PyGTK 2.0 Tutorial

John Finlay

October 7, 2012

PyGTK 2.0 Tutorial
by John Finlay

Published March 2, 2006

Contents

1 Introduction **1**
- 1.1 Exploring PyGTK . . . 2

2 Getting Started **5**
- 2.1 Hello World in PyGTK . . . 7
- 2.2 Theory of Signals and Callbacks . . . 9
- 2.3 Events . . . 10
- 2.4 Stepping Through Hello World . . . 11

3 Moving On **15**
- 3.1 More on Signal Handlers . . . 15
- 3.2 An Upgraded Hello World . . . 15

4 Packing Widgets **19**
- 4.1 Theory of Packing Boxes . . . 19
- 4.2 Details of Boxes . . . 20
- 4.3 Packing Demonstration Program . . . 22
- 4.4 Packing Using Tables . . . 27
- 4.5 Table Packing Example . . . 28

5 Widget Overview **31**
- 5.1 Widget Hierarchy . . . 31
- 5.2 Widgets Without Windows . . . 34

6 The Button Widget **35**
- 6.1 Normal Buttons . . . 35
- 6.2 Toggle Buttons . . . 38
- 6.3 Check Buttons . . . 40
- 6.4 Radio Buttons . . . 42

7 Adjustments **45**
- 7.1 Creating an Adjustment . . . 45
- 7.2 Using Adjustments the Easy Way . . . 45
- 7.3 Adjustment Internals . . . 46

8 Range Widgets **49**
- 8.1 Scrollbar Widgets . . . 49
- 8.2 Scale Widgets . . . 49
  - 8.2.1 Creating a Scale Widget . . . 49
  - 8.2.2 Methods and Signals (well, methods, at least) . . . 50
- 8.3 Common Range Methods . . . 50
  - 8.3.1 Setting the Update Policy . . . 50
  - 8.3.2 Getting and Setting Adjustments . . . 51
- 8.4 Key and Mouse Bindings . . . 51
- 8.5 Range Widget Example . . . 51

9 Miscellaneous Widgets **57**
- 9.1 Labels . . . 57
- 9.2 Arrows . . . 60
- 9.3 The Tooltips Object . . . 61
- 9.4 Progress Bars . . . 63
- 9.5 Dialogs . . . 67
  - 9.5.1 Message Dialogs . . . 68
- 9.6 Images . . . 68
  - 9.6.1 Pixmaps . . . 71

9.7 Rulers . . . 77
9.8 Statusbars . . . 79
9.9 Text Entries . . . 81
9.10 Spin Buttons . . . 83
9.11 Combo Widget . . . 89
9.12 Calendar . . . 90
9.13 Color Selection . . . 96
9.14 File Selections . . . 100
9.15 Font Selection Dialog . . . 102

10 Container Widgets **105**
10.1 The EventBox . . . 105
10.2 The Alignment widget . . . 106
10.3 Fixed Container . . . 107
10.4 Layout Container . . . 108
10.5 Frames . . . 111
10.6 Aspect Frames . . . 113
10.7 Paned Window Widgets . . . 115
10.8 Viewports . . . 118
10.9 Scrolled Windows . . . 118
10.10 Button Boxes . . . 120
10.11 Toolbar . . . 124
10.12 Notebooks . . . 128
10.13 Plugs and Sockets . . . 133
10.13.1 Plugs . . . 133
10.13.2 Sockets . . . 134

11 Menu Widget **137**
11.1 Manual Menu Creation . . . 137
11.2 Manual Menu Example . . . 139
11.3 Using ItemFactory . . . 141
11.4 Item Factory Example . . . 141

12 Drawing Area **145**
12.1 Graphics Context . . . 145
12.2 Drawing Methods . . . 149

13 TextView Widget **155**
13.1 TextView Overview . . . 155
13.2 TextViews . . . 155
13.3 Text Buffers . . . 160
13.3.1 TextBuffer Status Information . . . 161
13.3.2 Creating TextIters . . . 161
13.3.3 Text Insertion, Retrieval and Deletion . . . 162
13.3.4 TextMarks . . . 163
13.3.5 Creating and Applying TextTags . . . 163
13.3.6 Inserting Images and Widgets . . . 164
13.4 Text Iters . . . 165
13.4.1 TextIter Attributes . . . 165
13.4.2 Text Attributes at a TextIter . . . 165
13.4.3 Copying a TextIter . . . 166
13.4.4 Retrieving Text and Objects . . . 166
13.4.5 Checking Conditions at a TextIter . . . 166
13.4.6 Checking Location in Text . . . 167
13.4.7 Moving Through Text . . . 168
13.4.8 Moving to a Specific Location . . . 168
13.4.9 Searching in Text . . . 169
13.5 Text Marks . . . 169
13.6 Text Tags and Tag Tables . . . 170

13.6.1 Text Tags . . . 170
13.6.2 Text Tag Tables . . . 172
13.7 A TextView Example . . . 172

14 Tree View Widget 175
14.1 Overview . . . 175
14.2 The TreeModel Interface and Data Stores . . . 178
14.2.1 Introduction . . . 178
14.2.2 Creating TreeStore and ListStore Objects . . . 178
14.2.3 Referring to TreeModel Rows . . . 179
14.2.3.1 Tree Paths . . . 179
14.2.3.2 TreeIters . . . 180
14.2.3.3 TreeRowReferences . . . 180
14.2.4 Adding Rows . . . 181
14.2.4.1 Adding Rows to a ListStore . . . 181
14.2.4.2 Adding Rows to a TreeStore . . . 181
14.2.4.3 Large Data Stores . . . 182
14.2.5 Removing Rows . . . 182
14.2.5.1 Removing Rows From a ListStore . . . 182
14.2.5.2 Removing Rows From a TreeStore . . . 182
14.2.6 Managing Row Data . . . 183
14.2.6.1 Setting and Retrieving Data Values . . . 183
14.2.6.2 Rearranging ListStore Rows . . . 183
14.2.6.3 Rearranging TreeStore Rows . . . 184
14.2.6.4 Managing Multiple Rows . . . 185
14.2.7 Python Protocol Support . . . 186
14.2.8 TreeModel Signals . . . 188
14.2.9 Sorting TreeModel Rows . . . 188
14.2.9.1 The TreeSortable Interface . . . 188
14.2.9.2 Sorting in ListStores and TreeStores . . . 189
14.3 TreeViews . . . 189
14.3.1 Creating a TreeView . . . 189
14.3.2 Getting and Setting the TreeView Model . . . 189
14.3.3 Setting TreeView Properties . . . 190
14.4 CellRenderers . . . 191
14.4.1 Overview . . . 191
14.4.2 CellRenderer Types . . . 191
14.4.3 CellRenderer Properties . . . 191
14.4.4 CellRenderer Attributes . . . 193
14.4.5 Cell Data Function . . . 194
14.4.6 CellRendererText Markup . . . 196
14.4.7 Editable Text Cells . . . 197
14.4.8 Activatable Toggle Cells . . . 197
14.4.9 Editable and Activatable Cell Example Program . . . 198
14.5 TreeViewColumns . . . 200
14.5.1 Creating TreeViewColumns . . . 200
14.5.2 Managing CellRenderers . . . 201
14.6 Manipulating TreeViews . . . 201
14.6.1 Managing Columns . . . 201
14.6.2 Expanding and Collapsing Child Rows . . . 202
14.7 TreeView Signals . . . 202
14.8 TreeSelections . . . 203
14.8.1 Getting the TreeSelection . . . 203
14.8.2 TreeSelection Modes . . . 203
14.8.3 Retrieving the Selection . . . 204
14.8.4 Using a TreeSelection Function . . . 204
14.8.5 Selecting and Unselecting Rows . . . 205
14.9 TreeView Drag and Drop . . . 205
14.9.1 Drag and Drop Reordering . . . 205

14.9.2 External Drag and Drop . . . 205
14.9.3 TreeView Drag and Drop Example . . . 207
14.10 TreeModelSort and TreeModelFilter . . . 210
14.10.1 TreeModelSort . . . 210
14.10.2 TreeModelFilter . . . 211
14.11 The Generic TreeModel . . . 214
14.11.1 GenericTreeModel Overview . . . 214
14.11.2 The GenericTreeModel Interface . . . 214
14.11.3 Adding and Removing Rows . . . 218
14.11.4 Memory Management . . . 220
14.11.5 Other Interfaces . . . 220
14.11.6 Applying The GenericTreeModel . . . 221
14.12 The Generic CellRenderer . . . 221

15 New Widgets in PyGTK 2.2 **223**
15.1 Clipboards . . . 223
15.1.1 Creating A Clipboard . . . 223
15.1.2 Using Clipboards with Entry, Spinbutton and TextView . . . 223
15.1.3 Setting Data on a Clipboard . . . 224
15.1.4 Retrieving the Clipboard Contents . . . 225
15.1.5 A Clipboard Example . . . 225

16 New Widgets in PyGTK 2.4 **227**
16.1 The Action and ActionGroup Objects . . . 228
16.1.1 Actions . . . 228
16.1.1.1 Creating Actions . . . 228
16.1.1.2 Using Actions . . . 228
16.1.1.3 Creating Proxy Widgets . . . 230
16.1.1.4 Action Properties . . . 232
16.1.1.5 Actions and Accelerators . . . 233
16.1.1.6 Toggle Actions . . . 234
16.1.1.7 Radio Actions . . . 234
16.1.1.8 An Actions Example . . . 235
16.1.2 ActionGroups . . . 235
16.1.2.1 Creating ActionGroups . . . 235
16.1.2.2 Adding Actions . . . 235
16.1.2.3 Retrieving Actions . . . 237
16.1.2.4 Controlling Actions . . . 237
16.1.2.5 An ActionGroup Example . . . 237
16.1.2.6 ActionGroup Signals . . . 238
16.2 ComboBox and ComboBoxEntry Widgets . . . 238
16.2.1 ComboBox Widgets . . . 238
16.2.1.1 Basic ComboBox Use . . . 238
16.2.1.2 Advanced ComboBox Use . . . 239
16.2.2 ComboBoxEntry Widgets . . . 242
16.2.2.1 Basic ComboBoxEntry Use . . . 242
16.2.2.2 Advanced ComboBoxEntry Use . . . 243
16.3 ColorButton and FontButton Widgets . . . 244
16.3.1 ColorButton Widgets . . . 244
16.3.2 FontButton Widgets . . . 245
16.4 EntryCompletion Objects . . . 247
16.5 Expander Widgets . . . 249
16.6 File Selections using FileChooser-based Widgets . . . 250
16.7 The UIManager . . . 252
16.7.1 Overview . . . 252
16.7.2 Creating a UIManager . . . 252
16.7.3 Adding and Removing ActionGroups . . . 253
16.7.4 UI Descriptions . . . 253
16.7.5 Adding and Removing UI Descriptions . . . 255

16.7.6 Accessing UI Widgets . . . 256
16.7.7 A Simple UIManager Example . . . 257
16.7.8 Merging UI Descriptions . . . 257
16.7.9 UIManager Signals . . . 259

17 Undocumented Widgets **261**
17.1 Accel Label . . . 261
17.2 Option Menu . . . 261
17.3 Menu Items . . . 261
17.3.1 Check Menu Item . . . 261
17.3.2 Radio Menu Item . . . 261
17.3.3 Separator Menu Item . . . 261
17.3.4 Tearoff Menu Item . . . 261
17.4 Curves . . . 261
17.5 Gamma Curve . . . 261

18 Setting Widget Attributes **263**
18.1 Widget Flag Methods . . . 263
18.2 Widget Display Methods . . . 264
18.3 Widget Accelerators . . . 264
18.4 Widget Name Methods . . . 265
18.5 Widget Styles . . . 265

19 Timeouts, IO and Idle Functions **269**
19.1 Timeouts . . . 269
19.2 Monitoring IO . . . 269
19.3 Idle Functions . . . 270

20 Advanced Event and Signal Handling **271**
20.1 Signal Methods . . . 271
20.1.1 Connecting and Disconnecting Signal Handlers . . . 271
20.1.2 Blocking and Unblocking Signal Handlers . . . 271
20.1.3 Emitting and Stopping Signals . . . 272
20.2 Signal Emission and Propagation . . . 272

21 Managing Selections **273**
21.1 Selection Overview . . . 273
21.2 Retrieving the Selection . . . 273
21.3 Supplying the Selection . . . 277

22 Drag-and-drop (DND) **281**
22.1 DND Overview . . . 281
22.2 DND Properties . . . 281
22.3 DND Methods . . . 282
22.3.1 Setting Up the Source Widget . . . 282
22.3.2 Signals On the Source Widget . . . 283
22.3.3 Setting Up a Destination Widget . . . 283
22.3.4 Signals On the Destination Widget . . . 284

23 GTK's rc Files **289**
23.1 Functions For rc Files . . . 289
23.2 GTK's rc File Format . . . 289
23.3 Example rc file . . . 291

24 Scribble, A Simple Example Drawing Program **293**
24.1 Scribble Overview . . . 293
24.2 Event Handling . . . 293
24.2.1 Scribble - Event Handling . . . 298
24.3 The DrawingArea Widget, And Drawing . . . 300

25 Tips For Writing PyGTK Applications 303
25.1 The user should drive the interface, not the reverse . . . 303
25.2 Separate your data model from your interface . . . 303
25.3 How to Separate Callback Methods From Signal Handlers . . . 303
25.3.1 Overview . . . 303
25.3.2 Inheritance . . . 304
25.3.3 Inheritance Applied To PyGTK . . . 304

26 Contributing 309

27 Credits 311
27.1 Original GTK+ Credits . . . 311
27.2 PyGTK Tutorial Credits . . . 312

28 Tutorial Copyright and Permissions Notice 313

A GTK Signals 315
A.1 GtkObject . . . 315
A.2 GtkWidget . . . 315
A.3 GtkData . . . 317
A.4 GtkContainer . . . 317
A.5 GtkCalendar . . . 317
A.6 GtkEditable . . . 317
A.7 GtkNotebook . . . 318
A.8 GtkList . . . 318
A.9 GtkMenuShell . . . 318
A.10 GtkToolbar . . . 318
A.11 GtkButton . . . 319
A.12 GtkItem . . . 319
A.13 GtkWindow . . . 319
A.14 GtkHandleBox . . . 319
A.15 GtkToggleButton . . . 319
A.16 GtkMenuItem . . . 319
A.17 GtkCheckMenuItem . . . 319
A.18 GtkInputDialog . . . 320
A.19 GtkColorSelection . . . 320
A.20 GtkStatusBar . . . 320
A.21 GtkCurve . . . 320
A.22 GtkAdjustment . . . 320

B Code Examples 321
B.1 scribblesimple.py . . . 321

C ChangeLog 325

List of Figures

2 Getting Started
2.1 Simple PyGTK Window . . . 6
2.2 Hello World Example Program . . . 8

3 Moving On
3.1 Upgraded Hello World Example . . . 17

4 Packing Widgets
4.1 Packing: A Single Widget in a Container . . . 19
4.2 Packing: Two Widgets in a Container . . . 20
4.3 Packing: Five Variations . . . 21
4.4 Packing with Spacing and Padding . . . 22
4.5 Packing with pack_end() . . . 22
4.6 Packing using a Table . . . 28

6 The Button Widget
6.1 Button with Pixmap and Label . . . 36
6.2 Toggle Button Example . . . 39
6.3 Check Button Example . . . 41
6.4 Radio Buttons Example . . . 43

8 Range Widgets
8.1 Range Widgets Example . . . 52

9 Miscellaneous Widgets
9.1 Label Examples . . . 58
9.2 Arrows Buttons Examples . . . 60
9.3 Tooltips Example . . . 62
9.4 ProgressBar Example . . . 65
9.5 Example Images in Buttons . . . 69
9.6 Pixmap in a Button Example . . . 72
9.7 Wheelbarrow Example Shaped Window . . . 74
9.8 Rulers Example . . . 78
9.9 Statusbar Example . . . 80
9.10 Entry Example . . . 82
9.11 Spin Button Example . . . 86
9.12 Calendar Example . . . 92
9.13 Color Selection Dialog Example . . . 98
9.14 File Selection Example . . . 101
9.15 Font Selection Dialog . . . 102

10 Container Widgets
10.1 Event Box Example . . . 105
10.2 Fixed Example . . . 107
10.3 Layout Example . . . 110
10.4 Frame Example . . . 112
10.5 Aspect Frame Example . . . 114
10.6 Paned Example . . . 116

10.7 Scrolled Window Example 119
10.8 Toolbar Example 128
10.9 Notebook Example 130

11 Menu Widget
11.1 Menu Example 139
11.2 Item Factory Example 142

12 Drawing Area
12.1 Drawing Area Example 151

13 TextView Widget
13.1 Basic TextView Example 157
13.2 TextView Example 173

14 Tree View Widget
14.1 Basic TreeView Example Program 178
14.2 TreeViewColumns with CellRenderers 191
14.3 CellRenderer Data Function 194
14.4 File Listing Example Using Cell Data Functions 196
14.5 CellRendererText Markup 197
14.6 Editable and Activatable Cells 200
14.7 Expander Arrow in Second Column 202
14.8 TreeView Drag and Drop Example 209
14.9 TreeModelSort Example 211
14.10TreeModelFilter Visibility Example 213
14.11Generic TreeModel Example Program 218

15 New Widgets in PyGTK 2.2
15.1 Clipboard Example Program 226

16 New Widgets in PyGTK 2.4
16.1 Simple Action Example 230
16.2 Basic Action Example 232
16.3 Actions Example 235
16.4 ActionGroup Example 237
16.5 Basic ComboBox 239
16.6 ComboBox with Wrapped Layout 241
16.7 Basic ComboBoxEntry 243
16.8 ColorButton Example 245
16.9 FontButton Example 247
16.10EntryCompletion 248
16.11Expander Widget 249
16.12File Selection Example 251
16.13Simple UIManager Example 257
16.14UIMerge Example 259

21 Managing Selections
21.1 Get Selection Example 275
21.2 Set Selection Example 278

22 Drag-and-drop (DND)
22.1 Drag and Drop Example 285

24 Scribble, A Simple Example Drawing Program
24.1 Scribble Drawing Program Example 293
24.2 Simple Scribble Example 299

List of Tables

22 Drag-and-drop (DND)
22.1 Source Widget Signals 283
22.2 Destination Widget Signals 284

Abstract

This tutorial describes the use of the Python PyGTK module.

Chapter 1

Introduction

PyGTK 2.0 is a set of Python modules which provide a Python interface to GTK+ 2.X. Throughout the rest of this document PyGTK refers to the 2.X version of PyGTK and GTK and GTK+ refer to the 2.X version of GTK+. The primary web site for PyGTK is www.pygtk.org. The primary author of PyGTK is:

- James Henstridge james@daa.com.au

who is assisted by the developers listed in the AUTHORS file in the PyGTK distribution and the PyGTK community.

Python is an extensible, object-oriented interpreted programming language which is provided with a rich set of modules providing access to a large number of operating system services, internet services (such as HTML, XML, FTP, etc.), graphics (including OpenGL, TK, etc.), string handling functions, mail services (IMAP, SMTP, POP3, etc.), multimedia (audio, JPEG) and cryptographic services. In addition there are many other modules available from third parties providing many other services. Python is licensed under terms similar to the LGPL license and is available for Linux, Unix , Windows and Macintosh operating systems. More information on Python is available at www.python.org . The primary Author of Python is:

- Guido van Rossum guido@python.org

GTK (GIMP Toolkit) is a library for creating graphical user interfaces. It is licensed using the LGPL license, so you can develop open software, free software, or even commercial non-free software using GTK without having to spend anything for licenses or royalties.

It's called the GIMP toolkit because it was originally written for developing the GNU Image Manipulation Program (GIMP), but GTK has now been used in a large number of software projects, including the GNU Network Object Model Environment (GNOME) project. GTK is built on top of GDK (GIMP Drawing Kit) which is basically a wrapper around the low-level functions for accessing the underlying windowing functions (Xlib in the case of the X windows system). The primary authors of GTK are:

- Peter Mattis petm@xcf.berkeley.edu
- Spencer Kimball spencer@xcf.berkeley.edu
- Josh MacDonald jmacd@xcf.berkeley.edu

GTK is currently maintained by:

- Owen Taylor otaylor@redhat.com
- Tim Janik timj@gtk.org

GTK is essentially an object oriented application programmers interface (API). Although written completely in C, it is implemented using the idea of classes and callback functions (pointers to functions).

There is also a third component called GLib which contains a few replacements for some standard calls, as well as some additional functions for handling linked lists, etc. The replacement functions are used to increase GTK's portability, as some of the functions implemented here are not available or are nonstandard on other unixes such as `g_strerror()`. Some also contain enhancements to the libc versions, such as `g_malloc` that has enhanced debugging utilities.

