Examining the PyGtk Toolkit

PyGtk brings the benefits of a high-level programming language to Gtk+ developers, and gives Python programmers access to a modern, high-performance GUI toolkit.

By

Mitch Chapman and Brian Kelley

, Dr. Dobb's Journal
Apr 01, 2000
URL:http://www.ddj.com/architect/184404069

Mitch and Brian are researchers at Bioreason. They can be contacted at chapman@bioreason.com and kelley @bioreason.com, respectively.

Gtk+ is a rich GUI toolkit with excellent interactive performance. It includes a variety of widgets, such as multicolumn lists and hierarchical trees, which are not found in older toolkits such as Tcl/Tk. However, Gtk+ is written in C, making it a relatively poor choice for rapid prototyping -- especially for programmers accustomed to the object-oriented facilities and automatic memory management of more modern languages.

Python, on the other hand, is a powerful, object-oriented programming language. Its reference-counted memory management, versatile built-in types, and strong run-time type checking make it suitable for everything from rapid prototyping to full-blown GUI applications. Python's default GUI toolkit, Tkinter, provides an interface to the Tcl/Tk widget set. While it's certainly powerful, many Python developers have nonetheless complained about the relatively low performance and aging widget set of Tcl/Tk.

The PyGtk package addresses both of these problems. Developed by James Henstridge (http://www.daa.com.au/~james/ pygtk/), PyGtk provides a Python interface to Gtk+. It brings the benefits of a high-level programming language to Gtk+ developers, and gives Python programmers access to a modern, high-performance GUI toolkit. The result is a great environment for rapid development of GUI-based applications.

However, most of the effort in developing Gtk+ and PyGtk has gone to the code itself. While improving rapidly, the documentation still has many gaps. There are few published examples of how to perform common tasks such as creating modal dialogs, controlling *GtkCList*s, and the like.

At Bioreason, we use PyGtk extensively. Having learned the hard way how to take advantage of the toolkit, we'll share some techniques that you'll find useful as you move to PyGtk from Python's default Tkinter UI toolkit.

More specifically, we'll demonstrate several common PyGtk coding techniques and describe relatively unfamiliar widgets such as GtkCTree. All of the examples have been tested under Linux 2.0.36, using Python 1.5.2 (http://www.python .org/), Gtk+ 1.2.3, and PyGtk 0.6.2.

Shutting Down an Application

One of the first surprises in PyGtk programming is that, by default, an application doesn't exit when you close its main window.

When users close a window via the window manager, the *GtkWindow* widget emits a *delete_event*. By default, the event is ignored, but you can use the widget's *connect()* method to request a callback whenever a *delete_event* occurs. Your callback can cause application cleanup, then cause the application to exit by calling PyGtk's *mainquit()* function; see Listing One.

Aligning Widgets

Widget alignment in PyGtk can be as simple as in Tkinter -- if you use *GtkTable* to manage the layout. However, you need to be careful about the order in which you make widgets visible and add them to tables.

You can allocate the number of rows and columns within a *GtkTable*, or let it grow as needed to accommodate added children. If you choose the latter approach, you may find yourself cursing at the inconsistent geometry management of *GtkTable*.

Listing Two shows one way to add labeled entries to a *GtkTable*. When class *UI* is instantiated, it creates a new *AppWindow* and adds a *GtkVBox* as its child. The top of the *GtkVBox* will hold the table of form entries, and the bottom will contain a button for adding new entries at run time. As each entry is added, it is appended to the UI's entry instance variable.

Method *UI.addEntry()* adds a new labeled entry to the *AppWindow*. It begins by figuring out which row of the table will contain the entry. Then it creates a *GtkLabel*, setting the horizontal alignment to 1.0 (that is, to the right edge of the label's allocated space) and the vertical alignment to 0.5 (centered). The label extends from column 0 to column 1 of the table. Next, a *GtkEntry* is added, extending from column 1 to column 2 of the same row. The *GtkEntry* is added to the list of entries contained in the UI instance.

Unfortunately, this implementation has a problem. If you run the program, you'll see a window with three entries. But if you click the Add Row button, you'll see a new entry, labeled Row 3, which doesn't have the correct vertical spacing. Click Add Row again, and you'll find that subsequent entries are added with the correct spacing; see Figure 1. This appears to be a bug in Gtk+. Listing Three is a workaround: After adding each new entry row, explicitly set the spacing for that row.

