

# ScrolledList: A Tkinter scrollable list widget



John W. Shipman

2009-10-10 16:50

## Abstract

Describes a graphical user interface widget that combines a list box and scroll bars, for the Python programming language's Tkinter widget set.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to **tcc-doc@nmt.edu**.

## Table of Contents

1. Introduction .....	2
2. Layout of the ScrolledList widget .....	2
3. Using a ScrolledList widget in your Tkinter application .....	3
3.1. Public attributes of a ScrolledList .....	3
3.2. Methods on a ScrolledList .....	4
4. scrolledlisttest: A small test driver .....	4
4.1. The Application class .....	5
4.2. Application.__init__(): Constructor .....	5
4.3. Application.__createWidgets(): Widget layout .....	6
4.4. Application.__pickHandler(): Handler for clicking on the listbox .....	6
4.5. Application.__tests__(): Run initial tests .....	6
4.6. Main program .....	7
5. The ScrolledList module .....	8
5.1. Module prologue .....	8
5.2. class ScrolledList .....	8
5.3. ScrolledList.__init__(): Constructor .....	9
5.4. ScrolledList.__createWidgets(): Lay out internal widgets .....	9
5.5. ScrolledList.__clickHandler(): Event handler for button 1 .....	11
5.6. ScrolledList.count(): Return the line count .....	12
5.7. ScrolledList.__getitem__(): Implement the Python index operator .....	12
5.8. ScrolledList.append(): Add a line of text .....	13
5.9. ScrolledList.insert(): Insert a line of text .....	13
5.10. ScrolledList.delete(): Remove a line of text .....	13
5.11. ScrolledList.clear(): Empty the listbox .....	13

<sup>1</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/>

<sup>2</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/scrolledlist.pdf>

# 1. Introduction

---

This document describes the usage and implementation of a graphical user interface widget called a `ScrolledList`, whose purpose is to display a list of lines of text. This widget combines a *listbox* with one or more *scrollbars*, so the user can use the scrollbars to see all the text even if it doesn't all fit in the available screen space.

This widget works with the Tkinter graphical user interface for the Python programming language.

The document also contains the actual code of the widget, explained in a lightweight literate programming style, along with a small test driver script named `scrolledlisttest`.

Relevant documents:

- *Python 2.2 quick reference*<sup>3</sup> describes the Python language.
- *Tkinter reference: a GUI for Python*<sup>4</sup> describes the Tkinter widget set.
- See the author's *Lightweight Literate Programming* page<sup>5</sup> for an explanation of the technique of embedding the program in its internal documentation.
- See also the author's page on the *Cleanroom software methodology*<sup>6</sup> for a discussion of the author's preferred style of implementation.

Files generated from this document are available online:

- `scrolledlist.py`<sup>7</sup>, the module defining the `ScrolledList` widget.
- `scrolledlisttest`<sup>8</sup>, a small test driver.
- `scrolledlist.xml`<sup>9</sup>, the DocBook source file for this document.

## 2. Layout of the `ScrolledList` widget

---

The parts of a `ScrolledList` widget visible to the user are:

- A Tkinter `Listbox` widget that displays the lines of text.
- An optional vertical `Scrollbar` widget that allows the user to scroll up and down through the lines of text.
- An optional horizontal `Scrollbar` that allows the user to shift the text in the `Listbox` to the left or right when the lines are too long to fit in the window.

Here is a diagram showing the layout of these components in the most general case, with both vertical and horizontal scrollbars:

---

<sup>3</sup> <http://www.nmt.edu/tcc/help/pubs/python22/>

<sup>4</sup> <http://www.nmt.edu/tcc/help/pubs/tkinter/>

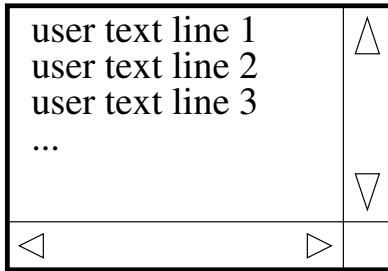
<sup>5</sup> <http://www.nmt.edu/~shipman/soft/litprog/>

<sup>6</sup> <http://www.nmt.edu/~shipman/soft/clean/>

<sup>7</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/scrolledlist.py>

<sup>8</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/scrolledlisttest>

<sup>9</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/scrolledlist.xml>



## 3. Using a ScrolledList widget in your Tkinter application

Here is how you call the constructor for a ScrolledList widget.

```
s = ScrolledList ( master, width=W, height=H, vscroll=VS,  
                  hscroll=HS, callback=c )
```

where:

**s**

The constructor returns a new ScrolledList widget that has *not* been registered. Be sure to register it with the `.grid()` method or it will not appear in your application.