In version 2.0, GLib has picked up the type system which forms the foundation for GTK's class hierarchy, the signal system which is used throughout GTK, a thread API which abstracts the different native thread APIs of the various platforms and a facility for loading modules.

As the last component, GTK uses the Pango library for internationalized text output.

This tutorial describes the Python interface to GTK+ and is based on the GTK+ 2.0 Tutorial written by Tony Gale and Ian Main. This tutorial attempts to document as much as possible of PyGTK, but is by no means complete.

This tutorial assumes some understanding of Python, and how to create and run Python programs. If you are not familiar with Python, please read the Python Tutorial first. This tutorial does not assume an understanding of GTK; if you are learning PyGTK to learn GTK, please comment on how you found this tutorial, and what you had trouble with. This tutorial does not describe how to compile or install Python, GTK+ or PyGTK.

This tutorial is based on:

- GTK+ 2.0 through GTK+ 2.4
- Python 2.2
- PyGTK 2.0 through PyGTK 2.4

The examples were written and tested on a RedHat 9.0 system.

This document is a "work in progress". Please look for updates on www.pygtk.org.

I would very much like to hear of any problems you have learning PyGTK from this document, and would appreciate input as to how it may be improved. Please see the section on Contributing for further information. If you encounter bugs please file a bug at bugzilla.gnome.org against the pygtk project. The information at www.pygtk.org about Bugzilla may help.

The PyGTK 2.0 Reference Manual is available at http://www.pygtk.org/reference.html. It describes in detail the PyGTK classes.

The PyGTK website (www.pygtk.org) contains other resources useful for learning about PyGTK including a link to the extensive FAQ and other articles and tutorials and an active maillist and IRC channel (see www.pygtk.org for details).

1.1 Exploring PyGTK

Johan Dahlin has written a small Python program (pygtkconsole.py) that runs on Linux and allows interactive exploration of PyGTK. The program provides a Python-like interactive interpreter interface that communicates with a child process that executes that entered commands. The PyGTK modules are loaded by default. A simple example session is:

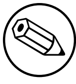
```
moe: 96:1095$ pygtkconsole.py
Python 2.2.2, PyGTK 1.99.14 (Gtk+ 2.0.6)
Interactive console to manipulate GTK+ widgets.
>>> w=Window()
>>> b=Button('Hello')
>>> w.add(b)
>>> def hello(b):
...     print "Hello, World!"
...
>>> b.connect('clicked', hello)
5
>>> w.show_all()
>>> Hello, World!
Hello, World!
Hello, World!

>>> b.set_label("Hi There")
>>>
```

This creates a window containing a button which prints a message ('Hello, World!') when clicked. This program makes it easy to try out various GTK widgets and PyGTK interfaces.

I also use a program that was developed by Brian McErlean as ActiveState recipe 65109 with some mods to make it run with PyGTK 2.X. I call it gpython.py. It works similar to the pygtkconsole.py

program.

NOTE



Both of these programs are known not to work on Microsoft Windows because they rely on Unix specific interfaces.

Chapter 2

Getting Started

To begin our introduction to PyGTK, we'll start with the simplest program possible. This program (base.py) will create a 200x200 pixel window and has no way of exiting except to be killed by using the shell.

```
 1   #!/usr/bin/env python
 2
 3   # example base.py
 4
 5   import pygtk
 6   pygtk.require('2.0')
 7   import gtk
 8
 9   class Base:
10       def __init__(self):
11           self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
12           self.window.show()
13
14       def main(self):
15           gtk.main()
16
17   print __name__
18   if __name__ == "__main__":
19       base = Base()
20       base.main()
```

You can run the above program using:

```
python base.py
```

If base.py is made executable and can be found in your `PATH`, it can be run using:

```
base.py
```

Line 1 will invoke python to run base.py in this case. Lines 5-6 help differentiate between various versions of PyGTK that may be installed on your system. These lines specify that we want to use PyGTK version 2.0 which covers all versions of PyGTK with the major number 2. This prevents the program from using the earlier version of PyGTK if it happens to be installed on your system. Lines 18-20 check if the `__name__` variable is `"__main__"` which indicates that the program is being run directly from python and not being imported into a running python interpreter. In this case the program creates a new instance of the Base class and saves a reference to it in the variable base. It then invokes the method main() to start the GTK+ event processing loop.

A window similar to Figure 2.1 should popup on your display.

Figure 2.1 Simple PyGTK Window

The first line allows the program base.py to be invoked from a Linux or Unix shell program assuming that **python** is found your `PATH`. This line will be the first line in all the example programs.

Lines 5-7 import the PyGTK 2 module and initializes the GTK+ environment. The PyGTK module defines the python interfaces to the GTK+ functions that will be used in the program. For those familiar with GTK+ the initialization includes calling the `gtk_init()` function. This sets up a few things for us such as the default visual and color map, default signal handlers, and checks the arguments passed to your application on the command line, looking for one or more of the following:

- --gtk-module
- --g-fatal-warnings
- --gtk-debug
- --gtk-no-debug
- --gdk-debug
- --gdk-no-debug
- --display
- --sync
- --name
- --class

It removes these from the argument list, leaving anything it does not recognize for your application to parse or ignore. These are a set of standard arguments accepted by all GTK+ applications.

Lines 9-15 define a python class named `Base` that defines a class instance initialization method __init__(). The __init__() function creates a top level window (line 11) and directs GTK+ to display it (line 12). The `gtk.Window` is created in line 11 with the argument `gtk.WINDOW_TOPLEVEL` that specifies that we want the window to undergo window manager decoration and placement. Rather than create a window of 0x0 size, a window without children is set to 200x200 by default so you can still manipulate it.

Lines 14-15 define the main() method that calls the PyGTK `main()` function that, in turn, invokes the GTK+ main event processing loop to handle mouse and keyboard events as well as window events.

Lines 18-20 allow the program to start automatically if called directly or passed as an argument to the python interpreter; in these cases the program name contained in the python variable `__name__`

will be the string "`__main__`" and the code in lines 18-20 will be executed. If the program is loaded into a running python interpreter using an import statement, lines 18-20 will not be executed.

Line 19 creates an instance of the `Base` class called base. A `gtk.Window` is created and displayed as a result.

Line 20 calls the main() method of the `Base` class which starts the GTK+ event processing loop. When control reaches this point, GTK+ will sleep waiting for X events (such as button or key presses), timeouts, or file IO notifications to occur. In our simple example, however, events are ignored.

2.1 Hello World in PyGTK

Now for a program with a widget (a button). It's the PyGTK version of the classic hello world program (helloworld.py).

```
 1   #!/usr/bin/env python
 2
 3   # example helloworld.py
 4
 5   import pygtk
 6   pygtk.require('2.0')
 7   import gtk
 8
 9   class HelloWorld:
10
11       # This is a callback function. The data arguments are ignored
12       # in this example. More on callbacks below.
13       def hello(self, widget, data=None):
14           print "Hello World"
15
16       def delete_event(self, widget, event, data=None):
17           # If you return FALSE in the "delete_event" signal handler,
18           # GTK will emit the "destroy" signal. Returning TRUE means
19           # you don't want the window to be destroyed.
20           # This is useful for popping up 'are you sure you want to quit?'
21           # type dialogs.
22           print "delete event occurred"
23
24           # Change FALSE to TRUE and the main window will not be destroyed
25           # with a "delete_event".
26           return False
27
28       # Another callback
29       def destroy(self, widget, data=None):
30           gtk.main_quit()
31
32       def __init__(self):
33           # create a new window
34           self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
35
36           # When the window is given the "delete_event" signal (this is  ↩
    given
37           # by the window manager, usually by the "close" option, or on the
38           # titlebar), we ask it to call the delete_event () function
39           # as defined above. The data passed to the callback
40           # function is NULL and is ignored in the callback function.
41           self.window.connect("delete_event", self.delete_event)
42
43           # Here we connect the "destroy" event to a signal handler.
44           # This event occurs when we call gtk_widget_destroy() on the  ↩
    window,
45           # or if we return FALSE in the "delete_event" callback.
46           self.window.connect("destroy", self.destroy)
47
48           # Sets the border width of the window.
```

```
49           self.window.set_border_width(10)
50
51           # Creates a new button with the label "Hello World".
52           self.button = gtk.Button("Hello World")
53
54           # When the button receives the "clicked" signal, it will call the
55           # function hello() passing it None as its argument.  The hello()
56           # function is defined above.
57           self.button.connect("clicked", self.hello, None)
58
59           # This will cause the window to be destroyed by calling
60           # gtk_widget_destroy(window) when "clicked".  Again, the destroy
61           # signal could come from here, or the window manager.
62           self.button.connect_object("clicked", gtk.Widget.destroy, self. ←
    window)
63
64           # This packs the button into the window (a GTK container).
65           self.window.add(self.button)
66
67           # The final step is to display this newly created widget.
68           self.button.show()
69
70           # and the window
71           self.window.show()
72
73       def main(self):
74           # All PyGTK applications must have a gtk.main(). Control ends  ←
    here
75           # and waits for an event to occur (like a key press or mouse  ←
    event).
76           gtk.main()
77
78   # If the program is run directly or passed as an argument to the python
79   # interpreter then create a HelloWorld instance and show it
80   if __name__ == "__main__":
81       hello = HelloWorld()
82       hello.main()
```

Figure 2.2 shows the window created by helloworld.py.

Figure 2.2 Hello World Example Program

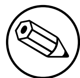
The variables and functions that are defined in the PyGTK module are named as `gtk.*`. For example, the helloworld.py program uses:

```
False
gtk.mainquit()
gtk.Window()
gtk.Button()
```

from the PyGTK module. In future sections I will not specify the gtk module prefix but it will be assumed. The example programs will of course use the module prefixes.

2.2 Theory of Signals and Callbacks

NOTE



In GTK+ version 2.0, the signal system has been moved from GTK to GLib. We won't go into details about the extensions which the GLib 2.0 signal system has relative to the GTK+ 1.2 signal system. The differences should not be apparent to PyGTK users.

Before we look in detail at helloworld.py, we'll discuss signals and callbacks. GTK+ is an event driven toolkit, which means it will sleep in `gtk.main()` until an event occurs and control is passed to the appropriate function.

This passing of control is done using the idea of "signals". (Note that these signals are not the same as the Unix system signals, and are not implemented using them, although the terminology is almost identical.) When an event occurs, such as the press of a mouse button, the appropriate signal will be "emitted" by the widget that was pressed. This is how GTK+ does most of its useful work. There are signals that all widgets inherit, such as "destroy", and there are signals that are widget specific, such as "toggled" on a toggle button.

To make a button perform an action, we set up a signal handler to catch these signals and call the appropriate function. This is done by using a `GtkWidget` (from the `GObject` class) method such as:

```
handler_id = object.connect(name, func, func_data)
```

where object is the `GtkWidget` instance which will be emitting the signal, and the first argument *name* is a string containing the name of the signal you wish to catch. The second argument, *func*, is the function you wish to be called when it is caught, and the third, *func_data*, the data you wish to pass to this function. The method returns a handler_id that can be used to disconnect or block the handler.

The function specified in the second argument is called a "callback function", and should generally be of the form:

```
def callback_func(widget, callback_data):
```

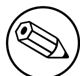
where the first argument will be a pointer to the *widget* that emitted the signal, and the second (*callback_data*) a pointer to the data given as the last argument to the connect() method as shown above.

If the callback function is an object method then it will have the general form:

```
def callback_meth(self, widget, callback_data):
```

where *self* is the object instance invoking the method. This is the form used in the helloworld.py example program.

NOTE



The above form for a signal callback function declaration is only a general guide, as some widget specific signals generate different calling parameters.

Another call used in the helloworld.py example is:

```
handler_id = object.connect_object(name, func, slot_object)
```

connect_object() is like connect(), except that it invokes *func* on *slot_object*, where *slot_object* is usually a widget. connect_object() allows the PyGTK widget methods that only take a single argument (*self*) to be used as signal handlers.

2.3 Events

In addition to the signal mechanism described above, there is a set of events that reflect the X event mechanism. Callbacks may also be attached to these events. These events are:

```
event
button_press_event
button_release_event
scroll_event
motion_notify_event
delete_event
destroy_event
expose_event
key_press_event
key_release_event
enter_notify_event
leave_notify_event
configure_event
focus_in_event
focus_out_event
map_event
unmap_event
property_notify_event
selection_clear_event
selection_request_event
selection_notify_event
proximity_in_event
proximity_out_event
visibility_notify_event
client_event
no_expose_event
window_state_event
```

In order to connect a callback function to one of these events you use the method connect() , as described above, using one of the above event names as the *name* parameter. The callback function (or method) for events has a slightly different form than that for signals:

```
def callback_func(widget, event, callback_data ):

def callback_meth(self, widget, event, callback_data ):
```

`GdkEvent` is a python object type whose type attribute will indicate which of the above events has occurred. The other attributes of the event will depend upon the type of the event. Possible values for the types are:

```
NOTHING
DELETE
DESTROY
EXPOSE
MOTION_NOTIFY
BUTTON_PRESS
_2BUTTON_PRESS
_3BUTTON_PRESS
BUTTON_RELEASE
KEY_PRESS
KEY_RELEASE
ENTER_NOTIFY
LEAVE_NOTIFY
FOCUS_CHANGE
CONFIGURE
MAP
UNMAP
PROPERTY_NOTIFY
SELECTION_CLEAR
SELECTION_REQUEST
```

```
SELECTION_NOTIFY
PROXIMITY_IN
PROXIMITY_OUT
DRAG_ENTER
DRAG_LEAVE
DRAG_MOTION
DRAG_STATUS
DROP_START
DROP_FINISHED
CLIENT_EVENT
VISIBILITY_NOTIFY
NO_EXPOSE
SCROLL
WINDOW_STATE
SETTING
```

These values are accessed by prefacing the event type with gtk.gdk. for example `gtk.gdk.DRAG_ENTER`.

So, to connect a callback function to one of these events we would use something like:

```
button.connect("button_press_event", button_press_callback)
```

This assumes that button is a `GtkButton` widget. Now, when the mouse is over the button and a mouse button is pressed, the function *`button_press_callback`* will be called. This function may be defined as:

```
def button_press_callback(widget, event, data ):
```

The value returned from this function indicates whether the event should be propagated further by the GTK+ event handling mechanism. Returning `True` indicates that the event has been handled, and that it should not propagate further. Returning `False` continues the normal event handling. See Chapter 20 for more details on this propagation process.

The GDK selection and drag-and-drop APIs also emit a number of events which are reflected in GTK+ by signals. See Section 22.3.2 and Section 22.3.4 for details on the signatures of the callback functions for these signals:

```
selection_received
selection_get
drag_begin_event
drag_end_event
drag_data_delete
drag_motion
drag_drop
drag_data_get
drag_data_received
```

2.4 Stepping Through Hello World

Now that we know the theory behind this, let's clarify by walking through the example helloworld.py program.

Lines 9-76 define the `HelloWorld` class that contains all the callbacks as object methods and the object instance initialization method. Let's examine the callback methods.

Lines 13-14 define the hello() callback method that will be called when the button is "clicked". When called the method, prints "Hello World" to the console. We ignore the object instance, the widget and the data parameters in this example, but most callbacks use them. The *`data`* is defined with a default value of `None` because PyGTK will not pass a data value if it is not included in the connect() call; this would trigger an error since the callback is expecting three parameters and may receive only two. Defining a default value of None allows the callback to be called with two or three parameters without error. In this case the data parameter could have been left out since the hello() method will always be called with just two parameters (never called with user data). The next example will use the *`data argument`* to tell us which button was pressed.

```
def hello(self, widget, data=None):
    print "Hello World"
```

The next callback (lines 16-26) is a bit special. The "delete_event" occurs when the window manager sends this event to the application. We have a choice here as to what to do about these events. We can ignore them, make some sort of response, or simply quit the application.

The value you return in this callback lets GTK+ know what action to take. By returning TRUE, we let it know that we don't want to have the "destroy" signal emitted, keeping our application running. By returning FALSE, we ask that "destroy" be emitted, which in turn will call our "destroy" signal handler. Note the comments have been removed for clarity.

```
def delete_event(widget, event, data=None):
    print "delete event occurred"
    return False
```

The destroy() callback method (lines 28-30) causes the program to quit by calling `gtk.main_quit`() . This function tells GTK+ that it is to exit from `gtk.main`() when control is returned to it.

```
def destroy(widget, data=None):
    print "destroy signal occurred"
    gtk.main_quit()
```

Lines 32-71 define the `HelloWorld` object instance initialization method __init__() that creates the window and widgets used by the program.

Line 34 creates a new window, but it is not displayed until we direct GTK+ to show the window near the end of our program. The window reference is saved in an object instance attribute (self.window) for later access.

```
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

Lines 41 and 46 illustrate two examples of connecting a signal handler to an object, in this case, the `window`. Here, the "delete_event" and "destroy" signals are caught. The first is emitted when we use the window manager to kill the window, or when we use the `GtkWidget` destroy() method call. The second is emitted when, in the "delete_event" handler, we return `FALSE`.

```
    self.window.connect("delete_event", self.delete_event)
    self.window.connect("destroy", self.destroy)
```

Line 49 sets an attribute of a container object (in this case the `window`) to have a blank area along the inside of it 10 pixels wide where no widgets will be placed. There are other similar methods that we will look at in Chapter 18

```
    self.window.set_border_width(10)
```

Line 52 creates a new button and saves a reference to it in `self.button`. The button will have the label "Hello World" when displayed.

```
    self.button = gtk.Button("Hello World")
```

In line 57 we attach a signal handler to the button so when it emits the "clicked" signal, our hello() callback method is called. We are not passing any data to hello() so we just pass `None` as the data. Obviously, the "clicked" signal is emitted when we click the button with our mouse pointer. The user data parameter value `None` is not required and could be removed. The callback would then be called with one less parameter.

```
    self.button.connect("clicked", self.hello, None)
```

We are also going to use this button to exit our program. Line 62 illustrates how the "destroy" signal may come from either the window manager, or from our program. When the button is "clicked", same as above, it calls the hello() callback first, and then the following one in the order they are set up. You may have as many callbacks as you need, and all will be executed in the order you connected them.

Since we want to use the `GtkWidget` destroy() method that accepts one argument (the widget to be destroyed - in this case the `window`), we use the connect_object() method and pass it the reference to the window. The connect_object() method arranges to pass the *`window`* as the first callback argument instead of the button.

When the `gtk.Widget` destroy() method is called it will cause the "destroy" signal to be emitted from the window which will in turn cause the `HelloWorld` destroy() method to be called to end the program.

```
self.button.connect_object("clicked", gtk.Widget.destroy, self.window)
```

Line 65 is a packing call, which will be explained in depth later on in Chapter 4 . But it is fairly easy to understand. It simply tells GTK+ that the button is to be placed in the window where it will be displayed. Note that a GTK+ container can only contain one widget. There are other widgets, described later, that are designed to layout multiple widgets in various ways.

```
self.window.add(self.button)
```

Now we have everything set up the way we want it to be. With all the signal handlers in place, and the button placed in the window where it should be, we ask GTK+ (lines 66 and 69) to "show" the widgets on the screen. The window widget is shown last so the whole window will pop up at once rather than seeing the window pop up, and then the button forming inside of it. Although with such a simple example, you'd never notice.

```
self.button.show()

self.window.show()
```

Widgets also have a hide() that is the opposite of show(). It doesn't actually destroy the widget, but it removes the widget renderining from your display. This can be reversed with another show() call.

Lines 73-75 define the main() method which calls the `gtk.main`() function

```
def main(self):
    gtk.main()
```

Lines 80-82 allow the program to run automatically if called directly or as an argument of the python interpreter. Line 81 creates an instance of the `HelloWorld` class and saves a reference to it in the `hello` variable. Line 82 calls the `HelloWorld` class main() method to start the GTK+ event processing loop.

```
if __name__ == "__main__":
    hello = HelloWorld()
    hello.main()
```

Now, when we click the mouse button on a GTK+ button, the widget emits a "clicked" signal. In order for us to use this information, our program sets up a signal handler to catch that signal, which dispatches the function of our choice. In our example, when the button we created is "clicked", the hello() method is called with the `None` argument, and then the next handler for this signal is called. The next handler calls the widget destroy() function with the window as its argument thereby causing the window to emit the "destroy" signal, which is caught, and calls our `HelloWorld` destroy() method

Another course of events is to use the window manager to kill the window, which will cause the "delete_event" to be emitted. This will call our "delete_event" handler. If we return `TRUE` here, the window will be left as is and nothing will happen. Returning `FALSE` will cause GTK+ to emit the "destroy" signal that causes the `HelloWorld` "destroy" callback to be called, exiting GTK.

Chapter 3

Moving On

3.1 More on Signal Handlers

Lets take another look at the connect() call.