The listings work around one other geometry-management idiosyncrasy of Gtk+: If you invoke the *show()* method of a *GtkWindow* before showing its children, it will not properly adjust its size to accommodate its children. That's why the *UI.mainloop()* method was added. It ensures that the application's main window becomes visible at the last possible moment, after the *show()* method has been invoked on its immediate children.

If you ever find yourself needing to add children to a *GtkWindow* after invoking its *show()* method, you can force it to recalculate its geometry, by invoking its *check_resize()* method.

Controlling Spinbutton Limits

Spinbuttons come in handy in applications such as database editors, where users want to browse through a set of available records. But in these applications, spinbuttons must be able to dynamically adjust their limits as the number of records in the database changes.

GtkAdjustment widgets control the operating ranges of spinbuttons, scrollbars, text widgets, progress bars, and a host of other widgets. You can change the limits for a spinbutton, scrollbar, or scale widget by asking it for its adjustment widget, then manipulating the settings of the adjustment.

PyGtk 0.6.2 has a few bugs that make it difficult to change an adjustment in place. Fortunately, PyGtk does let you change the ranges for spinbuttons and the like, by creating new *GtkAdjustment*s and associating them with the widgets. You can even couple spinbuttons, scrollbars, and scales together by associating them with the same *GtkAdjustment* instance.

Listing Four shows one way to dynamically adjust the ranges of a spinbutton, scrollbar, and scale. The script defines a UI containing these elements, and sets all three elements to use the same *GtkAdjustment* (with a default upper limit of 10.0). Thanks to the *connect()* call in *UI.__init__()*, users can change the upper limit of all three widgets by typing a new value into the spinbutton's entry field and hitting return.

The *connect()* call causes *UI.spinEntryActivateCB()* to be invoked whenever the user hits return in the spinbutton's entry field. *spinEntryActivateCB()* starts by getting the current text value from the field and trying to convert it to a number. (The *get_text()* method is used instead of *get_value()* because the latter method clips the user's input to the current spinbutton bounds before converting it to a number.)

UI.setUpperBound() checks to ensure the new upper bound is greater than the current lower bound; by default, spinbuttons have a lower bound of zero.

It should be possible to get at the spinbutton's *GtkAdjustment* widget by calling its *get_adjustment()* method. But, as noted, *GtkAdjustment.get_adjustment()* contains a bug, so *setUpperBound()* gets at the adjustment the hard way, using code from the *_gtk* module underlying PyGtk's *gtk* module.

Next, *setUpperBound()* constructs a new *GtkAdjustment* for the spinbutton to use. The new adjustment will be identical to the current adjustment, but it has a different upper limit (namely, the value provided by users).

By default, a new *GtkSpinButton* has a climb rate, or *step_increment*, of 1. But its associated *GtkAdjustment* has a *step_increment* of 0. So when creating the new *GtkAdjustment*, *setUpperBound()* takes care that the *step_increment*, *page_increment*, and *page_size* are greater than zero. Without this precaution, users would be faced with spinbuttons that never change their value, no matter how many times the up and down arrows are clicked.

Finally, *setUpperBound()* assigns the new *GtkAdjustment* instance to each of the UI widgets. Because they all refer to the same adjustment, when one of the widgets changes values, the others change in lockstep with it.

Working with *GtkCTree*s

GtkCTree is one of the most useful and poorly documented of the Gtk+ widgets. *GtkCTree* is inherited from a *GtkCList* and shares all of the capabilities of the *CGtkList*. The main difference is that the first column of a *CTree* expands into a tree view; see Figure 2. A *CTree* is initialized exactly as a *CList* with the desired number of columns and list of column titles. Because the tree can be expanded by users and it is difficult to determine the full extent of the widget, a *GtkCTree* should be embedded in a *GtkScrolledWindow* widget.

Tree nodes can be inserted and removed dynamically, see Example 1(a). If a parent is given, the node is added as a child of the parent node. A value of *None* adds the node to the top level of the tree. Insertion requires that the value of all columns be specified. Each column must have a text value. However, pixmaps can also be specified. For example, insertion arguments *pixmap_closed*, *pixmap_opened*, *mask_closed*, and *mask_opened* can be used to specify the opened and closed pixmaps and masks for the node. For an easy way of creating *pixmap*s, see the function *gtk.create_pixmap_from_xpm()*. The parameter *is_leaf=1* appears to prevent children to be added to the node, and the parameter *expanded=1* expands the node's children by default.