**master**

The parent Frame widget in which the new ScrolledList widget is to be mastered.

**width=W**

The width of the Listbox in characters. The default value is 40.

**height=H**

The height of the Listbox in lines. The default value is 25.

**vscroll=VS**

By default, you will get a vertical scrollbar. If you don't want one, use "vscroll=0".

**hscroll=HS**

By default, you will *not* get a horizontal scrollbar. If you *do* want one, use "hscroll=1".

**callback=c**

If you want your application to react whenever a user clicks on a line in the Listbox, use this keyword argument to supply a function `C`, and that function will be called whenever the user clicks on a line. You should define your function like this:

```
def c ( lineNo ):  
    ...
```

where the `lineNo` argument will be the index (starting at 0) of the line on which the user clicked. For example, if the user clicks on the third line, the ScrolledList widget will call your procedure with an argument 2.

### 3.1. Public attributes of a ScrolledList

These attributes of a ScrolledList widget are visible.

**.width**

The width of the listbox in characters when the widget was constructed.

### **.height**

The height of the listbox in lines when the widget was constructed.

### **.listbox**

The Tkinter `Listbox` widget inside the `ScrolledList` widget. If you want to change the appearance of the listbox, use this attribute's `.configure()` method.

For example, if you have a font object (`tkFont.Font`) named `ttFont`, and you would like to apply it to a `ScrolledList` widget named `optionBox`, this would do it:

```
optionBox.listbox.configure(font=ttFont)
```

## **3.2. Methods on a ScrolledList**

These methods are defined on a `ScrolledList` widget.

### **.count()**

Returns the number of lines currently contained in the listbox.

### **.\_\_getitem\_\_(self, i)**

To retrieve the text of the  $i^{\text{th}}$  line in the listbox, you can use the regular Python index operator. For example, if you have a `ScrolledList` object named `optionBox`, the expression “`optionBox[2]`” would return the contents of the third line.

### **.append(s)**

Adds a string `s` as the next line in the listbox.

### **.insert(linex, s)**

Inserts a string `s` as a new line before position `linex`. For example, to add a new first line containing “Merlin” to a `ScrolledList` widget named `s`:

```
s.insert(0, 'Merlin')
```

### **.delete(linex)**

Deletes a line from the listbox; `linex` is the index of the line, starting from 0.

### **.clear()**

Removes all lines from the listbox.

## **4. scrolledlisttest: A small test driver**

---

Here is a small, complete Tkinter application that exercises the functions of the `ScrolledList` widget. This is what it looks like:



We start out with the usual prologue: a comment pointing back at this documentation. We import the Tkinter module, then we import the module under test: ScrolledList.

scrolledlisttest

```
#!/usr/bin/env python
#=====
# scrolledlisttest: Test driver for ScrolledList
#
# For documentation, see:
#   http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/
#-----
from Tkinter import *
import scrolledlist
```

## 4.1. The Application class

Next we start declaring the Application class, which embodies the entire graphical user interface.

scrolledlisttest

```
class Application(Frame):
    """GUI for scrolledlisttest
    """
```

## 4.2. Application.\_\_init\_\_(): Constructor

The constructor is quite pro forma: call the parent class constructor, register the application with its .grid() method, create the widgets, and run some initial tests.

scrolledlisttest

```
def __init__ ( self ):
    """Constructor for the Application class.
    """
    Frame.__init__ ( self, None )
    self.grid()
```

```
self.__createWidgets()
self.__tests()
```

### 4.3. Application. `__createWidgets()`: Widget layout

There are only two widgets: the ScrolledList widget we are testing, and a quit button below it.

scrolledlisttest

```
def __createWidgets ( self ) :
    """Lay out the widgets.
    """
    self.sbox = scrolledlist.ScrolledList ( self,
        width=20, height=10, hscroll=1,
        callback=self.__pickHandler )
    self.sbox.grid ( row=0, column=0 )

    self.quitButton = Button ( self, text="Quit",
        command=self.quit )
    self.quitButton.grid ( row=1, column=0, columnspan=99,
        sticky=E+W, ipadx=5, ipady=5 )
```