```
object.connect(name, func, func_data)
```

The return value from a connect() call is an integer tag that identifies your callback. As stated above, you may have as many callbacks per signal and per object as you need, and each will be executed in turn, in the order they were attached.

This tag allows you to remove this callback from the list by using:

```
object.disconnect(id)
```

So, by passing in the tag returned by one of the signal connect methods, you can disconnect a signal handler.

You can also temporarily disable signal handlers with the handler_block() and handler_unblock() pair of methods.

```
object.handler_block(handler_id)

object.handler_unblock(handler_id)
```

3.2 An Upgraded Hello World

```
 1  #!/usr/bin/env python
 2
 3  # example helloworld2.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class HelloWorld2:
10
11      # Our new improved callback.  The data passed to this method
12      # is printed to stdout.
13      def callback(self, widget, data):
14          print "Hello again - %s was pressed" % data
15
16      # another callback
17      def delete_event(self, widget, event, data=None):
18          gtk.main_quit()
19          return False
20
21      def __init__(self):
22          # Create a new window
```

```
23          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
24
25          # This is a new call, which just sets the title of our
26          # new window to "Hello Buttons!"
27          self.window.set_title("Hello Buttons!")
28
29          # Here we just set a handler for delete_event that immediately
30          # exits GTK.
31          self.window.connect("delete_event", self.delete_event)
32
33          # Sets the border width of the window.
34          self.window.set_border_width(10)
35
36          # We create a box to pack widgets into.  This is described in ↩
    detail
37          # in the "packing" section. The box is not really visible, it
38          # is just used as a tool to arrange widgets.
39          self.box1 = gtk.HBox(False, 0)
40
41          # Put the box into the main window.
42          self.window.add(self.box1)
43
44          # Creates a new button with the label "Button 1".
45          self.button1 = gtk.Button("Button 1")
46
47          # Now when the button is clicked, we call the "callback" method
48          # with a pointer to "button 1" as its argument
49          self.button1.connect("clicked", self.callback, "button 1")
50
51          # Instead of add(), we pack this button into the invisible
52          # box, which has been packed into the window.
53          self.box1.pack_start(self.button1, True, True, 0)
54
55          # Always remember this step, this tells GTK that our preparation ↩
    for
56          # this button is complete, and it can now be displayed.
57          self.button1.show()
58
59          # Do these same steps again to create a second button
60          self.button2 = gtk.Button("Button 2")
61
62          # Call the same callback method with a different argument,
63          # passing a pointer to "button 2" instead.
64          self.button2.connect("clicked", self.callback, "button 2")
65
66          self.box1.pack_start(self.button2, True, True, 0)
67
68          # The order in which we show the buttons is not really important, ↩
     but I
69          # recommend showing the window last, so it all pops up at once.
70          self.button2.show()
71          self.box1.show()
72          self.window.show()
73
74  def main():
75      gtk.main()
76
77  if __name__ == "__main__":
78      hello = HelloWorld2()
79      main()
```

Running helloworld2.py produces the window illustrated in Figure 3.1.

Figure 3.1 Upgraded Hello World Example

You'll notice this time there is no easy way to exit the program, you have to use your window manager or command line to kill it. A good exercise for the reader would be to insert a third "Quit" button that will exit the program. You may also wish to play with the options to pack_start() while reading the next section. Try resizing the window, and observe the behavior.

A short commentary on the code differences from the first helloworld program is in order.

As noted above there is no "destroy" event handler in the upgraded helloworld.

Lines 13-14 define a callback method which is similar to the hello() callback in the first helloworld. The difference is that the callback prints a message including data passed in.

Line 27 sets a title string to be used on the titlebar of the window (see Figure 3.1).

Line 39 creates a horizontal box (`gtk.HBox`) to hold the two buttons that are created in lines 45 and 60. Line 42 adds the horizontal box to the window container.

Lines 49 and 64 connect the callback() method to the "clicked" signal of the buttons. Each button sets up a different string to be passed to the callback() method when invoked.

Lines 53 and 66 pack the buttons into the horizontal box. Lines 57 and 70 ask GTK to display the buttons.

Lines 71-72 ask GTK to display the box and the window respectively.

Chapter 4

Packing Widgets

When creating an application, you'll want to put more than one widget inside a window. Our first helloworld example only used one widget so we could simply use the `gtk.Container` add() method to "pack" the widget into the window. But when you want to put more than one widget into a window, how do you control where that widget is positioned? This is where packing comes in.

4.1 Theory of Packing Boxes

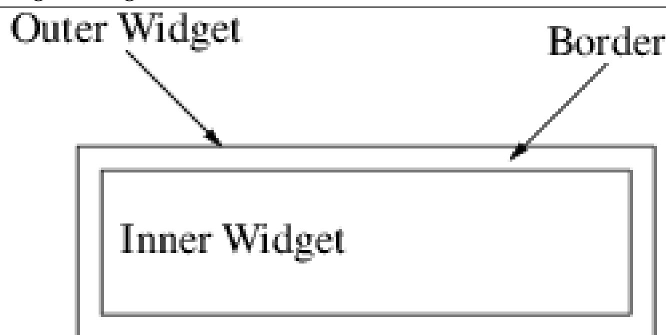
Most packing is done by creating boxes. These are invisible widget containers that we can pack our widgets into which come in two forms, a horizontal box, and a vertical box. When packing widgets into a horizontal box, the objects are inserted horizontally from left to right or right to left depending on the call used. In a vertical box, widgets are packed from top to bottom or vice versa. You may use any combination of boxes inside or beside other boxes to create the desired effect.

To create a new horizontal box, we use a call to `gtk.HBox`(), and for vertical boxes, `gtk.VBox`() . The pack_start() and pack_end() methods are used to place objects inside of these containers. The pack_start() method will start at the top and work its way down in a vbox, and pack left to right in an hbox. The pack_end() method will do the opposite, packing from bottom to top in a vbox, and right to left in an hbox. Using these methods allows us to right justify or left justify our widgets and may be mixed in any way to achieve the desired effect. We will use pack_start() in most of our examples. An object may be another container or a widget. In fact, many widgets are actually containers themselves, including the button, but we usually only use a label inside a button.

You may find when working with containers that the size (and aspect ratio) of your widget isn't quite what you would expect. That's an intentional consequence of the GTK+ box model. The size of any given widget is determined both by by how it packs among the widgets around it and whether or not its container offers it the possibility to expand and fill space available to it.

if you have a container a single child, this child will take up all its space minus its border:

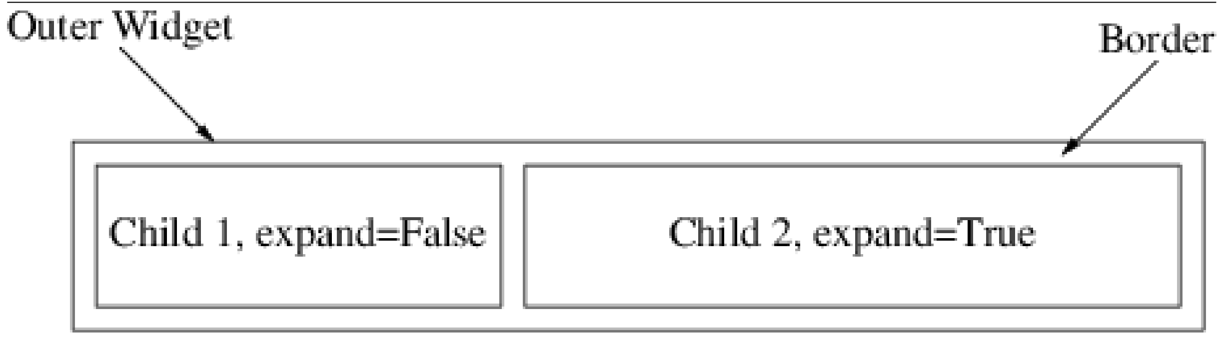
Figure 4.1 Packing: A Single Widget in a Container



If you have a container (say a VBox or HBox) with two (or more) children, they will fight[1] to determine who takes up more space:

[1] A cute analogy; in reality fill, expansion, requested sizes, widget expansion semantics, container packing semantics, electron spins and lunar cycles are computed to determine how much space each widget wins.

Figure 4.2 Packing: Two Widgets in a Container



How much each one actually gets is determined by:

- the default and requested sizes of the widgets, which normally depends on their contents (for labels, in particular).
- the expand and fill arguments supplied to add() or pack_start/pack_end(), all three of which we will describe in more detail later in this chapter:
  - *expand*=`True` means "I will fight for space"
  - *expand*=`False` means "I don't want more space"
  - *fill*=`True` means "If I got more space, I will occupy it with my content"
  - *fill*=`False` means "If I got more space, leave it blank"

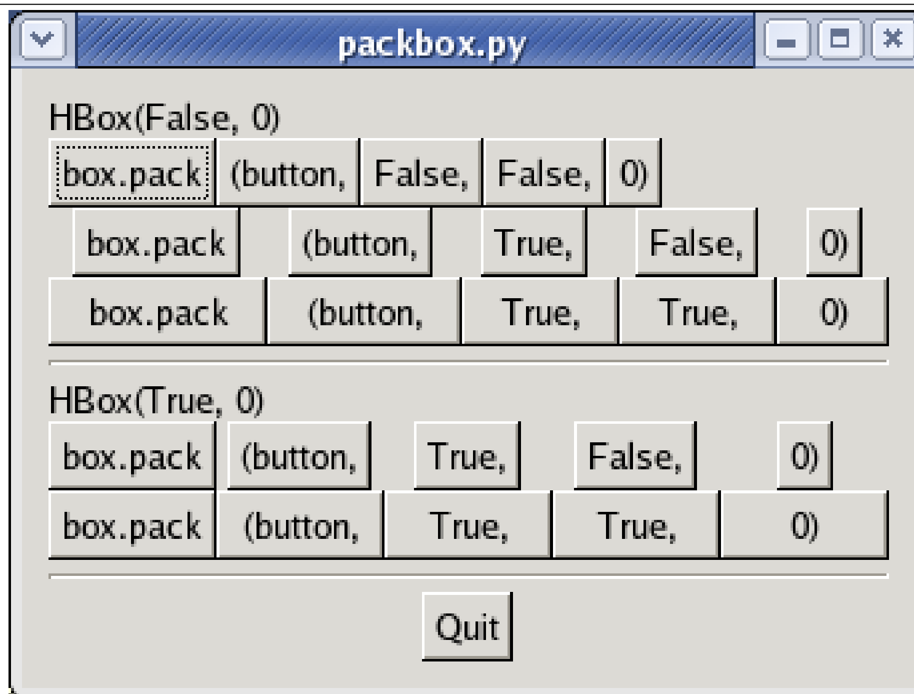
This is important to understand when assembling your interfaces, and is the most peculiar thing about GTK+ programming to a newbie; although the packing-based widget geometry is more complex to understand initially than fixed-width layouts, it is superior because GTK+ windows actually resize properly.

To get an intuitive grasp of the box model, spend some time experimenting with the "packing" tab in Glade.

4.2 Details of Boxes

Because of all this flexibility, packing boxes in GTK can be confusing at first. There are a lot of options, and it's not immediately obvious how they all fit together. In the end, however, there are basically five different styles. Figure 4.3 illustrates the result of running the program packbox.py with an argument of 1:

Figure 4.3 Packing: Five Variations



Each line contains one horizontal box (hbox) with several buttons. The call to pack is shorthand for the call to pack each of the buttons into the hbox. Each of the buttons is packed into the hbox the same way (i.e., same arguments to the pack_start() method).

This is an example of the pack_start() method.

```
box.pack_start(child, expand, fill, padding)
```

box is the box you are packing the object into; the first argument is the *child* object to be packed. The objects will all be buttons for now, so we'll be packing buttons into boxes.

As previously noted, the *expand* argument to pack_start() and pack_end() controls whether the widgets are laid out in the box to fill in all the extra space in the box so the box is expanded to fill the area allotted to it (`True`); or the box is shrunk to just fit the widgets (`False`). Setting expand to `False` will allow you to do right and left justification of your widgets. Otherwise, they will all expand to fit into the box, and the same effect could be achieved by using only one of pack_start() or pack_end().

As previously noted, the *fill* argument to the pack methods control whether the extra space is allocated to the objects themselves (`True`), or as extra padding in the box around these objects (`False`). It only has an effect if the expand argument is also `True`.

Python allows a method or function to be defined with default argument values and argument keywords. Throughout this tutorial I'll show the definition of the functions and methods with defaults and keywords bolded as applicable. For example the pack_start() method is defined as:

```
box.pack_start(child, expand=True, fill=True, padding=0)

box.pack_end(child, expand=True, fill=True, padding=0)
```

child, *expand*, *fill* and *padding* are keywords. The *expand*, *fill* and *padding* arguments have the defaults shown. The *child* argument must be specified.

When creating a new box, the function looks like this:

```
hbox = gtk.HBox(homogeneous=False, spacing=0)

vbox = gtk.VBox(homogeneous=False, spacing=0)
```

The *homogeneous* argument to `gtk.HBox`() and `gtk.VBox`() controls whether each object in the box has the same size (i.e., the same width in an hbox, or the same height in a vbox). If it is set, the pack routines function essentially as if the expand argument was always turned on.

What's the difference between *spacing* (set when the box is created) and *padding* (set when elements are packed)? Spacing is added between objects, and padding is added on either side of an object. Figure 4.4 illustrates the difference; pass an argument of 2 to packbox.py :

Figure 4.4 Packing with Spacing and Padding

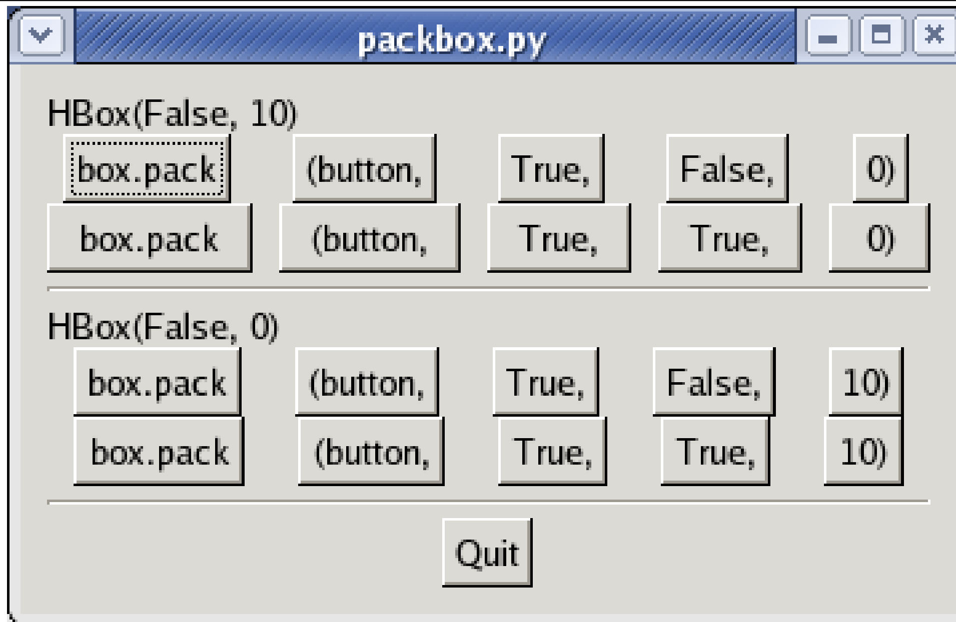
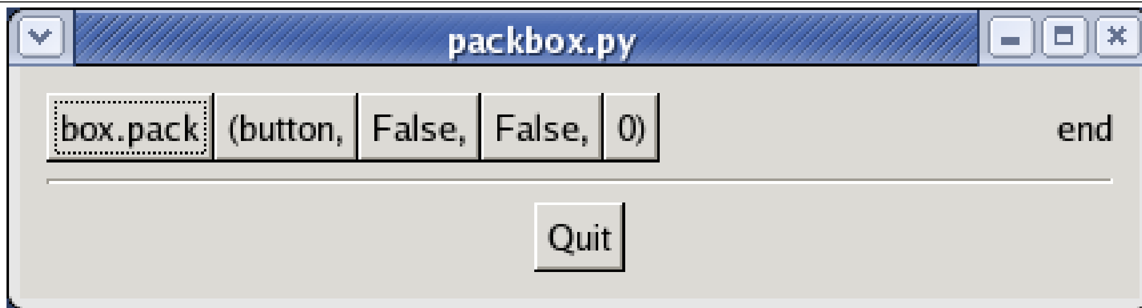


Figure 4.5 illustrates the use of the pack_end() method (pass an argument of 3 to packbox.py). The label "end" is packed with the pack_end() method. It will stick to the right edge of the window when the window is resized.

Figure 4.5 Packing with pack_end()



4.3 Packing Demonstration Program

Here is the code used to create the above images. It's commented fairly heavily so I hope you won't have any problems following it. Run it yourself and play with it.

```
 1 #!/usr/bin/env python
 2
 3 # example packbox.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8 import sys, string
 9
10 # Helper function that makes a new hbox filled with button-labels.  ↩
     Arguments
11 # for the variables we're interested are passed in to this function.  We do
```

```
12 # not show the box, but do show everything inside.
13
14 def make_box(homogeneous, spacing, expand, fill, padding):
15
16     # Create a new hbox with the appropriate homogeneous
17     # and spacing settings
18     box = gtk.HBox(homogeneous, spacing)
19
20     # Create a series of buttons with the appropriate settings
21     button = gtk.Button("box.pack")
22     box.pack_start(button, expand, fill, padding)
23     button.show()
24
25     button = gtk.Button("(button,")
26     box.pack_start(button, expand, fill, padding)
27     button.show()
28
29     # Create a button with the label depending on the value of
30     # expand.
31     if expand == True:
32         button = gtk.Button("True,")
33     else:
34         button = gtk.Button("False,")
35
36     box.pack_start(button, expand, fill, padding)
37     button.show()
38
39     # This is the same as the button creation for "expand"
40     # above, but uses the shorthand form.
41     button = gtk.Button(("False,", "True,")[fill==True])
42     box.pack_start(button, expand, fill, padding)
43     button.show()
44
45     padstr = "%d)" % padding
46
47     button = gtk.Button(padstr)
48     box.pack_start(button, expand, fill, padding)
49     button.show()
50     return box
51
52 class PackBox1:
53     def delete_event(self, widget, event, data=None):
54         gtk.main_quit()
55         return False
56
57     def __init__(self, which):
58
59         # Create our window
60         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
61
62         # You should always remember to connect the delete_event signal
63         # to the main window. This is very important for proper intuitive
64         # behavior
65         self.window.connect("delete_event", self.delete_event)
66         self.window.set_border_width(10)
67
68         # We create a vertical box (vbox) to pack the horizontal boxes into ↩
   .
69         # This allows us to stack the horizontal boxes filled with buttons  ↩
   one
70         # on top of the other in this vbox.
71         box1 = gtk.VBox(False, 0)
72
73         # which example to show. These correspond to the pictures above.
```

```
 74         if which == 1:
 75             # create a new label.
 76             label = gtk.Label("HBox(False, 0)")
 77 
 78             # Align the label to the left side.  We'll discuss this method
 79             # and others in the section on Widget Attributes.
 80             label.set_alignment(0, 0)
 81 
 82             # Pack the label into the vertical box (vbox box1).  Remember ↩
    that
 83             # widgets added to a vbox will be packed one on top of the ↩
    other in
 84             # order.
 85             box1.pack_start(label, False, False, 0)
 86 
 87             # Show the label
 88             label.show()
 89 
 90             # Call our make box function - homogeneous = False, spacing = ↩
    0,
 91             # expand = False, fill = False, padding = 0
 92             box2 = make_box(False, 0, False, False, 0)
 93             box1.pack_start(box2, False, False, 0)
 94             box2.show()
 95 
 96             # Call our make box function - homogeneous = False, spacing = ↩
    0,
 97             # expand = True, fill = False, padding = 0
 98             box2 = make_box(False, 0, True, False, 0)
 99             box1.pack_start(box2, False, False, 0)
100             box2.show()
101 
102             # Args are: homogeneous, spacing, expand, fill, padding
103             box2 = make_box(False, 0, True, True, 0)
104             box1.pack_start(box2, False, False, 0)
105             box2.show()
106 
107             # Creates a separator, we'll learn more about these later,
108             # but they are quite simple.
109             separator = gtk.HSeparator()
110 
111             # Pack the separator into the vbox. Remember each of these
112             # widgets is being packed into a vbox, so they'll be stacked
113             # vertically.
114             box1.pack_start(separator, False, True, 5)
115             separator.show()
116 
117             # Create another new label, and show it.
118             label = gtk.Label("HBox(True, 0)")
119             label.set_alignment(0, 0)
120             box1.pack_start(label, False, False, 0)
121             label.show()
122 
123             # Args are: homogeneous, spacing, expand, fill, padding
124             box2 = make_box(True, 0, True, False, 0)
125             box1.pack_start(box2, False, False, 0)
126             box2.show()
127 
128             # Args are: homogeneous, spacing, expand, fill, padding
129             box2 = make_box(True, 0, True, True, 0)
130             box1.pack_start(box2, False, False, 0)
131             box2.show()
132 
133             # Another new separator.
```