Inserting nodes dynamically requires some bookkeeping. If a node has no children, the expansion image (the plus or minus sign) is not displayed. For all intents and purposes, the node cannot be expanded. If a node's children are to be loaded dynamically, a bookkeeping child should be included in the tree so the node can be expanded. Once the children are loaded, this node can be removed. To assist this process, you can set the data of a node using the function *GtkCree.node_ set_row_data(node, data),* which is useful when using the *tree-expand* signal. Setting the data to an integer or string yields a hashable value for looking up the expanded node in a Python dictionary. Available electronically (see "Resource Center," page 5), you'll find a simple example of a file browser using this technique.

Each column in a *CTree* has a *GtkButton* widget that displays the column title. Pop-up menus can be bound to these menus to allow sorting or other operations. The *GtkButton* is available as the parent of the column widget and can be obtained as *GtkCTree.get_column_widget(column_index)["parent"]*. The process for actually determining this step required us to look at the PyGtk source and use Python interactively. Unfortunately, in this respect, the PyGtk hierarchy is not the same as the Gtk hierarchy, and it is in areas like this where the available documentation is lacking.

Like *GtkCList*, *GtkCTree* only displays a fixed number of columns. This can be annoying if information is dynamically loaded because rebuilding the *CTree* can be time consuming. Fortunately, just about everything in Gtk can be hidden, including the displayed columns using the *GtkCTree* method *set_column_visibility*. However, using features like these sometimes causes slowdowns when redrawing the tree. A workaround is to freeze the tree, perform the necessary operations, then thaw the tree. This prevents drawing every step along the way. Additionally, wrapping the freeze in a *try* block, in which the operations are performed, and thawing the tree in the final clause lets errors be caught and guarantees the screen will repaint; see Example 1(b). The full source to a *CTreeInsert* widget (where the titles are bound to pop-up menus and columns can be added dynamically) is available electronically.

Defining Modal Dialogs

In *Gtk+/GNOME Application Development* (accessible online at http://developer.gnome .org/doc/GGAD/ggad.html), Havoc Pennington cautions against overuse of modal dialogs. He notes that many users find them annoying. Nevertheless, there are times when you do need to implement modal dialogs. And developers familiar with writing modal dialogs in Python/Tkinter will find enough differences in the Gtk+ API to make the process confusing.

Until recently, the hard part of developing a Gtk+ modal dialog was discovering the right way to make it modal. The Gtk+ FAQ recommends using *grab_add()* and *grab_remove()* to route all user inputs to a modal dialog, but the API included a *gtk_window_set_modal()* function. Pennington recommends using *gtk_window_set _modal()*; because his book is likely to become the standard Gtk+ reference, perhaps the documentation conflict will be resolved soon.

After figuring out how to route all user inputs to the dialog, you still need a modal event loop. PyGtk provides a simple, if unintuitive, solution -- just invoke *mainloop()* recursively, and terminate the modal loop by invoking *mainquit()*; see Example 2.

A few more steps are needed to complete a PyGtk modal dialog. For example, you need to call *set_transient_for()* on the dialog to ensure that it remains stacked in front of the window that launched it. Otherwise, users could raise the window in front of the dialog, obscuring it and creating the appearance that your application is hung.

Finally, if you plan on reusing your dialog instances instead of creating a new dialog every time one is needed, then you need to protect the dialog against destruction, just as if it were a top-level window. If you don't supply your own handler for the dialog's *delete_event*, the widgets inside the dialog will be destroyed the first time users dismiss it. Thereafter, users will be faced with an empty window and a series of Gtk-CRITICAL error messages each time the dialog is re-displayed.

A *ModalDialog* class that does all of these basic steps is available electronically. In typical Python fashion, the module includes a *main()* function that is invoked when the module is run as a standalone program. The mainline serves both as an example of how to use the *ModalDialog* class, and as a simple unit test to verify that *ModalDialog* behaves correctly.