### 4.4. Application. `__pickHandler()`: Handler for clicking on the listbox

This method is the callback function that gets called whenever the user clicks on a line in the listbox. It displays the line number and the text on that line.

scrolledlisttest

```
def __pickHandler ( self, linex ) :
    """Handler for user clicks on lines in the listbox.
    """
    print "Click on line %d, '%s'" % (linex, self.sbox[linex])
```

### 4.5. Application. `__tests__()`: Run initial tests

This method adds some lines to the listbox using both the `.append()` and `.insert()` methods, exercises the `.delete()` method, and then tests the `.count()` method.

scrolledlisttest

```
def __tests ( self ) :
    """Initial testing of the ScrolledList widget.
    """
```

First we insure that the initial list length is zero. Then we add three lines and again check the length.

scrolledlisttest

```
print "Initial size is", self.sbox.count()
print "Add alpaca, buffalo, eagle:"
self.sbox.append ( "alpaca" )
self.sbox.append ( "buffalo" )
self.sbox.append ( "eagle" )
print "Size is now", self.sbox.count()
```

Next we test the `.clear()` method. The size after clearing should be zero.

scrolledlisttest

```
print "Clear listbox:"
self.sbox.clear()
print "Size is now", self.sbox.count()
```

We add the same three lines back in, then test insertion.

scrolledlisttest

```
print "Add alpaca, buffalo, eagle:"
self.sbox.append ( "alpaca" )
self.sbox.append ( "buffalo" )
self.sbox.append ( "eagle" )
print "Insert cachalot"
self.sbox.insert ( 2, "cachalot" )
print "Size is now", self.sbox.count()
```

Next we test deletion.

scrolledlisttest

```
print "Delete buffalo:"
self.sbox.delete ( 1 )
print "Size is now", self.sbox.count()
```

Finally, we insert enough lines, and one long line, to make it possible to test the scrollbars.

scrolledlisttest

```
print "Insert bunches o stuff"
self.sbox.append ( "finch" )
self.sbox.append ( "goshawk" )
self.sbox.append ( "harrier" )
self.sbox.append ( "indigobird" )
self.sbox.append ( "jabiru" )
self.sbox.append ( "kingfisher" )
self.sbox.append ( "Middendorff's grasshopper-warbler" )
self.sbox.append ( "merlin" )
self.sbox.append ( "northern flicker" )
self.sbox.append ( "ovenbird" )
self.sbox.append ( "parula" )
print "Size is now", self.sbox.count()
```

## 4.6. Main program

This is the usual main program for a Tkinter application: instantiate an `Application` object, set up the application's title in the window frame, and wait for events.

scrolledlisttest

```
#=====
# Main program
#-----

app = Application()
app.master.title ( "scrolledlisttest" )
app.mainloop()
```

## 5. The ScrolledList module

---

Here is the `scrolledlist.py` module that defines the `ScrolledList` widget.

### 5.1. Module prologue

The module starts with a comment pointing back to this documentation.

`scrolledlist.py`

```
"""scrolledlist.py: A Tkinter widget combining a Listbox with Scrollbar(s).

For details, see:
    http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/
"""
```

First we import the Tkinter module into our namespace.

`scrolledlist.py`

```
#=====
# Imports
#-----

from Tkinter import *
```

Next, we define two constants for the default `height` and `width` arguments to the widget's constructor.

`scrolledlist.py`

```
#=====
# Manifest constants
#-----

DEFAULT_WIDTH    = "40"
DEFAULT_HEIGHT   = "25"
```

### 5.2. class ScrolledList

Here we start the actual class declaration for `ScrolledList`.

`scrolledlist.py`

```
class ScrolledList(Frame):
    """A compound widget containing a listbox and up to two scrollbars.
```

Inside the class's documentation string, we document the public and internal attributes. The scrollbar widgets are technically public, in case anyone wants to configure their attributes.