```
134             separator = gtk.HSeparator()
135             # The last 3 arguments to pack_start are:
136             # expand, fill, padding.
137             box1.pack_start(separator, False, True, 5)
138             separator.show()
139         elif which == 2:
140             # Create a new label, remember box1 is a vbox as created
141             # near the beginning of __init__()
142             label = gtk.Label("HBox(False, 10)")
143             label.set_alignment( 0, 0)
144             box1.pack_start(label, False, False, 0)
145             label.show()
146 
147             # Args are: homogeneous, spacing, expand, fill, padding
148             box2 = make_box(False, 10, True, False, 0)
149             box1.pack_start(box2, False, False, 0)
150             box2.show()
151 
152             # Args are: homogeneous, spacing, expand, fill, padding
153             box2 = make_box(False, 10, True, True, 0)
154             box1.pack_start(box2, False, False, 0)
155             box2.show()
156 
157             separator = gtk.HSeparator()
158             # The last 3 arguments to pack_start are:
159             # expand, fill, padding.
160             box1.pack_start(separator, False, True, 5)
161             separator.show()
162 
163             label = gtk.Label("HBox(False, 0)")
164             label.set_alignment(0, 0)
165             box1.pack_start(label, False, False, 0)
166             label.show()
167 
168             # Args are: homogeneous, spacing, expand, fill, padding
169             box2 = make_box(False, 0, True, False, 10)
170             box1.pack_start(box2, False, False, 0)
171             box2.show()
172 
173             # Args are: homogeneous, spacing, expand, fill, padding
174             box2 = make_box(False, 0, True, True, 10)
175             box1.pack_start(box2, False, False, 0)
176             box2.show()
177 
178             separator = gtk.HSeparator()
179             # The last 3 arguments to pack_start are:
180             # expand, fill, padding.
181             box1.pack_start(separator, False, True, 5)
182             separator.show()
183 
184         elif which == 3:
185 
186             # This demonstrates the ability to use pack_end() to
187             # right justify widgets. First, we create a new box as before.
188             box2 = make_box(False, 0, False, False, 0)
189 
190             # Create the label that will be put at the end.
191             label = gtk.Label("end")
192             # Pack it using pack_end(), so it is put on the right
193             # side of the hbox created in the make_box() call.
194             box2.pack_end(label, False, False, 0)
195             # Show the label.
196             label.show()
197 
```

```
198             # Pack box2 into box1
199             box1.pack_start(box2, False, False, 0)
200             box2.show()
201
202             # A separator for the bottom.
203             separator = gtk.HSeparator()
204
205             # This explicitly sets the separator to 400 pixels wide by 5
206             # pixels high. This is so the hbox we created will also be 400
207             # pixels wide, and the "end" label will be separated from the
208             # other labels in the hbox. Otherwise, all the widgets in the
209             # hbox would be packed as close together as possible.
210             separator.set_size_request(400, 5)
211             # pack the separator into the vbox (box1) created near the  ↩
    start
212             # of __init__()
213             box1.pack_start(separator, False, True, 5)
214             separator.show()
215
216         # Create another new hbox.. remember we can use as many as we need!
217         quitbox = gtk.HBox(False, 0)
218
219         # Our quit button.
220         button = gtk.Button("Quit")
221
222         # Setup the signal to terminate the program when the button is  ↩
    clicked
223         button.connect("clicked", lambda w: gtk.main_quit())
224         # Pack the button into the quitbox.
225         # The last 3 arguments to pack_start are:
226         # expand, fill, padding.
227         quitbox.pack_start(button, True, False, 0)
228         # pack the quitbox into the vbox (box1)
229         box1.pack_start(quitbox, False, False, 0)
230
231         # Pack the vbox (box1) which now contains all our widgets, into the
232         # main window.
233         self.window.add(box1)
234
235         # And show everything left
236         button.show()
237         quitbox.show()
238
239         box1.show()
240         # Showing the window last so everything pops up at once.
241         self.window.show()
242
243 def main():
244     # And of course, our main loop.
245     gtk.main()
246     # Control returns here when main_quit() is called
247     return 0
248
249 if __name__ =="__main__":
250     if len(sys.argv) != 2:
251         sys.stderr.write("usage: packbox.py num, where num is 1, 2, or 3.\n ↩
    ")
252         sys.exit(1)
253     PackBox1(string.atoi(sys.argv[1]))
254     main()
```

A brief tour of the packbox.py code starts with lines 14-50 which define a helper function `make_box()` that creates a horizontal box and populates it with buttons according to the specified parameters. A reference to the horizontal box is returned.

Lines 52-241 define the `PackBox1` class initialization method __init__() that creates a window and a child vertical box that is populated with a different widget arrangement depending on the argument passed to it. If a 1 is passed, lines 75-138 create a window displaying the five unique packing arrangements that are available when varying the homogeneous, expand and fill parameters. If a 2 is passed, lines 140-182 create a window displaying the various combinations of fill with spacing and padding. Finally, if a 3 is passed, lines 188-214 create a window displaying the use of the pack_start() method to left justify the buttons and pack_end() method to right justify a label. Lines 215-235 create a horizontal box containing a button that is packed into the vertical box. The button "clicked" signal is connected to the PyGTK `main_quit`() function to terminate the program.

Lines 250-252 check the command line arguments and exit the program using the `sys.exit`() function if there isn't exactly one argument. Line 253 creates a PackBox1 instance. Line 254 invokes the `main`() function to start the GTK event processing loop.

In this example program, the references to the various widgets (except the window) are not saved in the object instance attributes because they are not needed later.

4.4 Packing Using Tables

Let's take a look at another way of packing - Tables. These can be extremely useful in certain situations.

Using tables, we create a grid that we can place widgets in. The widgets may take up as many spaces as we specify.

The first thing to look at, of course, is the `gtk.Table`() function:

```
table = gtk.Table(rows=1, columns=1, homogeneous=False)
```

The first argument is the number of rows to make in the table, while the second, obviously, is the number of columns.

The *homogeneous* argument has to do with how the table's boxes are sized. If *homogeneous* is `True`, the table boxes are resized to the size of the largest widget in the table. If *homogeneous* is `False`, the size of a table boxes is dictated by the tallest widget in its same row, and the widest widget in its column.

The rows and columns are laid out from 0 to n, where n was the number specified in the call to `gtk.Table`(). So, if you specify rows = 2 and columns = 2, the layout would look something like this:

```
 0          1          2
0+----------+----------+
 |          |          |
1+----------+----------+
 |          |          |
2+----------+----------+
```

Note that the coordinate system starts in the upper left hand corner. To place a widget into a box, use the following method:

```
table.attach(child, left_attach, right_attach, top_attach, bottom_attach,
             xoptions=EXPAND|FILL, yoptions=EXPAND|FILL, xpadding=0, ypadding ↩
                 =0)
```

The table instance is the table you created with `gtk.Table`(). The first parameter ("child") is the widget you wish to place in the table.

The *left_attach*, *right_attach*, *top_attach* and *bottom_attach* arguments specify where to place the widget, and how many boxes to use. If you want a button in the lower right table entry of our 2x2 table, and want it to fill that entry ONLY, *left_attach* would be = 1, *right_attach* = 2, *top_attach* = 1, *bottom_attach* = 2.

Now, if you wanted a widget to take up the whole top row of our 2x2 table, you'd use *left_attach* = 0, *right_attach* = 2, *top_attach* = 0, *bottom_attach* = 1.

The *xoptions* and *yoptions* are used to specify packing options and may be bitwise OR'ed together to allow multiple options.

These options are:

`FILL`	If the table cell is larger than the widget, and `FILL` is specified, the widget will expand to use all the room available in the cell.

SHRINK	If the table widget was allocated less space then was requested (usually by the user resizing the window), then the widgets would normally just be pushed off the bottom of the window and disappear. If SHRINK is specified, the widgets will shrink with the table.
EXPAND	This will cause the table cell to expand to use up any remaining space allocated to the table.

Padding is just like in boxes, creating a clear area around the widget specified in pixels.

We also have set_row_spacing() and set_col_spacing() methods. These add spacing between the rows at the specified row or column.

```
table.set_row_spacing(row, spacing)
```

and

```
table.set_col_spacing(column, spacing)
```

Note that for columns, the space goes to the right of the column, and for rows, the space goes below the row.

You can also set a consistent spacing of all rows and/or columns with:

```
table.set_row_spacings(spacing)
```

and,

```
table.set_col_spacings(spacing)
```

Note that with these calls, the last row and last column do not get any spacing.

4.5 Table Packing Example

The example program table.py makes a window with three buttons in a 2x2 table. The first two buttons will be placed in the upper row. A third, quit button, is placed in the lower row, spanning both columns. Figure 4.6 illustrates the resulting window:

Figure 4.6 Packing using a Table

Here's the source code:

```
 1 #!/usr/bin/env python
 2
 3 # example table.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class Table:
10     # Our callback.
11     # The data passed to this method is printed to stdout
12     def callback(self, widget, data=None):
13         print "Hello again - %s was pressed" % data
```

```
14
15     # This callback quits the program
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return False
19
20     def __init__(self):
21         # Create a new window
22         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         # Set the window title
25         self.window.set_title("Table")
26
27         # Set a handler for delete_event that immediately
28         # exits GTK.
29         self.window.connect("delete_event", self.delete_event)
30
31         # Sets the border width of the window.
32         self.window.set_border_width(20)
33
34         # Create a 2x2 table
35         table = gtk.Table(2, 2, True)
36
37         # Put the table in the main window
38         self.window.add(table)
39
40         # Create first button
41         button = gtk.Button("button 1")
42
43         # When the button is clicked, we call the "callback" method
44         # with a pointer to "button 1" as its argument
45         button.connect("clicked", self.callback, "button 1")
46
47
48         # Insert button 1 into the upper left quadrant of the table
49         table.attach(button, 0, 1, 0, 1)
50
51         button.show()
52
53         # Create second button
54
55         button = gtk.Button("button 2")
56
57         # When the button is clicked, we call the "callback" method
58         # with a pointer to "button 2" as its argument
59         button.connect("clicked", self.callback, "button 2")
60         # Insert button 2 into the upper right quadrant of the table
61         table.attach(button, 1, 2, 0, 1)
62
63         button.show()
64
65         # Create "Quit" button
66         button = gtk.Button("Quit")
67
68         # When the button is clicked, we call the main_quit function
69         # and the program exits
70         button.connect("clicked", lambda w: gtk.main_quit())
71
72         # Insert the quit button into the both lower quadrants of the table
73         table.attach(button, 0, 2, 1, 2)
74
75         button.show()
76
77         table.show()
```

```
78         self.window.show()
79
80 def main():
81     gtk.main()
82     return 0
83
84 if __name__ == "__main__":
85     Table()
86     main()
```

The `Table` class is defined in line 9-78. Lines 12-13 define the callback() method which is called when two of the buttons are "clicked". The callback just prints a message to the console indicating which button was pressed using the passed in string data.

Lines 16-18 define the delete_event() method which is called when the window is slated for deletion by the window manager.

Lines 20-78 define the `Table` instance initialization method __init__() . It creates a window (line 22), sets the window title (line 25), connects the delete_event() callback to the "delete_event" signal (line 29), and sets the border width (line 32). A `gtk.Table` is created in line 35 and added to the window in line 38.

The two upper buttons are created (lines 41 and 55), their "clicked" signals are connected to the callback() method (lines 45 and 59), and attached to the table in the first row (lines 49 and 61). Lines 66-72 create the "Quit" button, connect its "clicked" signal to the `main_quit`() function and attach it to the table spanning the whole second row.

Chapter 5

Widget Overview

The general steps to using a widget in PyGTK are:

- invoke gtk.* - one of various functions to create a new widget. These are all detailed in this section.
- Connect all signals and events we wish to use to the appropriate handlers.
- Set the attributes of the widget.
- Pack the widget into a container using the appropriate call such as gtk.Container.add() or gtk.Box.pack_start() .
- gtk.Widget.show() the widget.

`show`() lets GTK know that we are done setting the attributes of the widget, and it is ready to be displayed. You may also use `gtk.Widget.hide`() to make it disappear again. The order in which you show the widgets is not important, but I suggest showing the window last so the whole window pops up at once rather than seeing the individual widgets come up on the screen as they're formed. The children of a widget (a window is a widget too) will not be displayed until the window itself is shown using the `show`() method.

5.1 Widget Hierarchy

For your reference, here is the class hierarchy tree used to implement widgets. (Deprecated widgets and auxiliary classes have been omitted.)

```
gobject.GObject
|
+gtk.Object
| +gtk.Widget
| | +gtk.Misc
| | | +gtk.Label
| | | | `gtk.AccelLabel
| | | +gtk.Arrow
| | | `gtk.Image
| | +gtk.Container
| | | +gtk.Bin
| | | | +gtk.Alignment
| | | | +gtk.Frame
| | | | | `gtk.AspectFrame
| | | | +gtk.Button
| | | | | +gtk.ToggleButton
| | | | | | `gtk.CheckButton
| | | | | |   `gtk.RadioButton
| | | | | +gtk.ColorButton
| | | | | +gtk.FontButton
| | | | | `gtk.OptionMenu
| | | | +gtk.Item
| | | | | +gtk.MenuItem
```

```
| | | | |   +gtk.CheckMenuItem
| | | | |   | `gtk.RadioMenuItem
| | | | |   +gtk.ImageMenuItem
| | | | |   +gtk.SeparatorMenuItem
| | | | |   `gtk.TearoffMenuItem
| | | | +gtk.Window
| | | | | +gtk.Dialog
| | | | | | +gtk.ColorSelectionDialog
| | | | | | +gtk.FileChooserDialog
| | | | | | +gtk.FileSelection
| | | | | | +gtk.FontSelectionDialog
| | | | | | +gtk.InputDialog
| | | | | | `gtk.MessageDialog
| | | | | `gtk.Plug
| | | | +gtk.ComboBox
| | | | | `gtk.ComboBoxEntry
| | | | +gtk.EventBox
| | | | +gtk.Expander
| | | | +gtk.HandleBox
| | | | +gtk.ToolItem
| | | | | +gtk.ToolButton
| | | | | | +gtk.ToggleToolButton
| | | | | | | `gtk.RadioToolButton
| | | | | `gtk.SeparatorTooItem
| | | | +gtk.ScrolledWindow
| | | | `gtk.Viewport
| | | +gtk.Box
| | | | +gtk.ButtonBox
| | | | | +gtk.HButtonBox
| | | | | `gtk.VButtonBox
| | | | +gtk.VBox
| | | | | +gtk.ColorSelection
| | | | | +gtk.FontSelection
| | | | | `gtk.GammaCurve
| | | | `gtk.HBox
| | | |   +gtk.Combo
| | | |   `gtk.Statusbar
| | | +gtk.Fixed
| | | +gtk.Paned
| | | | +gtk.HPaned
| | | | `gtk.VPaned
| | | +gtk.Layout
| | | +gtk.MenuShell
| | | | +gtk.MenuBar
| | | | `gtk.Menu
| | | +gtk.Notebook
| | | +gtk.Socket
| | | +gtk.Table
| | | +gtk.TextView
| | | +gtk.Toolbar
| | | `gtk.TreeView
| | +gtk.Calendar
| | +gtk.DrawingArea
| | | `gtk.Curve
| | +gtk.Entry
| | | `gtk.SpinButton
| | +gtk.Ruler
| | | +gtk.HRuler
| | | `gtk.VRuler
| | +gtk.Range
| | | +gtk.Scale
| | | | +gtk.HScale
| | | | `gtk.VScale
| | | `gtk.Scrollbar
```

```
| | |   +gtk.HScrollbar
| | |   `gtk.VScrollbar
| | +gtk.Separator
| | | +gtk.HSeparator
| | | `gtk.VSeparator
| | +gtk.Invisible
| | +gtk.Progress
| | | `gtk.ProgressBar
| +gtk.Adjustment
| +gtk.CellRenderer
| | +gtk.CellRendererPixbuf
| | +gtk.CellRendererText
| | +gtk.CellRendererToggle
| +gtk.FileFilter
| +gtk.ItemFactory
| +gtk.Tooltips
| `gtk.TreeViewColumn
+gtk.Action
| +gtk.ToggleAction
| | `gtk.RadioAction
+gtk.ActionGroup
+gtk.EntryCompletion
+gtk.IconFactory
+gtk.IconTheme
+gtk.IMContext
| +gtk.IMContextSimple
| `gtk.IMMulticontext
+gtk.ListStore
+gtk.RcStyle
+gtk.Settings
+gtk.SizeGroup
+gtk.Style
+gtk.TextBuffer
+gtk.TextChildAnchor
+gtk.TextMark
+gtk.TextTag
+gtk.TextTagTable
+gtk.TreeModelFilter
+gtk.TreeModelSort
+gtk.TreeSelection
+gtk.TreeStore
+gtk.UIManager
+gtk.WindowGroup
+gtk.gdk.DragContext
+gtk.gdk.Screen
+gtk.gdk.Pixbuf
+gtk.gdk.Drawable
| +gtk.gdk.Pixmap
+gtk.gdk.Image
+gtk.gdk.PixbufAnimation
+gtk.gdk.Device

gobject.GObject
|
+gtk.CellLayout
+gtk.Editable
+gtk.CellEditable
+gtk.FileChooser
+gtk.TreeModel
+gtk.TreeDragSource
+gtk.TreeDragDest
+gtk.TreeSortable
```

5.2 Widgets Without Windows

The following widgets do not have an associated window. If you want to capture events, you'll have to use the `EventBox`. See the section on the `EventBox` widget.