Troubleshooting

Say your PyGtk application isn't behaving as expected. You've checked the Gtk+ FAQ, scanned the PyGtk and Gtk+ mailing list archives, and still have no idea what the problem is. What should you do? We've found a couple of useful approaches.

One strategy is to use the source code, together with the online Gtk+ reference pages. So, if you're trying to learn how to work with a new widget class, start with the gtk.py source file. By looking at the class definitions, you can gain a rough idea of what the widget can do.

All too often, however, the method and parameter names by themselves fail to tell what a method may do; so your next stop should be the online Gtk+ reference. Increasingly, it provides good summaries of the corresponding Gtk+ functions and their parameter lists.

However, the Gtk+ documents are chasing a moving target. Often, they simply list the available functions and provide no description of their intent thus, it's a good idea to keep the Gtk+ source tree lying around. Once you know the name of the function in which you're interested and the widget class to which it applies, you can navigate directly to the implementation of the function to see how it works.

The main directory of the PyGtk distribution contains a module called "description.py." This module contains no executable code; instead, it provides stubbed definitions for Gtk+ classes such as *GtkCTreeNode* and *GdkColor*. These classes don't appear in gtk.py; they are implemented completely in ANSI C in gtkmodule.c, and usually appear only as the return values of other PyGtk methods. When you find yourself with a reference to one of these objects, description.py gives you some idea of what you can do with it.

But, in many cases, the quickest way to find out what a function does is to invoke it. After all, Python is an interpreted language, and you can usually throw together an exploratory program in just a few minutes. For example, we wrote short test programs to figure out what *GtkCTree*'s *get_column_widget()* was doing, by traversing the widget hierarchy it returned and printing out widget classes along the way.

DDJ

Listing One

```
#!/usr/bin/env python
"""AppWindow.py:
This module provides an application window --
a GtkWindow that knows when to quit."""
import gtk
class AppWindow(gtk.GtkWindow):
    """This is just a GtkWindow which exits its event loop when destroyed."""
    def __init__(self, title=None):
        """Initialize a new instance.
        If specified, `title' will be displayed in the window's title bar."""
        gtk.GtkWindow.__init__(self)
        if title:
            self.set_title(title)
        self.connect("delete_event", self.quit)
    def quit(self, *args):
        """Exit the application -- or at least exit its event loop."""
        gtk.mainquit()
def main():
    """Module mainline (for standalone execution)"""
    win = AppWindow(title="Sample App Window")
    label = gtk.GtkLabel("Hello, world.")
    win.add(label)
    label.show()
    win.show()
    gtk.mainloop()
if __name__ == "__main__":
    main()
```

Back to Article

Listing Two

```
#!/usr/bin/env python
"""This module demonstates techniques for aligning labels in a form."""

import gtk
import AppWindow
class UI:
    """Builds a form consisting of several rows of labelled entries."""
    def __init__(self):
        """Initialize a new instance."""
        self.top = AppWindow.AppWindow(title="Alignment Demo")
        self.box = gtk.GtkVBox()
        self.top.add(self.box)
        self.box.show()
        self.table = gtk.GtkTable(cols=2)
        self.box.pack_start(self.table)
        self.table.show()
        self.addBtn = gtk.GtkButton("Add Row")
        self.addBtn.connect("clicked", self.addBtnCB)
        self.box.pack_end(self.addBtn, fill=gtk.FALSE, expand=gtk.FALSE)
        self.addBtn.show()

        # Keep a list of the entries added.
        self.entry = []
    def mainloop(self):
        """Show the window and run the main event loop."""

        # If you don't check_size() on the containing window after setting the
        # border width, you may get a nasty surprise: a minimally-sized window.
        # Work around the problem by setting the border width,
        # spacing, etc. _after_ creating all of the tablel content and
        # _before_ showing the containing window.
        self.table.set_border_width(5)
        self.table.set_col_spacings(5)
        self.table.set_row_spacings(5)
        self.top.show()
        gtk.mainloop()
    def addEntry(self, label):
        """Add a new labelled entry.
        `label' specifies the label to be displayed."""
        # Add each new entry to the next available row.
        row = len(self.entry)

        l = gtk.GtkLabel(label + ":")
        l.set_alignment(1.0, 0.5)
        self.table.attach(l, 0, 1, row, row + 1)
        l.show()
        e = gtk.GtkEntry()
        self.table.attach(e, 1, 2, row, row + 1)
        e.show()
        self.entry.append(e)
        return e
    def addBtnCB(self, *args):
        """Callback invoked when the Add button is clicked."""
        self.addEntry("Row %d" % len(self.entry))
def main():
    """Module mainline (for standalone execution)"""
    ui = UI()
    for label in ["Name", "Quest", "Favorite Color"]:
        ui.addEntry(label)
    ui.mainloop()
if __name__ == "__main__":
    main()
```