`scrolledlist.py`

```
State/invariants:
    .listbox:      [ The Listbox widget ]
    .vScrollbar:
        [ if self has a vertical scrollbar ->
          that scrollbar
```

```

        else -> None ]
    .hScrollbar:
        [ if self has a vertical scrollbar ->
          that scrollbar
          else -> None ]
    .callback:    [ as passed to constructor ]
    .vscroll:     [ as passed to constructor ]
    .hscroll:     [ as passed to constructor ]
    """

```

### 5.3. ScrolledList.\_\_init\_\_(): Constructor

The constructor defines default values for all the keyword arguments. Note that the vertical scrollbar is on by default, while the horizontal scrollbar is off by default.

scrolledlist.py

```

def __init__ ( self, master=None, width=DEFAULT_WIDTH,
              height=DEFAULT_HEIGHT, vscroll=1, hscroll=0, callback=None ):
    """Constructor for ScrolledList.
    """

```

The constructor's first job is to call the constructor for its parent class, Frame.

scrolledlist.py

```

#-- 1 --
# [ self := a new Frame widget child of master ]
Frame.__init__ ( self, master )

```

Next, we store the various constructor arguments inside the instance.

scrolledlist.py

```

#-- 2 --
self.width      = width
self.height     = height
self.vscroll    = vscroll
self.hscroll    = hscroll
self.callback   = callback

```

Finally, we lay out the internal widgets.

scrolledlist.py

```

#-- 3 --
# [ self := self with all widgets created and registered ]
self.__createWidgets()

```

### 5.4. ScrolledList.\_\_createWidgets(): Lay out internal widgets

This method creates and grids all our internal widgets.

scrolledlist.py

```

def __createWidgets ( self ):
    """Lay out internal widgets.
    """

```

Here is the grid plan for our internal widgets:

	0	1
0	.listbox	.vScrollbar
1	.hScrollbar	

First, we create the vertical scrollbar, if there is one. The `sticky=N+S` attribute makes the scrollbar stretch to the full height of grid row 0.

scrolledlist.py

```
#-- 1 --
# [ if self.vscroll ->
#     self := self with a vertical Scrollbar widget added
#     self.vScrollbar := that widget ]
# else -> I ]
if self.vscroll:
    self.vScrollbar = Scrollbar ( self, orient=VERTICAL )
    self.vScrollbar.grid ( row=0, column=1, sticky=N+S )
```

Next, we create the horizontal scrollbar, if there is one. The `sticky=E+W` attribute makes it stretch to the width of grid column 0.

scrolledlist.py

```
#-- 2 --
# [ if self.hscroll ->
#     self := self with a horizontal Scrollbar widget added
#     self.hScrollbar := that widget
# else -> I ]
if self.hscroll:
    self.hScrollbar = Scrollbar ( self, orient=HORIZONTAL )
    self.hScrollbar.grid ( row=1, column=0, sticky=E+W )
```

Now we create the `Listbox` widget. The `relief=SUNKEN` attribute makes the listbox's contents look like they are recessed into the window. The `borderwidth=2` attribute puts a 2-pixel border around the listbox.

scrolledlist.py

```
#-- 3 --
# [ self := self with a Listbox widget added
#     self.listbox := that widget ]
self.listbox = Listbox ( self, relief=SUNKEN,
                        width=self.width, height=self.height,
                        borderwidth=2 )
self.listbox.grid ( row=0, column=0 )
```

The next step is to create the linkages between the scrollbars and the `Listbox`. The `command` attribute of a vertical `Scrollbar` widget is a method that is called whenever the scrollbar is scrolled by the user; the `yview` method of a `Listbox` widget causes its contents to be repositioned. This linkage allows the scrollbar to move the listbox.

However, the linkage is bidirectional. There are operations on the `Listbox` widget that change its contents' position, and when that happens, the scrollbar's position should also be adjusted. This linkage sets the `yscrollcommand` attribute of the `Listbox` to the `.set()` method of the scrollbar.