```
gtk.Alignment
gtk.Arrow
gtk.Bin
gtk.Box
gtk.Button
gtk.CheckButton
gtk.Fixed
gtk.Image
gtk.Label
gtk.MenuItem
gtk.Notebook
gtk.Paned
gtk.RadioButton
gtk.Range
gtk.ScrolledWindow
gtk.Separator
gtk.Table
gtk.Toolbar
gtk.AspectFrame
gtk.Frame
gtk.VBox
gtk.HBox
gtk.VSeparator
gtk.HSeparator
```

We'll further our exploration of PyGTK by examining each widget in turn, creating a few simple example programs to display them.

Chapter 6

The Button Widget

6.1 Normal Buttons

We've almost seen all there is to see of the button widget. It's pretty simple. You can use the `gtk.Button()` function to create a button with a label by passing a string parameter, or to create a blank button by not specifying a label string. It's then up to you to pack a label or pixmap into this new button. To do this, create a new box, and then pack your objects into this box using the usual pack_start() method, and then use the add() method to pack the box into the button.

The function to create a button is:

```
button = gtk.Button(label=None, stock=None)
```

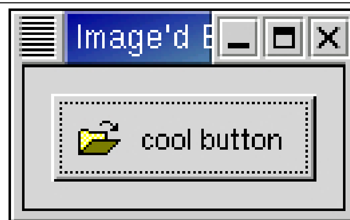
if label text is specified it is used as the text on the button. If stock is specified it is used to select a stock icon and text label for the button. The stock items are:

```
STOCK_DIALOG_INFO
STOCK_DIALOG_WARNING
STOCK_DIALOG_ERROR
STOCK_DIALOG_QUESTION
STOCK_DND
STOCK_DND_MULTIPLE
STOCK_ADD
STOCK_APPLY
STOCK_BOLD
STOCK_CANCEL
STOCK_CDROM
STOCK_CLEAR
STOCK_CLOSE
STOCK_CONVERT
STOCK_COPY
STOCK_CUT
STOCK_DELETE
STOCK_EXECUTE
STOCK_FIND
STOCK_FIND_AND_REPLACE
STOCK_FLOPPY
STOCK_GOTO_BOTTOM
STOCK_GOTO_FIRST
STOCK_GOTO_LAST
STOCK_GOTO_TOP
STOCK_GO_BACK
STOCK_GO_DOWN
STOCK_GO_FORWARD
STOCK_GO_UP
STOCK_HELP
STOCK_HOME
STOCK_INDEX
STOCK_ITALIC
STOCK_JUMP_TO
```

```
STOCK_JUSTIFY_CENTER
STOCK_JUSTIFY_FILL
STOCK_JUSTIFY_LEFT
STOCK_JUSTIFY_RIGHT
STOCK_MISSING_IMAGE
STOCK_NEW
STOCK_NO
STOCK_OK
STOCK_OPEN
STOCK_PASTE
STOCK_PREFERENCES
STOCK_PRINT
STOCK_PRINT_PREVIEW
STOCK_PROPERTIES
STOCK_QUIT
STOCK_REDO
STOCK_REFRESH
STOCK_REMOVE
STOCK_REVERT_TO_SAVED
STOCK_SAVE
STOCK_SAVE_AS
STOCK_SELECT_COLOR
STOCK_SELECT_FONT
STOCK_SORT_ASCENDING
STOCK_SORT_DESCENDING
STOCK_SPELL_CHECK
STOCK_STOP
STOCK_STRIKETHROUGH
STOCK_UNDELETE
STOCK_UNDERLINE
STOCK_UNDO
STOCK_YES
STOCK_ZOOM_100
STOCK_ZOOM_FIT
STOCK_ZOOM_IN
STOCK_ZOOM_OUT
```

The buttons.py program provides an example of using `gtk.Button()` to create a button with an image and a label in it. I've broken up the code to create a box from the rest so you can use it in your programs. There are further examples of using images later in the tutorial. Figure 6.1 shows the window containing a button with both a pixmap and a label:

Figure 6.1 Button with Pixmap and Label



The source code for the buttons.py program is:

```
 1  #!/usr/bin/env python
 2
 3  # example-start buttons buttons.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  # Create a new hbox with an image and a label packed into it
10  # and return the box.
```

```
11
12  def xpm_label_box(parent, xpm_filename, label_text):
13      # Create box for xpm and label
14      box1 = gtk.HBox(False, 0)
15      box1.set_border_width(2)
16
17      # Now on to the image stuff
18      image = gtk.Image()
19      image.set_from_file(xpm_filename)
20
21      # Create a label for the button
22      label = gtk.Label(label_text)
23
24      # Pack the pixmap and label into the box
25      box1.pack_start(image, False, False, 3)
26      box1.pack_start(label, False, False, 3)
27
28      image.show()
29      label.show()
30      return box1
31
32  class Buttons:
33      # Our usual callback method
34      def callback(self, widget, data=None):
35          print "Hello again - %s was pressed" % data
36
37      def __init__(self):
38          # Create a new window
39          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
40
41          self.window.set_title("Image'd Buttons!")
42
43          # It's a good idea to do this for all windows.
44          self.window.connect("destroy", lambda wid: gtk.main_quit())
45          self.window.connect("delete_event", lambda a1,a2:gtk.main_quit())
46
47          # Sets the border width of the window.
48          self.window.set_border_width(10)
49
50          # Create a new button
51          button = gtk.Button()
52
53          # Connect the "clicked" signal of the button to our callback
54          button.connect("clicked", self.callback, "cool button")
55
56          # This calls our box creating function
57          box1 = xpm_label_box(self.window, "info.xpm", "cool button")
58
59          # Pack and show all our widgets
60          button.add(box1)
61
62          box1.show()
63          button.show()
64
65          self.window.add(button)
66          self.window.show()
67
68  def main():
69      gtk.main()
70      return 0
71
72  if __name__ == "__main__":
73      Buttons()
74      main()
```

Lines 12-34 define the `xpm_label_box()` helper function which creates a horizontal box with a border width of 2 (lines 14-15), populates it with an image (lines 22-23) and a label (line 26).

Lines 36-70 define the `Buttons` class. Lines 41-70 define the instance initialization method which creates a window (line 43), sets the title (line 45), connects the "delete_event" and "destroy" signals (lines 48-49). Line 55 creates the button without a label. Its "clicked" signal gets connected to the callback() method in line 58. The `xpm_label_box()` function is called in line 61 to create the image and label to put in the button in line 64.

The `xpm_label_box()` function could be used to pack xpm's and labels into any widget that can be a container.

The Button widget has the following signals:

```
pressed - emitted when pointer button is pressed within Button widget

released - emitted when pointer button is released within Button widget

clicked - emitted when pointer button is pressed and then released within ↩
    Button widget

enter - emitted when pointer enters Button widget

leave - emitted when pointer leaves Button widget
```

6.2 Toggle Buttons

Toggle buttons are derived from normal buttons and are very similar, except they will always be in one of two states, alternated by a click. They may be depressed, and when you click again, they will pop back up. Click again, and they will pop back down.

Toggle buttons are the basis for check buttons and radio buttons, as such, many of the calls used for toggle buttons are inherited by radio and check buttons. I will point these out when we come to them.

Creating a new toggle button:

```
toggle_button = gtk.ToggleButton(label=None)
```

As you can imagine, these work identically to the normal button widget calls. If no label is specified the button will be blank. The label text will be parsed for '_'-prefixed mnemonic characters.

To retrieve the state of the toggle widget, including radio and check buttons, we use a construct as shown in our example below. This tests the state of the toggle, by calling the get_active() method of the toggle button object. The signal of interest to us that is emitted by toggle buttons (the toggle button, check button, and radio button widgets) is the "toggled" signal. To check the state of these buttons, set up a signal handler to catch the toggled signal, and access the object attributes to determine its state. The callback will look something like:

```
def toggle_button_callback(widget, data):
    if widget.get_active():
        # If control reaches here, the toggle button is down
    else:
        # If control reaches here, the toggle button is up
```

To force the state of a toggle button, and its children, the radio and check buttons, use this method:

```
toggle_button.set_active(is_active)
```

The above method can be used to set the state of the toggle button, and its children the radio and check buttons. Specifying a `TRUE` or `FALSE` for the *`is_active`* argument indicates whether the button should be down (depressed) or up (released). When the toggle button is created its default is up or `FALSE`.

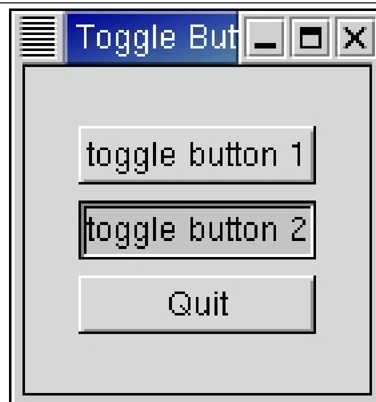
Note that when you use the set_active() method, and the state is actually changed, it causes the "clicked" and "toggled" signals to be emitted from the button.

```
toggle_button.get_active()
```

This method returns the current state of the toggle button as a boolean TRUE or FALSE value.

The togglebutton.py program provides a simple example using toggle buttons. Figure 6.2 illustrates the resulting window with the second toggle button active:

Figure 6.2 Toggle Button Example



The source code for the program is:

```
 1 #!/usr/bin/env python
 2
 3 # example togglebutton.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ToggleButton:
10     # Our callback.
11     # The data passed to this method is printed to stdout
12     def callback(self, widget, data=None):
13         print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active ↩
   ()])
14
15     # This callback quits the program
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return False
19
20     def __init__(self):
21         # Create a new window
22         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         # Set the window title
25         self.window.set_title("Toggle Button")
26
27         # Set a handler for delete_event that immediately
28         # exits GTK.
29         self.window.connect("delete_event", self.delete_event)
30
31         # Sets the border width of the window.
32         self.window.set_border_width(20)
33
34         # Create a vertical box
35         vbox = gtk.VBox(True, 2)
36
37         # Put the vbox in the main window
38         self.window.add(vbox)
39
40         # Create first button
```

```
41         button = gtk.ToggleButton("toggle button 1")
42 
43         # When the button is toggled, we call the "callback" method
44         # with a pointer to "button" as its argument
45         button.connect("toggled", self.callback, "toggle button 1")
46 
47 
48         # Insert button 1
49         vbox.pack_start(button, True, True, 2)
50 
51         button.show()
52 
53         # Create second button
54 
55         button = gtk.ToggleButton("toggle button 2")
56 
57         # When the button is toggled, we call the "callback" method
58         # with a pointer to "button 2" as its argument
59         button.connect("toggled", self.callback, "toggle button 2")
60         # Insert button 2
61         vbox.pack_start(button, True, True, 2)
62 
63         button.show()
64 
65         # Create "Quit" button
66         button = gtk.Button("Quit")
67 
68         # When the button is clicked, we call the main_quit function
69         # and the program exits
70         button.connect("clicked", lambda wid: gtk.main_quit())
71 
72         # Insert the quit button
73         vbox.pack_start(button, True, True, 2)
74 
75         button.show()
76         vbox.show()
77         self.window.show()
78 
79 def main():
80     gtk.main()
81     return 0
82 
83 if __name__ == "__main__":
84     ToggleButton()
85     main()
```

The interesting lines are 12-13 which define the callback() method that prints the toggle button label and its state when it is toggled. Lines 45 and 59 connect the "toggled" signal of the toggle buttons to the callback() method.

6.3 Check Buttons

Check buttons inherit many properties and methods from the the toggle buttons above, but look a little different. Rather than being buttons with text inside them, they are small squares with the text to the right of them. These are often used for toggling options on and off in applications.

The creation method is similar to that of the normal button.

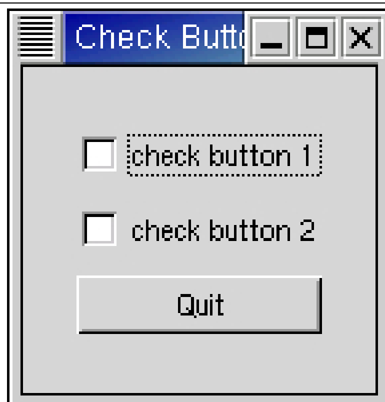
```
check_button = gtk.CheckButton(label=None)
```

If the *label* argument is specified the method creates a check button with a label beside it. The *label* text is parsed for '_'-prefixed mnemonic characters.

Checking and setting the state of the check button are identical to that of the toggle button.

The checkbutton.py program provides an example of the use of the check buttons. Figure 6.3 illustrates the resulting window:

Figure 6.3 Check Button Example



The source code for the checkbutton.py program is:

```
 1 #!/usr/bin/env python
 2
 3 # example checkbutton.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class CheckButton:
10     # Our callback.
11     # The data passed to this method is printed to stdout
12     def callback(self, widget, data=None):
13         print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active ↩
    ()])
14
15     # This callback quits the program
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return False
19
20     def __init__(self):
21         # Create a new window
22         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         # Set the window title
25         self.window.set_title("Check Button")
26
27         # Set a handler for delete_event that immediately
28         # exits GTK.
29         self.window.connect("delete_event", self.delete_event)
30
31         # Sets the border width of the window.
32         self.window.set_border_width(20)
33
34         # Create a vertical box
35         vbox = gtk.VBox(True, 2)
36
37         # Put the vbox in the main window
38         self.window.add(vbox)
39
40         # Create first button
41         button = gtk.CheckButton("check button 1")
```

```
42
43         # When the button is toggled, we call the "callback" method
44         # with a pointer to "button" as its argument
45         button.connect("toggled", self.callback, "check button 1")
46
47
48         # Insert button 1
49         vbox.pack_start(button, True, True, 2)
50
51         button.show()
52
53         # Create second button
54
55         button = gtk.CheckButton("check button 2")
56
57         # When the button is toggled, we call the "callback" method
58         # with a pointer to "button 2" as its argument
59         button.connect("toggled", self.callback, "check button 2")
60         # Insert button 2
61         vbox.pack_start(button, True, True, 2)
62
63         button.show()
64
65         # Create "Quit" button
66         button = gtk.Button("Quit")
67
68         # When the button is clicked, we call the mainquit function
69         # and the program exits
70         button.connect("clicked", lambda wid: gtk.main_quit())
71
72         # Insert the quit button
73         vbox.pack_start(button, True, True, 2)
74
75         button.show()
76         vbox.show()
77         self.window.show()
78
79 def main():
80     gtk.main()
81     return 0
82
83 if __name__ == "__main__":
84     CheckButton()
85     main()
```

6.4 Radio Buttons

Radio buttons are similar to check buttons except they are grouped so that only one may be selected/depressed at a time. This is good for places in your application where you need to select from a short list of options.

Creating a new radio button is done with this call:

```
radio_button = gtk.RadioButton(group=None, label=None)
```

You'll notice the extra argument to this call. Radio buttons require a *group* to operate properly. The first call to `gtk.RadioButton`() should pass `None` as the first argument and a new radio button group will be created with the new radio button as its only member.

To add more radio buttons to a group, pass in a reference to a radio button in *group* in subsequent calls to `gtk.RadioButton`().

If a *label* argument is specified the text will be parsed for '_'-prefixed mnemonic characters.

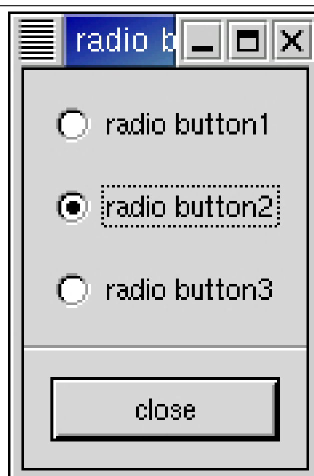
It is also a good idea to explicitly set which button should be the default depressed button with:

```
radio_button.set_active(is_active)
```

This is described in the section on toggle buttons, and works in exactly the same way. Once the radio buttons are grouped together, only one of the group may be active at a time. If the user clicks on one radio button, and then on another, the first radio button will first emit a "toggled" signal (to report becoming inactive), and then the second will emit its "toggled" signal (to report becoming active).

The example program radiobuttons.py creates a radio button group with three buttons. Figure 6.4 illustrates the resulting window:

Figure 6.4 Radio Buttons Example



The source code for the example program is:

```
 1 #!/usr/bin/env python
 2
 3 # example radiobuttons.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class RadioButtons:
10     def callback(self, widget, data=None):
11         print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active ←
    ()])
12
13     def close_application(self, widget, event, data=None):
14         gtk.main_quit()
15         return False
16
17     def __init__(self):
18         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
19
20         self.window.connect("delete_event", self.close_application)
21
22         self.window.set_title("radio buttons")
23         self.window.set_border_width(0)
24
25         box1 = gtk.VBox(False, 0)
26         self.window.add(box1)
27         box1.show()
28
29         box2 = gtk.VBox(False, 10)
30         box2.set_border_width(10)
31         box1.pack_start(box2, True, True, 0)
32         box2.show()
```

```
33
34         button = gtk.RadioButton(None, "radio button1")
35         button.connect("toggled", self.callback, "radio button 1")
36         box2.pack_start(button, True, True, 0)
37         button.show()
38
39         button = gtk.RadioButton(button, "radio button2")
40         button.connect("toggled", self.callback, "radio button 2")
41         button.set_active(True)
42         box2.pack_start(button, True, True, 0)
43         button.show()
44
45         button = gtk.RadioButton(button, "radio button3")
46         button.connect("toggled", self.callback, "radio button 3")
47         box2.pack_start(button, True, True, 0)
48         button.show()
49
50         separator = gtk.HSeparator()
51         box1.pack_start(separator, False, True, 0)
52         separator.show()
53
54         box2 = gtk.VBox(False, 10)
55         box2.set_border_width(10)
56         box1.pack_start(box2, False, True, 0)
57         box2.show()
58
59         button = gtk.Button("close")
60         button.connect_object("clicked", self.close_application, self. ↩
    window,
61                               None)
62         box2.pack_start(button, True, True, 0)
63         button.set_flags(gtk.CAN_DEFAULT)
64         button.grab_default()
65         button.show()
66         self.window.show()
67
68 def main():
69     gtk.main()
70     return 0
71
72 if __name__ == "__main__":
73     RadioButtons()
74     main()
```

The code is fairly straight forward. Lines 63-64 make the "close" button the default widget so that pressing the "Enter" key when the window is active causes the "close" button to emit the "clicked" signal.

Chapter 7

Adjustments

GTK has various widgets that can be visually adjusted by the user using the mouse or the keyboard, such as the range widgets, described in the Range Widgets section. There are also a few widgets that display some adjustable portion of a larger area of data, such as the text widget and the viewport widget.

Obviously, an application needs to be able to react to changes the user makes in range widgets. One way to do this would be to have each widget emit its own type of signal when its adjustment changes, and either pass the new value to the signal handler, or require it to look inside the widget's data structure in order to ascertain the value. But you may also want to connect the adjustments of several widgets together, so that adjusting one adjusts the others. The most obvious example of this is connecting a scrollbar to a panning viewport or a scrolling text area. If each widget has its own way of setting or getting the adjustment value, then the programmer may have to write their own signal handlers to translate between the output of one widget's signal and the "input" of another's adjustment setting method.

GTK solves this problem using the `Adjustment` object, which is not a widget but a way for widgets to store and pass adjustment information in an abstract and flexible form. The most obvious use of `Adjustment` is to store the configuration parameters and values of range widgets, such as scrollbars and scale controls. However, since `Adjustment`s are derived from `Object`, they have some special powers beyond those of normal data structures. Most importantly, they can emit signals, just like widgets, and these signals can be used not only to allow your program to react to user input on adjustable widgets, but also to propagate adjustment values transparently between adjustable widgets.

You will see how adjustments fit in when you see the other widgets that incorporate them: Progress Bars, Viewports, Scrolled Windows, and others.

7.1 Creating an Adjustment

Many of the widgets which use adjustment objects do so automatically, but some cases will be shown in later examples where you may need to create one yourself. You create an adjustment using:

```
adjustment = gtk.Adjustment(value=0, lower=0, upper=0, step_incr=0, page_incr ↩
    =0, page_size=0)
```

The *`value`* argument is the initial value you want to give to the *`adjustment`*, usually corresponding to the topmost or leftmost position of an adjustable widget. The *`lower`* argument specifies the lowest value which the *`adjustment`* can hold. The *`step_incr`* argument specifies the "smaller" of the two increments by which the user can change the value, while the *`page_incr`* is the "larger" one. The *`page_size`* argument usually corresponds somehow to the visible area of a panning widget. The *`upper`* argument is used to represent the bottom most or right most coordinate in a panning widget's child. Therefore it is not always the largest number that *`value`* can take, since the *`page_size`* of such widgets is usually non-zero.

7.2 Using Adjustments the Easy Way

The adjustable widgets can be roughly divided into those which use and require specific units for these values, and those which treat them as arbitrary numbers. The group which treats the values as arbitrary

numbers includes the range widgets (scrollbars and scales, the progress bar widget, and the spin button widget). These widgets are all the widgets which are typically "adjusted" directly by the user with the mouse or keyboard. They will treat the lower and upper values of an adjustment as a range within which the user can manipulate the adjustment's value. By default, they will only modify the value of an adjustment.

The other group includes the text widget, the viewport widget, the compound list widget, and the scrolled window widget. All of these widgets use pixel values for their adjustments. These are also all widgets which are typically "adjusted" indirectly using scrollbars. While all widgets which use adjustments can either create their own adjustments or use ones you supply, you'll generally want to let this particular category of widgets create its own adjustments. Usually, they will eventually override all the values except the value itself in whatever adjustments you give them, but the results are, in general, undefined (meaning, you'll have to read the source code to find out, and it may be different from widget to widget).

Now, you're probably thinking, since text widgets and viewports insist on setting everything except the value of their adjustments, while scrollbars will only touch the adjustment's value, if you share an adjustment object between a scrollbar and a text widget, manipulating the scrollbar will automagically adjust the text widget? Of course it will! Just like this:

```
# creates its own adjustments
viewport = gtk.Viewport()
# uses the newly-created adjustment for the scrollbar as well
vscrollbar = gtk.VScrollbar(viewport.get_vadjustment())
```

7.3 Adjustment Internals

Ok, you say, that's nice, but what if I want to create my own handlers to respond when the user adjusts a range widget or a spin button, and how do I get at the value of the adjustment in these handlers? To answer these questions and more, let's start by taking a look at the attributes of a `gtk.Adjustment` itself:

```
lower
upper
value
step_increment
page_increment
page_size
```

Given a `gtk.Adjustment` instance *`adj`*, each of the attributes are retrieved or set by *`adj.lower`*, *`adj.value`*, etc.

Since, when you set the value of an adjustment, you generally want the change to be reflected by every widget that uses this adjustment, PyGTK provides a method to do this:

```
adjustment.set_value(value)
```

As mentioned earlier, `Adjustment` is a subclass of `Object` just like all the various widgets, and thus it is able to emit signals. This is, of course, why updates happen automagically when you share an adjustment object between a scrollbar and another adjustable widget; all adjustable widgets connect signal handlers to their adjustment's value_changed signal, as can your program. Here's the definition of this signal callback:

```
def value_changed(adjustment):
```

The various widgets that use the `Adjustment` object will emit this signal on an adjustment whenever they change its value. This happens both when user input causes the slider to move on a range widget, as well as when the program explicitly changes the value with the set_value() method. So, for example, if you have a scale widget, and you want to change the rotation of a picture whenever its value changes, you would create a callback like this:

