23     """Selectable Text used for nodes in our example"""
24     def __init__(self, txt):
25         t = urwid.Text(txt)
26         w = urwid.AttrMap(t, 'body', 'focus')
27         urwid.WidgetWrap.__init__(self, w)
28
29     def selectable(self):
30         return True
31
32     def keypress(self, size, key):
33         return key
34
35 # define a test tree in the format accepted by SimpleTree. Essentially, a
36 # tree is given as (nodewidget, [list, of, subtrees]). SimpleTree accepts
37 # lists of such trees.
38
39
40 def construct_example_simpletree_structure(selectable_nodes=True, children=3):
41
42     Text = FocusableText if selectable_nodes else urwid.Text
43
44     # define root node
45     tree = (Text('ROOT'), [])
46
47     # define some children
48     c = g = gg = 0 # counter
49     for i in range(children):
50         subtree = (Text('Child {0:d}'.format(c)), [])
51         # and grandchildren..
52         for j in range(children):
53             subsubtree = (Text('Grandchild {0:d}'.format(g)), [])
54             for k in range(children):
55                 leaf = (Text('Grand Grandchild {0:d}'.format(gg)), None)
56                 subsubtree[1].append(leaf)

```

```

57         gg += 1 # inc grand-grandchild counter
58         subtree[1].append(subsubtree)
59         g += 1 # inc grandchild counter
60         tree[1].append(subtree)
61         c += 1
62     return tree
63
64
65 def construct_example_tree(selectable_nodes=True, children=2):
66     # define a list of tree structures to be passed on to SimpleTree
67     forrest = [construct_example_simpletree_structure(selectable_nodes,
68                                                       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).run() # go

```

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).run() # go

```

Collapsible subtrees

```

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 CollapsibleIndentedTree # for Decoration
7  from urwidtrees.widgets import TreeBox
8  import urwid
9
10 if __name__ == "__main__":
11     # get some SimpleTree
12     stree = construct_example_tree()
13
14     # Use (subclasses of) the wrapper decoration.CollapsibleTree to construct a
15     # tree where collapsible subtrees. Apart from the original tree, these take
16     # a callable `is_collapsed` that defines initial collapsed-status if a
17     # given position.
18
19     # We want all grandchildren collapsed initially
20     if_grandchild = lambda pos: stree.depth(pos) > 1
21
22     # We use CollapsibleIndentedTree around the original example tree.
23     # This uses Indentation to indicate the tree structure and squeezes in
24     # text-icons to indicate the collapsed status.
25     # Also try CollapsibleTree or CollapsibleArrowTree..
26     tree = CollapsibleIndentedTree(stree,
27                                   is_collapsed=if_grandchild,
28                                   icon_focussed_att='focus',
29                                   # indent=6,
30                                   # childbar_offset=1,
31                                   # icon_frame_left_char=None,
32                                   # icon_frame_right_char=None,
33                                   # icon_expanded_char='-',
34                                   # icon_collapsed_char='+',
35                                   )
36
37     # put the tree into a treebox
38     treebox = TreeBox(tree)
39     rootwidget = urwid.AttrMap(treebox, 'body')
40     #add a text footer

```



```

41 footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
42 #enclose all in a frame
43 urwid.MainLoop(urwid.Frame(rootwidget, footer=footer), palette, unhandled_input =
↳ unhandled_input).run() # go
44

```

Custom Trees: Walking the filesystem

```

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  import urwid
6  import os
7  from example1 import palette, unhandled_input # example data
8  from urwidtrees.widgets import TreeBox
9  from urwidtrees.tree import Tree
10 from urwidtrees.decoration import CollapsibleArrowTree
11
12
13 # define selectable urwid.Text widgets to display paths
14 class FocusableText(urwid.WidgetWrap):
15     """Widget to display paths lines"""
16     def __init__(self, txt):
17         t = urwid.Text(txt)
18         w = urwid.AttrMap(t, 'body', 'focus')
19         urwid.WidgetWrap.__init__(self, w)
20
21     def selectable(self):
22         return True
23
24     def keypress(self, size, key):
25         return key
26
27 # define Tree that can walk your filesystem
28
29
30 class DirectoryTree(Tree):
31     """
32     A custom Tree representing our filesystem structure.
33     This implementation is rather inefficient: basically every position-lookup
34     will call `os.listdir`. This makes navigation in the tree quite slow.
35     In real life you'd want to do some caching.
36
37     As positions we use absolute path strings.
38     """
39     # determine dir separator and form of root node
40     pathsep = os.path.sep
41     drive, _ = os.path.splitdrive(pathsep)
42
43     # define root node This is part of the Tree API!
44     root = drive + pathsep
45
46     def __getitem__(self, pos):
47         return FocusableText(pos)