The linkages are similar for a horizontal scrollbar.

```

#-- 4 --
# [ if self.vscroll ->
#     self.listbox := self.listbox linked so that
#         self.vScrollbar can reposition it ]
#     self.vScrollbar := self.vScrollbar linked so that
#         self.listbox can reposition it
# else -> I ]
if self.vscroll:
    self.listbox["yscrollcommand"] = self.vScrollbar.set
    self.vScrollbar["command"] = self.listbox.yview

#-- 5 --
# [ if self.hscroll ->
#     self.listbox := self.listbox linked so that
#         self.hScrollbar can reposition it ]
#     self.hScrollbar := self.hScrollbar linked so that
#         self.listbox can reposition it
# else -> I ]
if self.hscroll:
    self.listbox["xscrollcommand"] = self.hScrollbar.set
    self.hScrollbar["command"] = self.listbox.xview

```

So that our widget will respond to the user clicking on a line in the listbox, we use the `.bind()` method to set up an event binding that will call our `.__clickHandler` method when that happens.

```

#-- 6 --
# [ self.listbox := self.listbox with an event handler
#     for button-1 clicks that causes self.callback
#     to be called if there is one ]
self.listbox.bind ( "<Button-1>", self.__clickHandler )

```

## 5.5. ScrolledList.\_\_clickHandler(): Event handler for button 1

This method is called when the user clicks mouse button 1 (usually the left button, but left-handers can set it up to be the right-hand button). If the user has provided a `callback` procedure, we will figure out which line of the listbox the user clicked on, and pass the number of that line to that callback.

```

def __clickHandler ( self, event ):
    """Called when the user clicks on a line in the listbox.
    """

```

If there is no callback, don't do anything.

```

#-- 1 --
if not self.callback:
    return

```

The `event` argument holds the screen y-coordinate of the mouse click in its `.y` attribute. The `.nearest()` method on the `Listbox` widget converts this coordinate into a line number.

```
#-- 2 --
# [ call self.callback(c) where c is the line index
#   corresponding to event.y ]
lineNo = self.listbox.nearest ( event.y )
self.callback ( lineNo )
```

So that the listbox will respond to the *PageUp* and *PageDown* keys, we move the keyboard focus to the listbox whenever a family is selected.

```
#-- 3 --
self.listbox.focus_set()
```

## 5.6. ScrolledList.count(): Return the line count

This method returns the number of lines currently in use inside the listbox. The `.size()` method on the `Listbox` widget is exactly what we need.

### Note

Originally I wanted to define a `.__len__()` method so that the user could use the Python `len()` function on a `ScrolledList` to get the line count. However, this caused some bizarre bugs, so it is now a conventional method.

```
def count ( self ):
    """Return the number of lines in use in the listbox.
    """
    return self.listbox.size()
```

## 5.7. ScrolledList.\_\_getitem\_\_(): Implement the Python index operator

This method is called when the user indexes a `ScrolledList` widget to get the text from a specific line of the listbox. The `.get()` method on the `Listbox` widget does just what we need. We raise an `IndexError` exception if the index is out of range.

```
def __getitem__ ( self, k ):
    """Get the (k)th line from the listbox.
    """

    #-- 1 --
    if ( 0 <= k < self.count() ):
        return self.listbox.get ( k )
    else:
        raise IndexError, ( "ScrolledList[%d] out of range." % k )
```

## 5.8. ScrolledList.append(): Add a line of text

This method appends a new last line to the text in the listbox. The constant END specifies a position just after the last existing line.

scrolledlist.py

```
def append ( self, text ):  
    """Append a line to the listbox.  
    """  
    self.listbox.insert ( END, text )
```

## 5.9. ScrolledList.insert(): Insert a line of text

This method inserts a line of text before the line at the specified position. If the position is out of range, we'll just place it at the end.

scrolledlist.py

```
def insert ( self, linex, text ):  
    """Insert a line between two existing lines.  
    """  
  
    #-- 1 --  
    if 0 <= linex < self.count():  
        where = linex  
    else:  
        where = END  
  
    #-- 2 --  
    self.listbox.insert ( where, text )
```

## 5.10. ScrolledList.delete(): Remove a line of text

This method removes a specified line from the listbox. If the given position is out of range, we do nothing.

scrolledlist.py

```
def delete ( self, linex ):  
    """Delete a line from the listbox.  
    """  
    if 0 <= linex < self.count():  
        self.listbox.delete ( linex )
```

## 5.11. ScrolledList.clear(): Empty the listbox

Removes all lines from the listbox. The .delete() method on a Listbox object takes two arguments, a start position and an end position. A zero value refers to the start of the listbox, and the constant END refers to the position just after the last line.

```
def clear ( self ):  
    """Remove all lines.  
    """  
    self.listbox.delete ( 0, END )
```