```
def cb_rotate_picture(adj, picture):
    set_picture_rotation (picture, adj.value)
...
```

and connect it to the scale widget's adjustment like this:

```
adj.connect("value_changed", cb_rotate_picture, picture)
```

What about when a widget reconfigures the *upper* or *lower* fields of its adjustment, such as when a user adds more text to a text widget? In this case, it emits the changed signal, which looks like this:

```
def changed(adjustment):
```

Range widgets typically connect a handler to this signal, which changes their appearance to reflect the change - for example, the size of the slider in a scrollbar will grow or shrink in inverse proportion to the difference between the lower and upper values of its adjustment.

You probably won't ever need to attach a handler to this signal, unless you're writing a new type of range widget. However, if you change any of the values in a Adjustment directly, you should emit this signal on it to reconfigure whatever widgets are using it, like this:

```
adjustment.emit("changed")
```

Chapter 8

Range Widgets

The category of range widgets includes the ubiquitous scrollbar widget and the less common "scale" widget. Though these two types of widgets are generally used for different purposes, they are quite similar in function and implementation. All range widgets share a set of common graphic elements, each of which has its own X window and receives events. They all contain a "trough" and a "slider" (what is sometimes called a "thumbwheel" in other GUI environments). Dragging the slider with the pointer moves it back and forth within the trough, while clicking in the trough advances the slider towards the location of the click, either completely, or by a designated amount, depending on which mouse button is used.

As mentioned in Chapter 7 above, all range widgets are associated with an `Adjustment` object, from which they calculate the length of the slider and its position within the trough. When the user manipulates the slider, the range widget will change the value of the adjustment.

8.1 Scrollbar Widgets

These are your standard, run-of-the-mill scrollbars. These should be used only for scrolling some other widget, such as a list, a text box, or a viewport (and it's generally easier to use the scrolled window widget in most cases). For other purposes, you should use scale widgets, as they are friendlier and more featureful.

There are separate types for horizontal and vertical scrollbars. There really isn't much to say about these. You create them with the following methods:

```
hscrollbar = gtk.HSscrollbar(adjustment=None)

vscrollbar = gtk.VSscrollbar(adjustment=None)
```

and that's about it. The *`adjustment`* argument can either be a reference to an existing `Adjustment` (see Chapter 7), or nothing, in which case one will be created for you. Specifying nothing might be useful in the case, where you wish to pass the newly-created adjustment to the constructor function of some other widget which will configure it for you, such as a text widget.

8.2 Scale Widgets

`Scale` widgets are used to allow the user to visually select and manipulate a value within a specific range. You might want to use a scale widget, for example, to adjust the magnification level on a zoomed preview of a picture, or to control the brightness of a color, or to specify the number of minutes of inactivity before a screensaver takes over the screen.

8.2.1 Creating a Scale Widget

As with scrollbars, there are separate widget types for horizontal and vertical scale widgets. (Most programmers seem to favour horizontal scale widgets.) Since they work essentially the same way, there's no need to treat them separately here. The following methods create vertical and horizontal scale widgets, respectively:

```
vscale = gtk.VScale(adjustment=None)

hscale = gtk.HScale(adjustment=None)
```

The *adjustment* argument can either be an adjustment which has already been created with gtk.-Adjustment() , or nothing, in which case, an anonymous Adjustment is created with all of its values set to 0.0 (which isn't very useful in this case). In order to avoid confusing yourself, you probably want to create your adjustment with a *page_size* of 0.0 so that its *upper* value actually corresponds to the highest value the user can select. (If you're already thoroughly confused, read Chapter 7 again for an explanation of what exactly adjustments do and how to create and manipulate them.)

8.2.2 Methods and Signals (well, methods, at least)

Scale widgets can display their current value as a number beside the trough. The default behaviour is to show the value, but you can change this with this method:

```
scale.set_draw_value(draw_value)
```

As you might have guessed, *draw_value* is either TRUE or FALSE, with predictable consequences for either one.

The value displayed by a scale widget is rounded to one decimal point by default, as is the value field in its Adjustment. You can change this with:

```
scale.set_digits(digits)
```

where *digits* is the number of decimal places you want. You can set digits to anything you like, but no more than 13 decimal places will actually be drawn on screen.

Finally, the value can be drawn in different positions relative to the trough:

```
scale.set_value_pos(pos)
```

The argument *pos* can take one of the following values:

```
POS_LEFT
POS_RIGHT
POS_TOP
POS_BOTTOM
```

If you position the value on the "side" of the trough (e.g., on the top or bottom of a horizontal scale widget), then it will follow the slider up and down the trough.

8.3 Common Range Methods

The Range widget class is fairly complicated internally, but, like all the "base class" widgets, most of its complexity is only interesting if you want to hack on it. Also, almost all of the methods and signals it defines are only really used in writing derived widgets. There are, however, a few useful methods that will work on all range widgets.

8.3.1 Setting the Update Policy

The "update policy" of a range widget defines at what points during user interaction it will change the value field of its Adjustment and emit the "value_changed" signal on this Adjustment. The update policies are:

UPDATE_CONTINUOUS This is the default. The "value_changed" signal is emitted continuously, i.e., whenever the slider is moved by even the tiniest amount.

UPDATE_DISCONTINUOUS The "value_changed" signal is only emitted once the slider has stopped moving and the user has released the mouse button.

UPDATE_DELAYED The "value_changed" signal is emitted when the user releases the mouse button, or if the slider stops moving for a short period of time.

The update policy of a range widget can be set by passing it to this method:

```
range.set_update_policy(policy)
```

8.3.2 Getting and Setting Adjustments

Getting and setting the adjustment for a range widget "on the fly" is done, predictably, with:

```
adjustment = range.get_adjustment()

range.set_adjustment(adjustment)
```

The get_adjustment() method returns a reference to the *adjustment* to which range is connected.

The set_adjustment() method does absolutely nothing if you pass it the *adjustment* that *range* is already using, regardless of whether you changed any of its fields or not. If you pass it a new Adjustment, it will unreference the old one if it exists (possibly destroying it), connect the appropriate signals to the new one, and will recalculate the size and/or position of the slider and redraw if necessary. As mentioned in the section on adjustments, if you wish to reuse the same Adjustment, when you modify its values directly, you should emit the "changed" signal on it, like this:

```
adjustment.emit("changed")
```

8.4 Key and Mouse Bindings

All of the GTK+ range widgets react to mouse clicks in more or less the same way. Clicking button-1 in the trough will cause its adjustment's *page_increment* to be added or subtracted from its *value*, and the slider to be moved accordingly. Clicking mouse button-2 in the trough will jump the slider to the point at which the button was clicked. Clicking any button on a scrollbar's arrows will cause its adjustment's value to change *step_increment* at a time.

Scrollbars are not focusable, thus have no key bindings. The key bindings for the other range widgets (which are, of course, only active when the widget has focus) do not differentiate between horizontal and vertical range widgets.

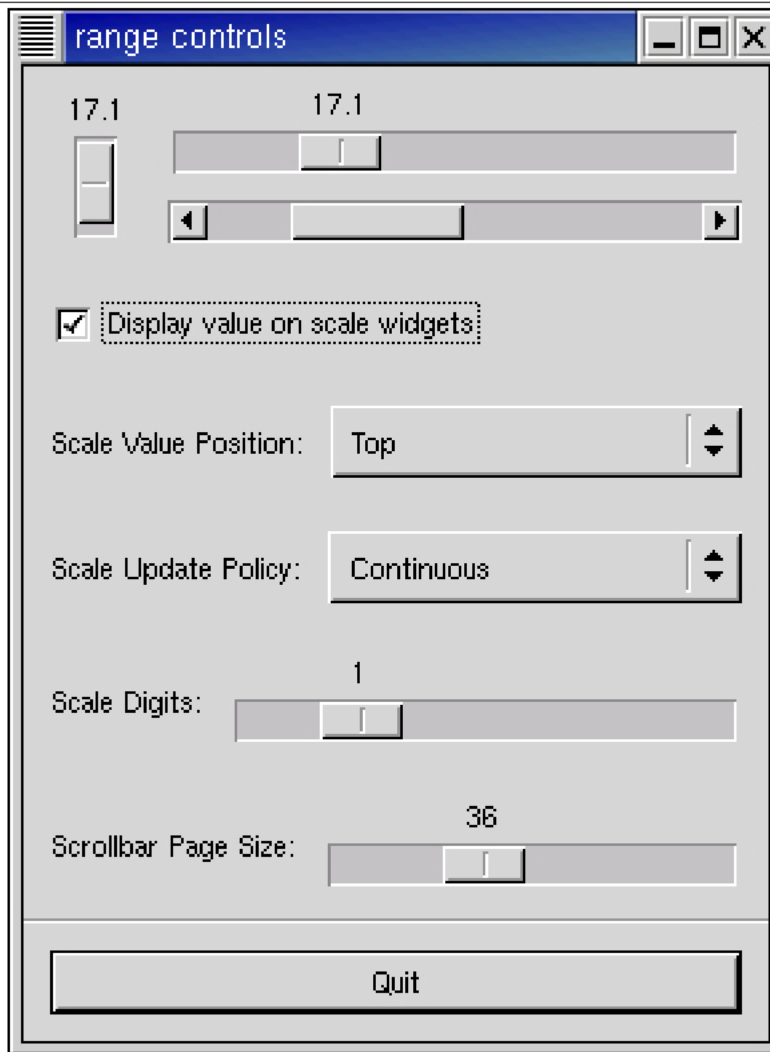
All range widgets can be operated with the left arrow, right arrow, up arrow and down arrow keys, as well as with the **Page Up** and **Page Down** keys. The arrows move the slider by *step_increment*, while **Page Up** and **Page Down** move it by *page_increment*.

The user can also move the slider all the way to one end or the other of the trough using the keyboard. This is done with the **Home** and **End** keys.

8.5 Range Widget Example

The example program (rangewidgets.py) puts up a window with three range widgets all connected to the same adjustment, and a couple of controls for adjusting some of the parameters mentioned above and in the section on adjustments, so you can see how they affect the way these widgets work for the user. Figure 8.1 illustrates the result of running the program:

Figure 8.1 Range Widgets Example



The rangewidgets.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example rangewidgets.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # Convenience functions
10
11 def make_menu_item(name, callback, data=None):
12     item = gtk.MenuItem(name)
13     item.connect("activate", callback, data)
14     item.show()
15     return item
16
17 def scale_set_default_values(scale):
18     scale.set_update_policy(gtk.UPDATE_CONTINUOUS)
19     scale.set_digits(1)
20     scale.set_value_pos(gtk.POS_TOP)
21     scale.set_draw_value(True)
22
23 class RangeWidgets:
24     def cb_pos_menu_select(self, item, pos):
```

```
25          # Set the value position on both scale widgets
26          self.hscale.set_value_pos(pos)
27          self.vscale.set_value_pos(pos)
28
29      def cb_update_menu_select(self, item, policy):
30          # Set the update policy for both scale widgets
31          self.hscale.set_update_policy(policy)
32          self.vscale.set_update_policy(policy)
33
34      def cb_digits_scale(self, adj):
35          # Set the number of decimal places to which adj->value is rounded
36          self.hscale.set_digits(adj.value)
37          self.vscale.set_digits(adj.value)
38
39      def cb_page_size(self, get, set):
40          # Set the page size and page increment size of the sample
41          # adjustment to the value specified by the "Page Size" scale
42          set.page_size = get.value
43          set.page_incr = get.value
44          # Now emit the "changed" signal to reconfigure all the widgets that
45          # are attached to this adjustment
46          set.emit("changed")
47
48      def cb_draw_value(self, button):
49          # Turn the value display on the scale widgets off or on depending
50          # on the state of the checkbutton
51          self.hscale.set_draw_value(button.get_active())
52          self.vscale.set_draw_value(button.get_active())
53
54      # makes the sample window
55
56      def __init__(self):
57          # Standard window-creating stuff
58          self.window = gtk.Window (gtk.WINDOW_TOPLEVEL)
59          self.window.connect("destroy", lambda w: gtk.main_quit())
60          self.window.set_title("range controls")
61
62          box1 = gtk.VBox(False, 0)
63          self.window.add(box1)
64          box1.show()
65
66          box2 = gtk.HBox(False, 10)
67          box2.set_border_width(10)
68          box1.pack_start(box2, True, True, 0)
69          box2.show()
70
71          # value, lower, upper, step_increment, page_increment, page_size
72          # Note that the page_size value only makes a difference for
73          # scrollbar widgets, and the highest value you'll get is actually
74          # (upper - page_size).
75          adj1 = gtk.Adjustment(0.0, 0.0, 101.0, 0.1, 1.0, 1.0)
76
77          self.vscale = gtk.VScale(adj1)
78          scale_set_default_values(self.vscale)
79          box2.pack_start(self.vscale, True, True, 0)
80          self.vscale.show()
81
82          box3 = gtk.VBox(False, 10)
83          box2.pack_start(box3, True, True, 0)
84          box3.show()
85
86          # Reuse the same adjustment
87          self.hscale = gtk.HScale(adj1)
88          self.hscale.set_size_request(200, 30)
```

```
 89         scale_set_default_values(self.hscale)
 90         box3.pack_start(self.hscale, True, True, 0)
 91         self.hscale.show()
 92 
 93         # Reuse the same adjustment again
 94         scrollbar = gtk.HScrollbar(adj1)
 95         # Notice how this causes the scales to always be updated
 96         # continuously when the scrollbar is moved
 97         scrollbar.set_update_policy(gtk.UPDATE_CONTINUOUS)
 98         box3.pack_start(scrollbar, True, True, 0)
 99         scrollbar.show()
100 
101         box2 = gtk.HBox(False, 10)
102         box2.set_border_width(10)
103         box1.pack_start(box2, True, True, 0)
104         box2.show()
105 
106         # A checkbutton to control whether the value is displayed or not
107         button = gtk.CheckButton("Display value on scale widgets")
108         button.set_active(True)
109         button.connect("toggled", self.cb_draw_value)
110         box2.pack_start(button, True, True, 0)
111         button.show()
112 
113         box2 = gtk.HBox(False, 10)
114         box2.set_border_width(10)
115 
116         # An option menu to change the position of the value
117         label = gtk.Label("Scale Value Position:")
118         box2.pack_start(label, False, False, 0)
119         label.show()
120 
121         opt = gtk.OptionMenu()
122         menu = gtk.Menu()
123 
124         item = make_menu_item ("Top", self.cb_pos_menu_select, gtk.POS_TOP)
125         menu.append(item)
126 
127         item = make_menu_item ("Bottom", self.cb_pos_menu_select,
128                                gtk.POS_BOTTOM)
129         menu.append(item)
130 
131         item = make_menu_item ("Left", self.cb_pos_menu_select, gtk. ↩
    POS_LEFT)
132         menu.append(item)
133 
134         item = make_menu_item ("Right", self.cb_pos_menu_select, gtk. ↩
    POS_RIGHT)
135         menu.append(item)
136 
137         opt.set_menu(menu)
138         box2.pack_start(opt, True, True, 0)
139         opt.show()
140 
141         box1.pack_start(box2, True, True, 0)
142         box2.show()
143 
144         box2 = gtk.HBox(False, 10)
145         box2.set_border_width(10)
146 
147         # Yet another option menu, this time for the update policy of the
148         # scale widgets
149         label = gtk.Label("Scale Update Policy:")
150         box2.pack_start(label, False, False, 0)
```

```
151         label.show()
152 
153         opt = gtk.OptionMenu()
154         menu = gtk.Menu()
155 
156         item = make_menu_item("Continuous", self.cb_update_menu_select,
157                               gtk.UPDATE_CONTINUOUS)
158         menu.append(item)
159 
160         item = make_menu_item ("Discontinuous", self.cb_update_menu_select,
161                                gtk.UPDATE_DISCONTINUOUS)
162         menu.append(item)
163 
164         item = make_menu_item ("Delayed", self.cb_update_menu_select,
165                                gtk.UPDATE_DELAYED)
166         menu.append(item)
167 
168         opt.set_menu(menu)
169         box2.pack_start(opt, True, True, 0)
170         opt.show()
171 
172         box1.pack_start(box2, True, True, 0)
173         box2.show()
174 
175         box2 = gtk.HBox(False, 10)
176         box2.set_border_width(10)
177 
178         # An HScale widget for adjusting the number of digits on the
179         # sample scales.
180         label = gtk.Label("Scale Digits:")
181         box2.pack_start(label, False, False, 0)
182         label.show()
183 
184         adj2 = gtk.Adjustment(1.0, 0.0, 5.0, 1.0, 1.0, 0.0)
185         adj2.connect("value_changed", self.cb_digits_scale)
186         scale = gtk.HScale(adj2)
187         scale.set_digits(0)
188         box2.pack_start(scale, True, True, 0)
189         scale.show()
190 
191         box1.pack_start(box2, True, True, 0)
192         box2.show()
193 
194         box2 = gtk.HBox(False, 10)
195         box2.set_border_width(10)
196 
197         # And, one last HScale widget for adjusting the page size of the
198         # scrollbar.
199         label = gtk.Label("Scrollbar Page Size:")
200         box2.pack_start(label, False, False, 0)
201         label.show()
202 
203         adj2 = gtk.Adjustment(1.0, 1.0, 101.0, 1.0, 1.0, 0.0)
204         adj2.connect("value_changed", self.cb_page_size, adj1)
205         scale = gtk.HScale(adj2)
206         scale.set_digits(0)
207         box2.pack_start(scale, True, True, 0)
208         scale.show()
209 
210         box1.pack_start(box2, True, True, 0)
211         box2.show()
212 
213         separator = gtk.HSeparator()
214         box1.pack_start(separator, False, True, 0)
```

```
215         separator.show()
216 
217         box2 = gtk.VBox(False, 10)
218         box2.set_border_width(10)
219         box1.pack_start(box2, False, True, 0)
220         box2.show()
221 
222         button = gtk.Button("Quit")
223         button.connect("clicked", lambda w: gtk.main_quit())
224         box2.pack_start(button, True, True, 0)
225         button.set_flags(gtk.CAN_DEFAULT)
226         button.grab_default()
227         button.show()
228         self.window.show()
229 
230 def main():
231     gtk.main()
232     return 0
233 
234 if __name__ == "__main__":
235     RangeWidgets()
236     main()
```

You will notice that the program does not call the connect() method for the "delete_event", but only for the "destroy" signal. This will still perform the desired operation, because an unhandled "delete_event" will result in a "destroy" signal being given to the window.

Chapter 9

Miscellaneous Widgets

9.1 Labels

`Labels` are used a lot in GTK, and are relatively simple. `Labels` emit no signals as they do not have an associated X window. If you need to catch signals, or do clipping, place it inside a `EventBox` (see Section 10.1) widget or a `Button` (see Section 6.1) widget.

To create a new label, use:

```
label = gtk.Label(str)
```

The sole argument is the string you wish the label to display. To change the label's text after creation, use the method:

```
label.set_text(str)
```

label is the label you created previously, and *str* is the new string. The space needed for the new string will be automatically adjusted if needed. You can produce multi-line labels by putting line breaks in the label string.

To retrieve the current string, use:

```
str = label.get_text()
```

label is the label you've created, and *str* is the return string. The *label* text can be justified using:

```
label.set_justify(jtype)
```

Values for *jtype* are:

```
JUSTIFY_LEFT # the default
JUSTIFY_RIGHT
JUSTIFY_CENTER
JUSTIFY_FILL # does not work
```

The label widget is also capable of line wrapping the text automatically. This can be activated using:

```
label.set_line_wrap(wrap)
```

The *wrap* argument takes a `TRUE` or `FALSE` value.