Back to Article

Listing Three

```
#!/usr/bin/env python
"""This module shows ways of aligning labels in a form -- and also how
to maintain spacing for dynamically-added labels."""

import gtk
import AppWindow

class UI:
    """Builds a form consisting of several rows of labelled entries."""
    def __init__(self):
        """Initialize a new instance."""
        self.top = AppWindow.AppWindow(title="Alignment Demo")
        self.box = gtk.GtkVBox()
        self.top.add(self.box)
        self.box.show()
        self.table = gtk.GtkTable(cols=2)
        self.box.pack_start(self.table)
        self.table.show()
        self.addBtn = gtk.GtkButton("Add Row")
        self.addBtn.connect("clicked", self.addBtnCB)
        self.box.pack_end(self.addBtn, fill=gtk.FALSE, expand=gtk.FALSE)
        self.addBtn.show()

        # Keep a list of the entries added.
        self.entry = []
    def mainloop(self):
        """Show the window and run the main event loop."""
        # If you don't check_size() on the containing window after setting the
        # border width, you may get a nasty surprise: a minimally-sized window.
        # Work around the problem by setting the border width,
        # spacing, etc. _after_ creating all of the tablel content and
        # _before_ showing the containing window.
        self.table.set_border_width(5)
        self.table.set_col_spacings(5)
        self.table.set_row_spacings(5)

        self.top.show()
        gtk.mainloop()
    def addEntry(self, label):
        """Add a new labelled entry.
        `label' specifies the label to be displayed."""
        # Each new entry is added to the next available row.
        row = len(self.entry)

        l = gtk.GtkLabel(label + ":")
        l.set_alignment(1.0, 0.5)
        self.table.attach(l, 0, 1, row, row + 1)
        l.show()

        e = gtk.GtkEntry()
        self.table.attach(e, 1, 2, row, row + 1)
        e.show()

        self.entry.append(e)
        # set_row_spacings() affects only the rows of which the
        # table is already aware, or so it seems.
        # To work around this, set the spacing for each row as it is added.
        if row > 0:
            self.table.set_row_spacing(row - 1, 5)
        return e
    def addBtnCB(self, *args):
        """Callback invoked when the Add button is clicked."""
        self.addEntry("Row %d" % len(self.entry))
def main():
    """Module mainline (for standalone execution)"""
    ui = UI()
    for label in ["Name", "Quest", "Favorite Color"]:
        ui.addEntry(label)
    ui.mainloop()
if __name__ == "__main__":
    main()
```