```

```
48
49 # generic helper
50 def _list_dir(self, path):
51     """returns absolute paths for all entries in a directory"""
52     try:
53         elements = [os.path.join(
54             path, x) for x in os.listdir(path) if os.path.isdir(path)]
55         elements.sort()
56     except OSError:
57         elements = None
58     return elements
59
60 def _get_siblings(self, pos):
61     """lists the parent directory of pos """
62     parent = self.parent_position(pos)
63     siblings = [pos]
64     if parent is not None:
65         siblings = self._list_dir(parent)
66     return siblings
67
68 # Tree API
69 def parent_position(self, pos):
70     parent = None
71     if pos != '/':
72         parent = os.path.split(pos)[0]
73     return parent
74
75 def first_child_position(self, pos):
76     candidate = None
77     if os.path.isdir(pos):
78         children = self._list_dir(pos)
79         if children:
80             candidate = children[0]
81     return candidate
82
83 def last_child_position(self, pos):
84     candidate = None
85     if os.path.isdir(pos):
86         children = self._list_dir(pos)
87         if children:
88             candidate = children[-1]
89     return candidate
90
91 def next_sibling_position(self, pos):
92     candidate = None
93     siblings = self._get_siblings(pos)
94     myindex = siblings.index(pos)
95     if myindex + 1 < len(siblings): # pos is not the last entry
96         candidate = siblings[myindex + 1]
97     return candidate
98
99 def prev_sibling_position(self, pos):
100     candidate = None
101     siblings = self._get_siblings(pos)
102     myindex = siblings.index(pos)
103     if myindex > 0: # pos is not the first entry
104         candidate = siblings[myindex - 1]
105     return candidate
```

```

106
107
108 if __name__ == "__main__":
109     cwd = os.getcwd() # get current working directory
110     dtree = DirectoryTree() # get a directory walker
111
112     # Use CollapsibleArrowTree for decoration.
113     # define initial collapse:
114     as_deep_as_cwd = lambda pos: dtree.depth(pos) >= dtree.depth(cwd)
115
116     # We hide the usual arrow tip and use a customized collapse-icon.
117     decorated_tree = CollapsibleArrowTree(dtree,
118                                           is_collapsed=as_deep_as_cwd,
119                                           arrow_tip_char=None,
120                                           icon_frame_left_char=None,
121                                           icon_frame_right_char=None,
122                                           icon_collapsed_char=u'\u25B6',
123                                           icon_expanded_char=u'\u25B7',)
124
125     # stick it into a TreeBox and use 'body' color attribute for gaps
126     tb = TreeBox(decorated_tree, focus=cwd)
127     root_widget = urwid.AttrMap(tb, 'body')
128     #add a text footer
129     footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
130     #enclose all in a frame
131     urwid.MainLoop(urwid.Frame(root_widget, footer=footer), palette, unhandled_input_
↳= unhandled_input).run() # go

```

Nesting Trees

```

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 palette, construct_example_tree # example data
6  from example1 import FocusableText, unhandled_input # Selectable Text used for nodes
7  from urwidtrees.widgets import TreeBox
8  from urwidtrees.tree import SimpleTree
9  from urwidtrees.nested import NestedTree
10 from urwidtrees.decoration import ArrowTree, CollapsibleArrowTree # decoration
11 import urwid
12 import logging
13
14 if __name__ == "__main__":
15     #logging.basicConfig(filename='example.log',level=logging.DEBUG)
16     # Take some Arrow decorated Tree that we later stick inside another tree.
17     innertree = ArrowTree(construct_example_tree())
18     # Some collapsible, arrow decorated tree with extra indent
19     anotherinnertree = CollapsibleArrowTree(construct_example_tree(),
20                                           indent=10)
21
22     # A SimpleTree, that contains the two above
23     middletree = SimpleTree(
24         [
25             (FocusableText('Middle ROOT'),

```

```

26     [
27         (FocusableText('Mid Child One'), None),
28         (FocusableText('Mid Child Two'), None),
29         (innertree, None),
30         (FocusableText('Mid Child Three'),
31         [
32             (FocusableText('Mid Grandchild One'), None),
33             (FocusableText('Mid Grandchild Two'), None),
34         ]
35         ),
36         (anotherinnertree,
37         # middletree defines a childnode here. This is usually
38         # covered by the tree 'anotherinnertree', unless the
39         # interpreting NestedTree's constructor gets parameter
40         # interpret_covered=True..
41         [
42             (FocusableText('XXX I\'m invisible!'), None),
43
44         ]),
45     ]
46 )
47 ]
48 ) # end SimpleTree constructor for middletree
49 # use customized arrow decoration for middle tree
50 middletree = ArrowTree(middletree,
51                        arrow_hbar_char=u'\u2550',
52                        arrow_vbar_char=u'\u2551',
53                        arrow_tip_char=u'\u25B7',
54                        arrow_connector_tchar=u'\u2560',
55                        arrow_connector_lchar=u'\u255A')
56
57 # define outmost tree
58 outertree = SimpleTree(
59     [
60         (FocusableText('Outer ROOT'),
61         [
62             (FocusableText('Child One'), None),
63             (middletree, None),
64             (FocusableText('last outer child'), None),
65         ]
66         )
67     ]
68 ) # end SimpleTree constructor
69
70 # add some Arrow decoration
71 outertree = ArrowTree(outertree)
72 # wrap the whole thing into a Nested Tree
73 outertree = NestedTree(outertree,
74                        # show covered nodes like XXX
75                        interpret_covered=False
76                        )
77
78 # put it into a treebox and run
79 treebox = TreeBox(outertree)
80 rootwidget = urwid.AttrMap(treebox, 'body')
81 #add a text footer
82 footer = urwid.AttrMap(urwid.Text('Q to quit'), 'focus')
83 #enclose all in a frame