If you want your label underlined, then you can set a pattern on the label:

```
label.set_pattern(pattern)
```

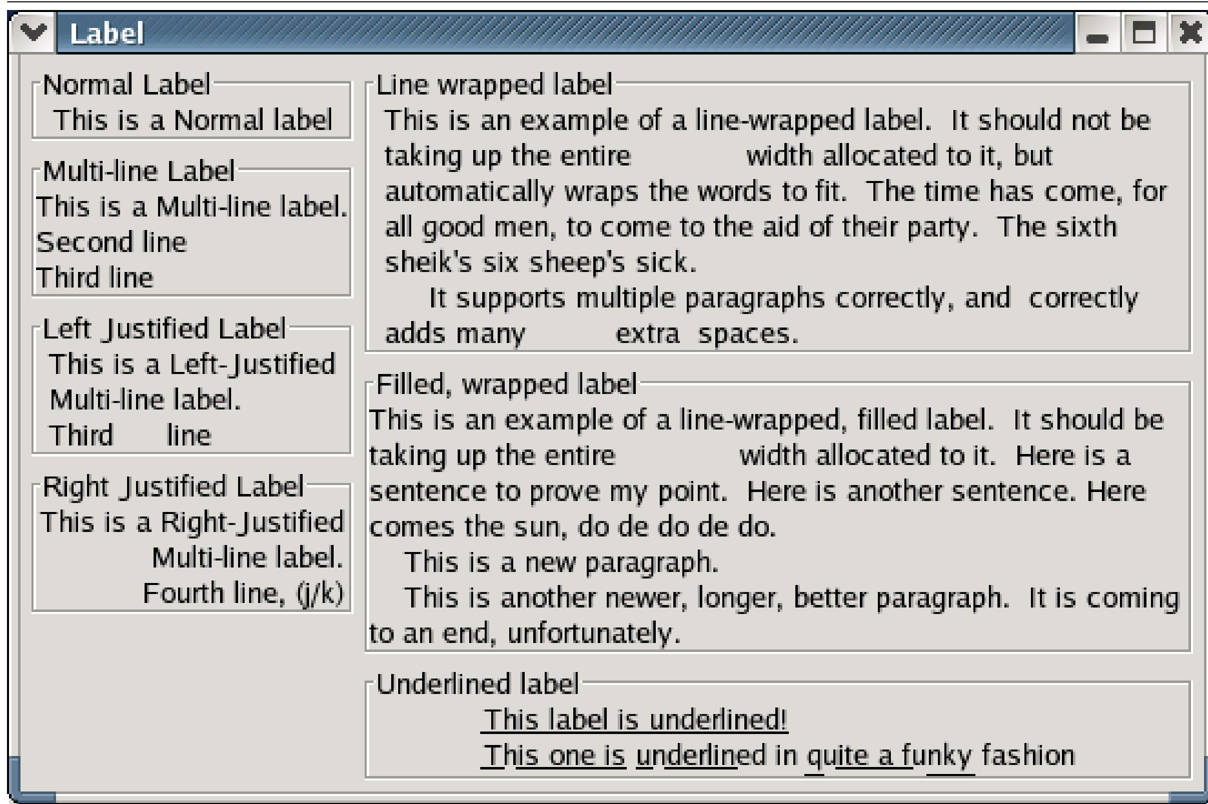
The *pattern* argument indicates how the underlining should look. It consists of a string of underscore and space characters. An underscore indicates that the corresponding character in the label should be underlined. For example, the string `"__ __"` would underline the first two characters and fourth and fifth characters. If you simply want to have an underlined accelerator ("mnemonic") in your label, you should use set_text_with_mnemonic(*str*), not set_pattern().

The label.py program is a short example to illustrate these methods. This example makes use of the `Frame` (see Section 10.5) widget to better demonstrate the label styles. You can ignore this for now as the `Frame` widget is explained later on.

In GTK+ 2.0, label text can contain markup for font and other text attribute changes, and labels may be selectable (for copy-and-paste). These advanced features won't be explained here.

Figure 9.1 illustrates the result of running the example program:

Figure 9.1 Label Examples



The label.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example label.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class Labels:
10     def __init__(self):
11         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
12         self.window.connect("destroy", lambda w: gtk.main_quit())
13
14         self.window.set_title("Label")
15         vbox = gtk.VBox(False, 5)
16         hbox = gtk.HBox(False, 5)
17         self.window.add(hbox)
18         hbox.pack_start(vbox, False, False, 0)
19         self.window.set_border_width(5)
20
21         frame = gtk.Frame("Normal Label")
22         label = gtk.Label("This is a Normal label")
23         frame.add(label)
24         vbox.pack_start(frame, False, False, 0)
25
26         frame = gtk.Frame("Multi-line Label")
27         label = gtk.Label("This is a Multi-line label.\nSecond line\n"
28                           "Third line")
```

```
29          frame.add(label)
30          vbox.pack_start(frame, False, False, 0)
31
32          frame = gtk.Frame("Left Justified Label")
33          label = gtk.Label("This is a Left-Justified\n"
34                            "Multi-line label.\nThird      line")
35          label.set_justify(gtk.JUSTIFY_LEFT)
36          frame.add(label)
37          vbox.pack_start(frame, False, False, 0)
38
39          frame = gtk.Frame("Right Justified Label")
40          label = gtk.Label("This is a Right-Justified\nMulti-line label.\n"
41                            "Fourth line, (j/k)")
42          label.set_justify(gtk.JUSTIFY_RIGHT)
43          frame.add(label)
44          vbox.pack_start(frame, False, False, 0)
45
46          vbox = gtk.VBox(False, 5)
47          hbox.pack_start(vbox, False, False, 0)
48          frame = gtk.Frame("Line wrapped label")
49          label = gtk.Label("This is an example of a line-wrapped label.  It  ↩
   "
50                            "should not be taking up the entire  ↩
            "
51                            "width allocated to it, but automatically "
52                            "wraps the words to fit.  "
53                            "The time has come, for all good men, to come  ↩
   to "
54                            "the aid of their party.  "
55                            "The sixth sheik's six sheep's sick.\n"
56                            "     It supports multiple paragraphs  ↩
   correctly, "
57                            "and  correctly   adds "
58                            "many          extra  spaces. ")
59          label.set_line_wrap(True)
60          frame.add(label)
61          vbox.pack_start(frame, False, False, 0)
62
63          frame = gtk.Frame("Filled, wrapped label")
64          label = gtk.Label("This is an example of a line-wrapped, filled  ↩
   label.  "
65                            "It should be taking "
66                            "up the entire              width allocated to ↩
    it.  "
67                            "Here is a sentence to prove "
68                            "my point.  Here is another sentence. "
69                            "Here comes the sun, do de do de do.\n"
70                            "    This is a new paragraph.\n"
71                            "    This is another newer, longer, better "
72                            "paragraph.  It is coming to an end, "
73                            "unfortunately.")
74          label.set_justify(gtk.JUSTIFY_FILL)
75          label.set_line_wrap(True)
76          frame.add(label)
77          vbox.pack_start(frame, False, False, 0)
78
79          frame = gtk.Frame("Underlined label")
80          label = gtk.Label("This label is underlined!\n"
81                            "This one is underlined in quite a funky  ↩
   fashion")
82          label.set_justify(gtk.JUSTIFY_LEFT)
83          label.set_pattern(
84              "_________________________ _ _________ _ ______     __ _______  ↩
   ___")
```

```
85          frame.add(label)
86          vbox.pack_start(frame, False, False, 0)
87          self.window.show_all ()
88
89 def main():
90     gtk.main()
91     return 0
92
93 if __name__ == "__main__":
94     Labels()
95     main()
```

Note that the "Filled, wrapped label" is not fill justified.

9.2 Arrows

The `Arrow` widget draws an arrowhead, facing in a number of possible directions and having a number of possible styles. It can be very useful when placed on a button in many applications. Like the `Label` widget, it emits no signals.

There are only two calls for manipulating an `Arrow` widget:

```
arrow = gtk.Arrow(arrow_type, shadow_type)

arrow.set(arrow_type, shadow_type)
```

The first creates a new arrow widget with the indicated type and appearance. The second allows these values to be altered retrospectively. The *`arrow_type`* argument may take one of the following values:

```
ARROW_UP
ARROW_DOWN
ARROW_LEFT
ARROW_RIGHT
```

These values obviously indicate the direction in which the arrow will point. The *`shadow_type`* argument may take one of these values:

```
SHADOW_IN
SHADOW_OUT    # the default
SHADOW_ETCHED_IN
SHADOW_ETCHED_OUT
```

The arrow.py example program briefly illustrates their use. Figure 9.2 illustrates the result of running the program:

Figure 9.2 Arrows Buttons Examples

The source code for arrow.py is:

```
1 #!/usr/bin/env python
2
3 # example arrow.py
4
5 import pygtk
6 pygtk.require('2.0')
7 import gtk
```

```
 8
 9 # Create an Arrow widget with the specified parameters
10 # and pack it into a button
11 def create_arrow_button(arrow_type, shadow_type):
12     button = gtk.Button();
13     arrow = gtk.Arrow(arrow_type, shadow_type);
14     button.add(arrow)
15     button.show()
16     arrow.show()
17     return button
18
19 class Arrows:
20     def __init__(self):
21         # Create a new window
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         window.set_title("Arrow Buttons")
25
26         # It’s a good idea to do this for all windows.
27         window.connect("destroy", lambda x: gtk.main_quit())
28
29         # Sets the border width of the window.
30         window.set_border_width(10)
31
32         # Create a box to hold the arrows/buttons
33         box = gtk.HBox(False, 0)
34         box.set_border_width(2)
35         window.add(box)
36
37         # Pack and show all our widgets
38         box.show()
39
40         button = create_arrow_button(gtk.ARROW_UP, gtk.SHADOW_IN)
41         box.pack_start(button, False, False, 3)
42
43         button = create_arrow_button(gtk.ARROW_DOWN, gtk.SHADOW_OUT)
44         box.pack_start(button, False, False, 3)
45
46         button = create_arrow_button(gtk.ARROW_LEFT, gtk.SHADOW_ETCHED_IN)
47         box.pack_start(button, False, False, 3)
48
49         button = create_arrow_button(gtk.ARROW_RIGHT, gtk.SHADOW_ETCHED_OUT ↩
   )
50         box.pack_start(button, False, False, 3)
51
52         window.show()
53
54 def main():
55     gtk.main()
56     return 0
57
58 if __name__ == "__main__":
59     Arrows()
60     main()
```

9.3 The Tooltips Object

`Tooltips` are the little text strings that pop up when you leave your pointer over a button or other widget for a few seconds.

Widgets that do not receive events (widgets that do not have their own window) will not work with tooltips.

The first call you will use creates a new tooltip. You only need to do this once for a set of tooltips as the `gtk.Tooltips` object this function returns can be used to create multiple tooltips.

```
tooltips = gtk.Tooltips()
```

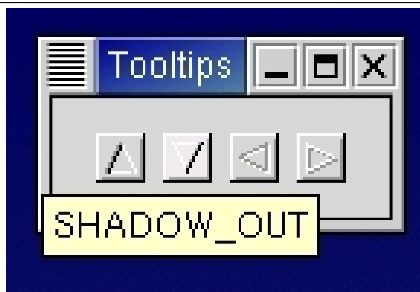
Once you have created a new tooltip, and the widget you wish to use it on, simply use this call to set it:

```
tooltips.set_tip(widget, tip_text, tip_private=None)
```

The object *tooltips* is the tooltip you've already created. The first argument (*widget*) is the widget you wish to have this tooltip pop up for; the second (*tip_text*), the text you wish it to display. The last argument (*tip_private*) is a text string that can be used as an identifier.

The tooltip.py example program modifies the arrow.py program to add a tooltip for each button. Figure 9.3 illustrates the resulting display with the tooltip for the second arrow button displayed:

Figure 9.3 Tooltips Example



The source code for tooltip.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example tooltip.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # Create an Arrow widget with the specified parameters
10 # and pack it into a button
11 def create_arrow_button(arrow_type, shadow_type):
12     button = gtk.Button()
13     arrow = gtk.Arrow(arrow_type, shadow_type)
14     button.add(arrow)
15     button.show()
16     arrow.show()
17     return button
18
19 class Tooltips:
20     def __init__(self):
21         # Create a new window
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         window.set_title("Tooltips")
25
26         # It's a good idea to do this for all windows.
27         window.connect("destroy", lambda w: gtk.main_quit())
28
29         # Sets the border width of the window.
30         window.set_border_width(10)
31
32         # Create a box to hold the arrows/buttons
33         box = gtk.HBox(False, 0)
34         box.set_border_width(2)
```

```
35          window.add(box)
36
37          # create a tooltips object
38          self.tooltips = gtk.Tooltips()
39
40          # Pack and show all our widgets
41          box.show()
42
43          button = create_arrow_button(gtk.ARROW_UP, gtk.SHADOW_IN)
44          box.pack_start(button, False, False, 3)
45          self.tooltips.set_tip(button, "SHADOW_IN")
46
47          button = create_arrow_button(gtk.ARROW_DOWN, gtk.SHADOW_OUT)
48          box.pack_start(button, False, False, 3)
49          self.tooltips.set_tip(button, "SHADOW_OUT")
50
51          button = create_arrow_button(gtk.ARROW_LEFT, gtk.SHADOW_ETCHED_IN)
52          box.pack_start(button, False, False, 3)
53          self.tooltips.set_tip(button, "SHADOW_ETCHED_IN")
54
55          button = create_arrow_button(gtk.ARROW_RIGHT, gtk.SHADOW_ETCHED_OUT ↩
    )
56          box.pack_start(button, False, False, 3)
57          self.tooltips.set_tip(button, "SHADOW_ETCHED_OUT")
58
59          window.show()
60
61  def main():
62      gtk.main()
63      return 0
64
65  if __name__ == "__main__":
66      tt = Tooltips()
67      main()
```

There are other methods that can be used with tooltips. I will just list them with a brief description of what they do.

```
tooltips.enable()
```

Enable a disabled set of tooltips.

```
tooltips.disable()
```

Disable an enabled set of tooltips.

```
tooltips.set_delay(delay)
```

Sets how many milliseconds you have to hold your pointer over the widget before the tooltip will pop up. The default is 500 milliseconds (half a second).

And that's all the methods associated with tooltips. More than you'll ever want to know :-)

9.4 Progress Bars

Progress bars are used to show the status of an operation. They are pretty easy to use, as you will see with the code below. But first lets start out with the call to create a new progress bar.

```
progressbar = gtk.ProgressBar(adjustment=None)
```

The `adjustment` argument specifies an adjustment to use with the `progressbar`. If not specified an adjustment will be created. Now that the progress bar has been created we can use it.

```
progressbar.set_fraction(fraction)
```

The *progressbar* object is the progress bar you wish to operate on, and the argument (*fraction*) is the amount "completed", meaning the amount the progress bar has been filled from 0-100%. This is passed to the method as a real number ranging from 0 to 1.

A progress bar may be set to one of a number of orientations using the method:

```
progressbar.set_orientation(orientation)
```

The *orientation* argument may take one of the following values to indicate the direction in which the progress bar moves:

```
PROGRESS_LEFT_TO_RIGHT
PROGRESS_RIGHT_TO_LEFT
PROGRESS_BOTTOM_TO_TOP
PROGRESS_TOP_TO_BOTTOM
```

As well as indicating the amount of progress that has occurred, the progress bar may be set to just indicate that there is some activity. This can be useful in situations where progress cannot be measured against a value range. The following function indicates that some progress has been made.

```
progressbar.pulse()
```

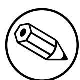
The step size of the activity indicator is set using the following function where fraction is between 0.0 and 1.0.

```
progressbar.set_pulse_step(fraction)
```

When not in activity mode, the progress bar can also display a configurable text string within its trough, using the following method:

```
progressbar.set_text(text)
```

NOTE



Note that set_text() doesn't support the `printf()`-like formatting of the GTK+ 1.2 Progressbar.

You can turn off the display of the string by calling set_text() again with no argument.

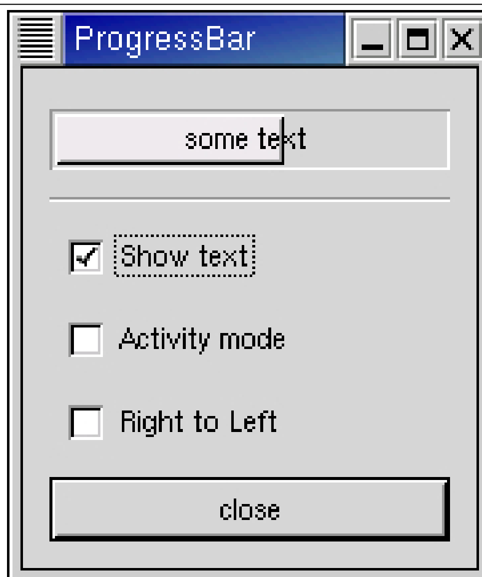
The current text setting of a progressbar can be retrieved with the following method:

```
text = progressbar.get_text()
```

Progress Bars are usually used with timeouts or other such functions (see Chapter 19) to give the illusion of multitasking. All will employ the set_fraction() or pulse() methods in the same manner.

The progressbar.py program provides an example of the progress bar, updated using timeouts. This code also shows you how to reset the Progress Bar. Figure 9.4 illustrates the resulting display:

Figure 9.4 ProgressBar Example



The source code for progressbar.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example progressbar.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk, gobject
 8
 9 # Update the value of the progress bar so that we get
10 # some movement
11 def progress_timeout(pbobj):
12     if pbobj.activity_check.get_active():
13         pbobj.pbar.pulse()
14     else:
15         # Calculate the value of the progress bar using the
16         # value range set in the adjustment object
17         new_val = pbobj.pbar.get_fraction() + 0.01
18         if new_val > 1.0:
19             new_val = 0.0
20         # Set the new value
21         pbobj.pbar.set_fraction(new_val)
22
23     # As this is a timeout function, return TRUE so that it
24     # continues to get called
25     return True
26
27 class ProgressBar:
28     # Callback that toggles the text display within the progress
29     # bar trough
30     def toggle_show_text(self, widget, data=None):
31         if widget.get_active():
32             self.pbar.set_text("some text")
33         else:
34             self.pbar.set_text("")
35
36     # Callback that toggles the activity mode of the progress
37     # bar
38     def toggle_activity_mode(self, widget, data=None):
39         if widget.get_active():
40             self.pbar.pulse()
```

```
 41         else:
 42             self.pbar.set_fraction(0.0)
 43 
 44     # Callback that toggles the orientation of the progress bar
 45     def toggle_orientation(self, widget, data=None):
 46         if self.pbar.get_orientation() == gtk.PROGRESS_LEFT_TO_RIGHT:
 47             self.pbar.set_orientation(gtk.PROGRESS_RIGHT_TO_LEFT)
 48         elif self.pbar.get_orientation() == gtk.PROGRESS_RIGHT_TO_LEFT:
 49             self.pbar.set_orientation(gtk.PROGRESS_LEFT_TO_RIGHT)
 50 
 51     # Clean up allocated memory and remove the timer
 52     def destroy_progress(self, widget, data=None):
 53         gobject.source_remove(self.timer)
 54         self.timer = 0
 55         gtk.main_quit()
 56 
 57     def __init__(self):
 58         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
 59         self.window.set_resizable(True)
 60 
 61         self.window.connect("destroy", self.destroy_progress)
 62         self.window.set_title("ProgressBar")
 63         self.window.set_border_width(0)
 64 
 65         vbox = gtk.VBox(False, 5)
 66         vbox.set_border_width(10)
 67         self.window.add(vbox)
 68         vbox.show()
 69 
 70         # Create a centering alignment object
 71         align = gtk.Alignment(0.5, 0.5, 0, 0)
 72         vbox.pack_start(align, False, False, 5)
 73         align.show()
 74 
 75         # Create the ProgressBar
 76         self.pbar = gtk.ProgressBar()
 77 
 78         align.add(self.pbar)
 79         self.pbar.show()
 80 
 81         # Add a timer callback to update the value of the progress bar
 82         self.timer = gobject.timeout_add (100, progress_timeout, self)
 83 
 84         separator = gtk.HSeparator()
 85         vbox.pack_start(separator, False, False, 0)
 86         separator.show()
 87 
 88         # rows, columns, homogeneous
 89         table = gtk.Table(2, 2, False)
 90         vbox.pack_start(table, False, True, 0)
 91         table.show()
 92 
 93         # Add a check button to select displaying of the trough text
 94         check = gtk.CheckButton("Show text")
 95         table.attach(check, 0, 1, 0, 1,
 96                      gtk.EXPAND | gtk.FILL, gtk.EXPAND | gtk.FILL,
 97                      5, 5)
 98         check.connect("clicked", self.toggle_show_text)
 99         check.show()
100 
101         # Add a check button to toggle activity mode
102         self.activity_check = check = gtk.CheckButton("Activity mode")
103         table.attach(check, 0, 1, 1, 2,
104                      gtk.EXPAND | gtk.FILL, gtk.EXPAND | gtk.FILL,
```

```
105                     5, 5)
106         check.connect("clicked", self.toggle_activity_mode)
107         check.show()
108
109         # Add a check button to toggle orientation
110         check = gtk.CheckButton("Right to Left")
111         table.attach(check, 0, 1, 2, 3,
112                      gtk.EXPAND | gtk.FILL, gtk.EXPAND | gtk.FILL,
113                      5, 5)
114         check.connect("clicked", self.toggle_orientation)
115         check.show()
116
117         # Add a button to exit the program
118         button = gtk.Button("close")
119         button.connect("clicked", self.destroy_progress)
120         vbox.pack_start(button, False, False, 0)
121
122         # This makes it so the button is the default.
123         button.set_flags(gtk.CAN_DEFAULT)
124
125         # This grabs this button to be the default button. Simply hitting
126         # the "Enter" key will cause this button to activate.
127         button.grab_default ()
128         button.show()
129
130         self.window.show()
131
132 def main():
133     gtk.main()
134     return 0
135
136 if __name__ == "__main__":
137     ProgressBar()
138     main()
```

9.5 Dialogs

The `Dialog` widget is very simple, and is actually just a window with a few things pre-packed into it for you. It simply creates a window, and then packs a `VBox` into the top, which contains a separator and then an `HBox` called the "action_area".

The `Dialog` widget can be used for pop-up messages to the user, and other similar tasks. It is really basic, and there is only one function for the dialog box, which is:

```
dialog = gtk.Dialog(title=None, parent=None, flags=0, buttons=None)
```

where *title* is the text to be used in the titlebar, *parent* is the main application window and *flags* set various modes of operation for the dialog:

```
DIALOG_MODAL - make the dialog modal
DIALOG_DESTROY_WITH_PARENT - destroy dialog when its parent is destroyed
DIALOG_NO_SEPARATOR - omit the separator between the vbox and the action_area
```

The *buttons* argument is a tuple of button text and response pairs. All arguments have defaults and can be specified using keywords.

This will create the dialog box, and it is now up to you to use it. You could pack a button in the action_area:

```
button = ...
dialog.action_area.pack_start(button, TRUE, TRUE, 0)
button.show()
```

And you could add to the *vbox* area by packing, for instance, a label in it, try something like this:

```
label = gtk.Label("Dialogs are groovy")
dialog.vbox.pack_start(label, TRUE, TRUE, 0)
label.show()
```

As an example in using the dialog box, you could put two buttons in the *action_area*, a Cancel button and an Ok button, and a label in the *vbox* area, asking the user a question or giving an error, etc. Then you could attach a different signal to each of the buttons and perform the operation the user selects.

If the simple functionality provided by the default vertical and horizontal boxes in the two areas doesn't give you enough control for your application, then you can simply pack another layout widget into the boxes provided. For example, you could pack a table into the vertical box.

9.5.1 Message Dialogs

A message dialog is a specialization of the already rather simple *Dialog* widget for displaying standardized error, question, and information popups. Invoke it like this:

```
message = gtk.MessageDialog(parent=None,
                            flags=0,
                            type=gtk.MESSAGE_INFO,
                            buttons=gtk.BUTTONS_NONE,
                            message_format=None)
```

The *type* flag selects a stock icon to be displayed in the message:

```
MESSAGE_INFO - information message
MESSAGE_WARNING - warning (or recoverable error) message
MESSAGE_QUESTION - question that can be answered with a button click
MESSAGE_ERROR - error message
```

To set the text for the message, feed it a Pango markup string. As a matter of style, you probably want to stick to relatively terse, one-sentence messages when using this widget.