Back to Article

Listing Four

```
#!/usr/bin/env python
"""This module shows how to adjust the limits on a GtkSpinButton, and
how to use GtkAdjustments to keep several widgets in lock step."""

import gtk, GTK, AppWindow
class UI:
    """Demonstrates how to control the limits on a spinbutton."""
    def __init__(self, upperBound=10):
        """Initialize a new instance."""
        self.top = AppWindow.AppWindow(title="Spinbutton Limits")
        self.table = gtk.GtkTable()
        self.top.add(self.table)
        self.table.show()
        self.spinBtn = gtk.GtkSpinButton()
        self.spinBtn.connect("activate", self.spinEntryActivateCB)
        self.table.attach(self.spinBtn, 0, 1, 0, 1)
        self.spinBtn.show()
        self.scrollbar = gtk.GtkHScrollbar()
        self.scrollbar['width'] = 200
        self.table.attach(self.scrollbar, 0, 1, 1, 2)
        self.scrollbar.show()
        self.scale = gtk.GtkHScale()
        self.table.attach(self.scale, 0, 1, 2, 3)
        self.scale.show()
        self.setUpperBound(upperBound)
    def mainloop(self):
        """Show the application window and run the main event loop."""
        self.top.show()
        gtk.mainloop()
    def spinEntryActivateCB(self, *args):
        """Callback invoked when the user enters a new value in the
        spinbutton entry field."""
        # Use get_text() and convert the value to a number; get_value() clips
        # the value to the current spinbutton bounds before returning it.
        self.setUpperBound(float(self.spinBtn.get_text()))
    def setUpperBound(self, upperBound):
        """Set a new upper bound for the spinbutton, scrollbar et al."""
        # Ignore any upper bounds which are less than the current
        # lower bound. (Default lower bound for a spinbutton is zero.)
        if upperBound <= 0:
            return
        # Apply the new upper bound, but keep all other adjustments
        # at their current value.

        # Can't use self.spinBtn.get_adjustment(), because it references
        # a non-existent attribute ('sel' should be 'self').
        o = self.spinBtn._o
        adj = gtk._obj2inst(gtk._gtk.gtk_spin_button_get_adjustment(o))

        # Create a new GtkAdjustment and tell spinbutton, etc, to use it.
        newAdj = gtk.GtkAdjustment(adj.value, adj.lower, upperBound,
                     max(1, adj.step_increment), max(1, adj.page_increment),
                     max(1, adj.page_size))
        self.spinBtn.set_adjustment(newAdj)
        self.scrollbar.set_adjustment(newAdj)
        self.scale.set_adjustment(newAdj)
def main():
    """Module mainline (for standalone execution)"""
    ui = UI()
    ui.mainloop()
if __name__ == "__main__":
    main()
```

Back to Article

(a)
```
tree.insert_node(parent, sibling, text, spacing=5,
                 pixmap_closed=None, mask_closed=None,
                 pixmap_opened=None, mask_opened=None,
                 is_leaf=TRUE, expanded=FALSE)
```

(b)
```
tree.freeze()
try:
    ...
finally:
    tree.thaw()
```

Example 1: (a) GtkCTree.insert_node default parameters; (b) GtkCTree and using the finally clause.

```
class GtkModalDialog:
    def runModal(self):
        self.window.show()
        gtk.mainloop()
    def endModal(self):
        self.window.hide()
        gtk.mainquit()
```

Example 2: Implementing a modal event loop.

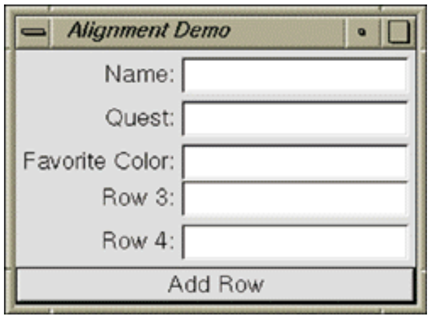
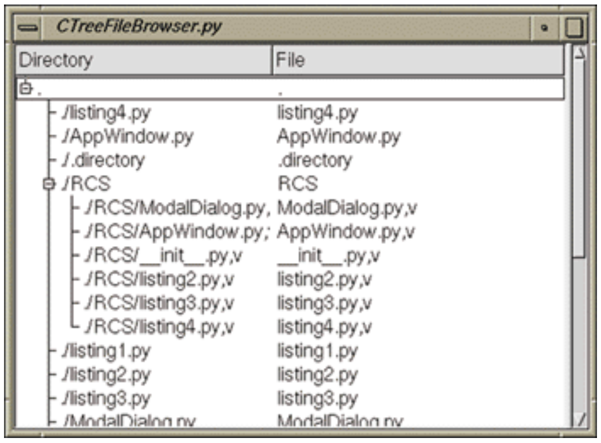


Figure 1: Aligning widgets.


CTreeFileBrowser.py

Directory	File
.	.
./listing4.py	listing4.py
./AppWindow.py	AppWindow.py
./.directory	.directory
./RCS	RCS
./RCS/ModalDialog.py,	ModalDialog.py,v
./RCS/AppWindow.py;	AppWindow.py,v
./RCS/__init__.py,v	__init__.py,v
./RCS/listing2.py,v	listing2.py,v
./RCS/listing3.py,v	listing3.py,v
./RCS/listing4.py,v	listing4.py,v
./listing1.py	listing1.py
./listing2.py	listing2.py
./listing3.py	listing3.py
./ModalDialog.py	ModalDialog.py

Figure 2: Working with GtkCTrees.