```

```

84  urwid.MainLoop(urwid.Frame(rootwidget, footer=footer), palette, unhandled_input =
↳ unhandled_input).run() # go

```

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

```

```

1  import subprocess
2  import urwid
3  import urwidtrees
4
5  root_node = [urwid.Text('root'), None]
6  tree_widget = urwidtrees.widgets.TreeBox(
7      urwidtrees.decoration.ArrowTree(
8          urwidtrees.tree.SimpleTree([root_node])
9      )
10 )
11
12 def exit_on_q(key):
13     if key in ['q', 'Q']:
14         raise urwid.ExitMainLoop()
15
16 loop = urwid.MainLoop(tree_widget,
17     unhandled_input=exit_on_q)
18
19
20 def on_stdout(data):
21     if not root_node[1]:
22         root_node[1] = []
23     root_node[1].append((urwid.Text(data), None))
24     tree_widget.refresh()
25
26
27 proc = subprocess.Popen(
28     ['ping', '127.0.0.1'],
29     stdout=loop.watch_pipe(on_stdout),
30     close_fds=True)
31
32 loop.run()
33 proc.kill()

```


CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`

U

urwidtrees, ??
urwidtrees.decoration, 7
urwidtrees.tree, 3
urwidtrees.widgets, 5

A

ArrowTree (class in urwidtrees.decoration), 7

C

clear_cache() (urwidtrees.widgets.TreeListWalker method), 6

collapse_all() (urwidtrees.widgets.TreeBox method), 5

collapse_focussed() (urwidtrees.widgets.TreeBox method), 5

CollapseIconMixin (class in urwidtrees.decoration), 8

CollapseMixin (class in urwidtrees.decoration), 8

CollapsibleArrowTree (class in urwidtrees.decoration), 8

CollapsibleIndentedTree (class in urwidtrees.decoration), 8

CollapsibleTree (class in urwidtrees.decoration), 8

D

decorate() (urwidtrees.decoration.ArrowTree method), 7

decorate() (urwidtrees.decoration.CollapsibleIndentedTree method), 8

decorate() (urwidtrees.decoration.DecoratedTree method), 8

DecoratedTree (class in urwidtrees.decoration), 8

depth() (urwidtrees.tree.SimpleTree static method), 4

depth() (urwidtrees.tree.Tree method), 4

E

expand_all() (urwidtrees.widgets.TreeBox method), 5

expand_focussed() (urwidtrees.widgets.TreeBox method), 5

F

first_ancestor() (urwidtrees.tree.Tree method), 4

first_child_position() (urwidtrees.tree.Tree method), 4

first_sibling_position() (urwidtrees.tree.Tree method), 4

focus_first_child() (urwidtrees.widgets.TreeBox method), 5

focus_last_child() (urwidtrees.widgets.TreeBox method), 5

focus_next() (urwidtrees.widgets.TreeBox method), 6

focus_next_sibling() (urwidtrees.widgets.TreeBox method), 6

focus_parent() (urwidtrees.widgets.TreeBox method), 6

focus_prev() (urwidtrees.widgets.TreeBox method), 6

focus_prev_sibling() (urwidtrees.widgets.TreeBox method), 6

G

get_decorated() (urwidtrees.decoration.DecoratedTree method), 8

I

IndentedTree (class in urwidtrees.decoration), 8

is_collapsed() (urwidtrees.decoration.CollapseMixin method), 8

is_leaf() (urwidtrees.tree.Tree method), 4

L

last_child_position() (urwidtrees.tree.Tree method), 4

last_decendant() (urwidtrees.tree.Tree method), 4

last_sibling_position() (urwidtrees.tree.Tree method), 4

N

next_position() (urwidtrees.tree.Tree method), 4

next_sibling_position() (urwidtrees.tree.Tree method), 4

NoSpaceError, 8

P

parent_position() (urwidtrees.tree.Tree method), 4

positions() (urwidtrees.tree.Tree method), 4

positions() (urwidtrees.widgets.TreeListWalker method), 6

prev_position() (urwidtrees.tree.Tree method), 4

prev_sibling_position() (urwidtrees.tree.Tree method), 4

S

SimpleTree (class in urwidtrees.tree), 4

T

Tree (class in urwidtrees.tree), 3

TreeBox (class in urwidtrees.widgets), 5

TreeListWalker (class in urwidtrees.widgets), 6

U

urwidtrees (module), 1

urwidtrees.decoration (module), 7

urwidtrees.tree (module), 3

urwidtrees.widgets (module), 5