```
message.set_markup("Sample message, could contain pango markup")
```

Here's an example program, message.py

```
1  #!/usr/bin/env python

2  # message.py -- example program illustrating use of message dialog widget

3  import pygtk
4  pygtk.require('2.0')
5  import gtk

6  if __name__ == "__main__":
7      message = gtk.MessageDialog(type=gtk.MESSAGE_ERROR, buttons=gtk. ↩
    BUTTONS_OK)
8      message.set_markup("An example error popup.")
9      message.run()
```

We used run() here to make the dialog modal; we could have achieved the same result by setting *flags* to `DIALOG_MODAL`, and doing this instead:

```
message.show()
gtk.main()
```

9.6 Images

`Images` are data structures that contain pictures. These pictures can be used in various places.

`Images` can be created from `Pixbufs`, `Pixmaps`, image files (e.g. XPM, PNG, JPEG, TIFF, etc.) and even animation files.

`Images` are created using the function:

```
image = gtk.Image()
```

The image is then loaded using one of the following methods:

```
image.set_from_pixbuf(pixbuf)
image.set_from_pixmap(pixmap, mask)
image.set_from_image(image)
image.set_from_file(filename)
image.set_from_stock(stock_id, size)
image.set_from_icon_set(icon_set, size)
image.set_from_animation(animation)
```

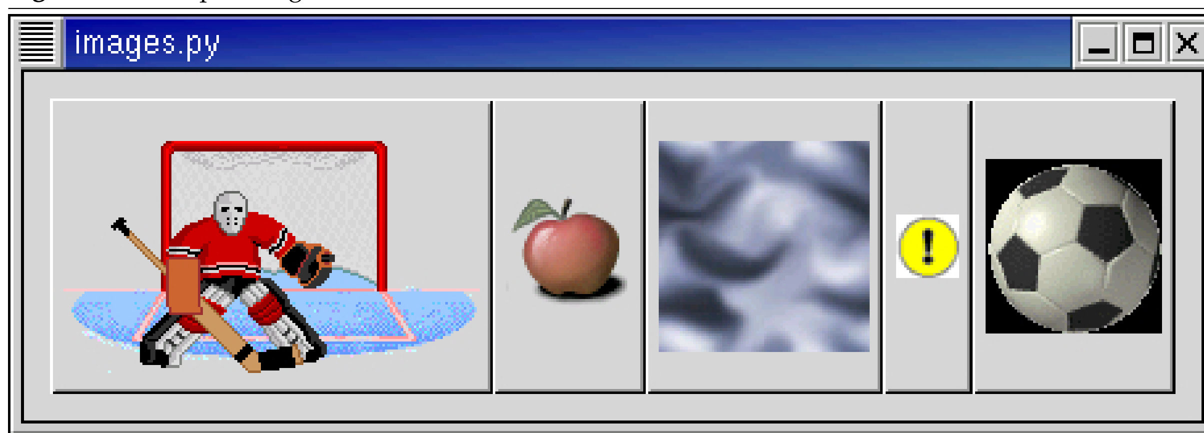
Where *pixbuf* is a `gtk.gdk.Pixbuf`; *pixmap* and *mask* are `gtk.gdk.Pixmaps`; *image* is a `gtk.gdk.Image`; *stock_id* is the name of a `gtk.StockItem`; *icon_set* is a `gtk.IconSet`; and, *animation* is a `gtk.gdk.PixbufAnimation`. the *size* argument is one of:

```
ICON_SIZE_MENU
ICON_SIZE_SMALL_TOOLBAR
ICON_SIZE_LARGE_TOOLBAR
ICON_SIZE_BUTTON
ICON_SIZE_DND
ICON_SIZE_DIALOG
```

The easiest way to create an image is using the set_from_file() method which automatically determines the image type and loads it.

The program images.py illustrates loading various image types (goalie.gif, apple-red.png, chaos.jpg, important.tif, soccerball.gif) into images which are then put into buttons:

Figure 9.5 Example Images in Buttons



The source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example images.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ImagesExample:
10     # when invoked (via signal delete_event), terminates the application.
11     def close_application(self, widget, event, data=None):
12         gtk.main_quit()
13         return False
14
15     # is invoked when the button is clicked.  It just prints a message.
16     def button_clicked(self, widget, data=None):
```

```
17          print "button %s clicked" % data
18
19      def __init__(self):
20          # create the main window, and attach delete_event signal to ↩
    terminating
21          # the application
22          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23          window.connect("delete_event", self.close_application)
24          window.set_border_width(10)
25          window.show()
26
27          # a horizontal box to hold the buttons
28          hbox = gtk.HBox()
29          hbox.show()
30          window.add(hbox)
31
32          pixbufanim = gtk.gdk.PixbufAnimation("goalie.gif")
33          image = gtk.Image()
34          image.set_from_animation(pixbufanim)
35          image.show()
36          # a button to contain the image widget
37          button = gtk.Button()
38          button.add(image)
39          button.show()
40          hbox.pack_start(button)
41          button.connect("clicked", self.button_clicked, "1")
42
43          # create several images with data from files and load images into
44          # buttons
45          image = gtk.Image()
46          image.set_from_file("apple-red.png")
47          image.show()
48          # a button to contain the image widget
49          button = gtk.Button()
50          button.add(image)
51          button.show()
52          hbox.pack_start(button)
53          button.connect("clicked", self.button_clicked, "2")
54
55          image = gtk.Image()
56          image.set_from_file("chaos.jpg")
57          image.show()
58          # a button to contain the image widget
59          button = gtk.Button()
60          button.add(image)
61          button.show()
62          hbox.pack_start(button)
63          button.connect("clicked", self.button_clicked, "3")
64
65          image = gtk.Image()
66          image.set_from_file("important.tif")
67          image.show()
68          # a button to contain the image widget
69          button = gtk.Button()
70          button.add(image)
71          button.show()
72          hbox.pack_start(button)
73          button.connect("clicked", self.button_clicked, "4")
74
75          image = gtk.Image()
76          image.set_from_file("soccerball.gif")
77          image.show()
78          # a button to contain the image widget
79          button = gtk.Button()
```

```
80          button.add(image)
81          button.show()
82          hbox.pack_start(button)
83          button.connect("clicked", self.button_clicked, "5")
84
85
86 def main():
87     gtk.main()
88     return 0
89
90 if __name__ == "__main__":
91     ImagesExample()
92     main()
```

9.6.1 Pixmaps

`Pixmaps` are data structures that contain pictures. These pictures can be used in various places, but most commonly as icons on the X desktop, or as cursors.

A pixmap which only has 2 colors is called a bitmap, and there are a few additional routines for handling this common special case.

To understand pixmaps, it would help to understand how X window system works. Under X, applications do not need to be running on the same computer that is interacting with the user. Instead, the various applications, called "clients", all communicate with a program which displays the graphics and handles the keyboard and mouse. This program which interacts directly with the user is called a "display server" or "X server." Since the communication might take place over a network, it's important to keep some information with the X server. `Pixmaps`, for example, are stored in the memory of the X server. This means that once pixmap values are set, they don't need to keep getting transmitted over the network; instead a command is sent to "display pixmap number XYZ here." Even if you aren't using X with GTK+ currently, using constructs such as `Pixmaps` will make your programs work acceptably under X.

To use pixmaps in PyGTK, we must first build a `gtk.gdk.Pixmap` using gtk.gdk functions in PyGTK. `Pixmaps` can either be created from in-memory data, or from data read from a file. We'll go through each of the calls to create a pixmap.

```
pixmap = gtk.gdk.pixmap_create_from_data(window, data, width, height, depth, fg ↩
    , bg)
```

This routine is used to create a *`pixmap`* from *`data`* in memory with the color depth given by *`depth`*. If *`depth`* is -1 the color depth is derived from the depth of *`window`*. Each pixel uses *`depth`* bits of data to represent the color. *`Width`* and *`height`* are in pixels. The *`window`* argument must refer to a realized `gtk.gdk.Window`, since a pixmap's resources are meaningful only in the context of the screen where it is to be displayed. *`fg`* and *`bg`* are the foreground and background colors of the pixmap.

Pixmaps can be created from XPM files using:

```
pixmap, mask = gtk.gdk.pixmap_create_from_xpm(window, transparent_color,  ↩
    filename)
```

XPM format is a readable pixmap representation for the X Window System. It is widely used and many different utilities are available for creating image files in this format. In the `pixmap_create-_from_xpm`() function the first argument is a `gtk.gdk.Window` type. (Most GTK+ widgets have an underlying `gtk.gdk.Window` which can be retrieved by using the widget's window attribute.) The file, specified by *`filename`*, must contain an image in the XPM format and the image is loaded into the *`pixmap`* structure. The *`mask`* is a bitmap that specifies which bits of *`pixmap`* are opaque; it is created by the function. All other pixels are colored using the color specified by *`transparent_color`*. An example using this function is below.

Pixmaps can also be created from data in memory using the function:

```
pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(window, transparent_color, data ↩
    )
```

Small images can be incorporated into a program as data in the XPM format using the above function. A pixmap is created using this data, instead of reading it from a file. An example of such data is:

```
xpm_data = [
"16 16 3 1",
"       c None",
".      c #000000000000",
"X      c #FFFFFFFFFFFF",
"                ",
"   ......       ",
"   .XXX.X.      ",
"   .XXX.XX.     ",
"   .XXX.XXX.    ",
"   .XXX.....    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .........    ",
"                ",
"                "
]
```

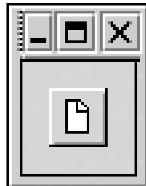
The final way to create a blank pixmap suitable for drawing operations is:

```
pixmap = gtk.gdk.Pixmap(window, width, height, depth=-1)
```

window is either a `gtk.gdk.Window`. or `None`. If *window* is a `gtk.gdk.Window` then *depth* can be -1 to indicate that the depth should be determined from the window. If *window* is `None` then the *depth* must be specified.

The pixmap.py program is an example of using a pixmap in a button. Figure 9.6 shows the result:

Figure 9.6 Pixmap in a Button Example



The source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example pixmap.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # XPM data of Open-File icon
10 xpm_data = [
11 "16 16 3 1",
12 "       c None",
13 ".      c #000000000000",
14 "X      c #FFFFFFFFFFFF",
15 "                ",
16 "   ......       ",
17 "   .XXX.X.      ",
18 "   .XXX.XX.     ",
19 "   .XXX.XXX.    ",
20 "   .XXX.....    ",
```

```
21 "    .XXXXXXX.    ",
22 "    .XXXXXXX.    ",
23 "    .XXXXXXX.    ",
24 "    .XXXXXXX.    ",
25 "    .XXXXXXX.    ",
26 "    .XXXXXXX.    ",
27 "    .XXXXXXX.    ",
28 "    .........    ",
29 "                 ",
30 "                 "
31 ]
32
33 class PixmapExample:
34     # when invoked (via signal delete_event), terminates the application.
35     def close_application(self, widget, event, data=None):
36         gtk.main_quit()
37         return False
38
39     # is invoked when the button is clicked.  It just prints a message.
40     def button_clicked(self, widget, data=None):
41         print "button clicked"
42
43     def __init__(self):
44         # create the main window, and attach delete_event signal to ↩
   terminating
45         # the application
46         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
47         window.connect("delete_event", self.close_application)
48         window.set_border_width(10)
49         window.show()
50
51         # now for the pixmap from XPM data
52         pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(window.window,
53                                                         None,
54                                                         xpm_data)
55
56         # an image widget to contain the pixmap
57         image = gtk.Image()
58         image.set_from_pixmap(pixmap, mask)
59         image.show()
60
61         # a button to contain the image widget
62         button = gtk.Button()
63         button.add(image)
64         window.add(button)
65         button.show()
66
67         button.connect("clicked", self.button_clicked)
68
69 def main():
70     gtk.main()
71     return 0
72
73 if __name__ == "__main__":
74     PixmapExample()
75     main()
```

A disadvantage of using pixmaps is that the displayed object is always rectangular, regardless of the image. We would like to create desktops and applications with icons that have more natural shapes. For example, for a game interface, we would like to have round buttons to push. The way to do this is using shaped windows.

A shaped window is simply a pixmap where the background pixels are transparent. This way, when the background image is multi-colored, we don't overwrite it with a rectangular, non-matching border around our icon. The wheelbarrow.py example program displays a full wheelbarrow image on the

desktop. Figure 9.7 shows the wheelbarrow over a terminal window:

Figure 9.7 Wheelbarrow Example Shaped Window

The source code for wheelbarrow.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example wheelbarrow.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # XPM
10 WheelbarrowFull_xpm = [
11 "48 48 64 1",
12 "        c None",
13 ".       c #DF7DCF3CC71B",
14 "X       c #965875D669A6",
15 "o       c #71C671C671C6",
16 "O       c #A699A289A699",
17 "+       c #965892489658",
18 "@       c #8E38410330C2",
19 "#       c #D75C7DF769A6",
20 "$       c #F7DECF3CC71B",
21 "%       c #96588A288E38",
22 "&       c #A69992489E79",
23 "*       c #8E3886178E38",
24 "=       c #104008200820",
25 "-       c #596510401040",
26 ";       c #C71B30C230C2",
27 ":       c #C71B9A699658",
28 ">       c #618561856185",
29 ",       c #20811C712081",
30 "<       c #104000000000",
31 "1       c #861720812081",
32 "2       c #DF7D4D344103",
33 "3       c #79E769A671C6",
34 "4       c #861782078617",
35 "5       c #41033CF34103",
36 "6       c #000000000000",
37 "7       c #49241C711040",
38 "8       c #492445144924",
39 "9       c #082008200820",
40 "0       c #69A618611861",
41 "q       c #B6DA71C65144",
42 "w       c #410330C238E3",
43 "e       c #CF3CBAEAB6DA",
44 "r       c #71C6451430C2",
```

```
 45 "t      c #EFBEDB6CD75C",
 46 "y      c #28A208200820",
 47 "u      c #186110401040",
 48 "i      c #596528A21861",
 49 "p      c #71C661855965",
 50 "a      c #A69996589658",
 51 "s      c #30C228A230C2",
 52 "d      c #BEFBA289AEBA",
 53 "f      c #596545145144",
 54 "g      c #30C230C230C2",
 55 "h      c #8E3882078617",
 56 "j      c #208118612081",
 57 "k      c #38E30C300820",
 58 "l      c #30C2208128A2",
 59 "z      c #38E328A238E3",
 60 "x      c #514438E34924",
 61 "c      c #618555555965",
 62 "v      c #30C2208130C2",
 63 "b      c #38E328A230C2",
 64 "n      c #28A228A228A2",
 65 "m      c #41032CB228A2",
 66 "M      c #104010401040",
 67 "N      c #492438E34103",
 68 "B      c #28A2208128A2",
 69 "V      c #A699596538E3",
 70 "C      c #30C21C711040",
 71 "Z      c #30C218611040",
 72 "A      c #965865955965",
 73 "S      c #618534D32081",
 74 "D      c #38E31C711040",
 75 "F      c #082000000820",
 76 "                                                ",
 77 "          .XoO                                  ",
 78 "         +@#$%o&                                ",
 79 "         *=-;#::o+                              ",
 80 "           >,<12#:34                            ",
 81 "             45671#:X3                          ",
 82 "               +89<02qwo                        ",
 83 "e*                >,67;ro                       ",
 84 "ty>                 459@>+&&                    ",
 85 "$2u+                  ><ipas8*                  ",
 86 "%$;=*                *3:.Xa.dfg>                ",
 87 "Oh$;ya             *3d.a8j,Xe.d3g8+             ",
 88 " Oh$;ka          *3d$a8lz,,xxc:.e3g54           ",
 89 "  Oh$;kO       *pd$%svbzz,sxxxxfX..&wn>         ",
 90 "   Oh$@mO    *3dthwlsslszjzxxxxxxx3:td8M4       ",
 91 "    Oh$@g& *3d$XNlvvvlllm,mNwxxxxxxxfa.:,B*     ",
 92 "     Oh$@,Od.czlllllzlmmqV@V#V@fxxxxxxxf:%j5&   ",
 93 "      Oh$1hd5lllslllCCZrV#r#:#2AxxxxxxxxxcdwM*  ",
 94 "       OXq6c.%8vvvllZZiqqApA:mq:Xxcpcxxxxxfdc9* ",
 95 "        2r<6gde3bllZZrVi7S@SV77A::qApxxxxxxfdcM ",
 96 "        :,q-6MN.dfmZZrrSS:#riirDSAX@Af5xxxxxfevo",
 97 "         +A26jguXtAZZZC7iDiCCrVVii7Cmmmxxxxxx%3g",
 98 "          *#16jszN..3DZZZZrCVSA2rZrV7Dmmwxxxx&en",
 99 "           p2yFvzssXe:fCZZCiiD7iiZDiDSSZwwxx8e*>",
100 "           OA1<jzxwwc:$d%NDZZZZCCCZCCZZCmxxfd.B ",
101 "            3206Bwxxszx%et.eaAp77m77mmmf3&eeeg* ",
102 "             @26MvzxNzvlbwfpdettttttttttt.c,n&  ",
103 "             *;16=lsNwwNwgsvslbwwvccc3pcfu<o    ",
104 "              p;<69BvwwsszslllbBlllllllu<5+     ",
105 "              OS0y6FBlvvvzvzss,u=Blllj=54       ",
106 "               c1-699Blvlllllu7k96MMMg4         ",
107 "               *10y8n6FjvllllB<166668           ",
108 "                S-kg+>666<M<996-y6n<8*          ",
```

```
109 "                p71=4 m69996kD8Z-66698&&        ",
110 "                &i0ycm6n4 ogk17,0<6666g         ",
111 "                 N-k-<>     >=01-kuu666>        ",
112 "                 ,6ky&      &46-10ul,66,        ",
113 "                 Ou0<>       o66y<ulw<66&       ",
114 "                  *kk5       >66By7=xu664       ",
115 "                   <<M4      466lj<Mxu66o       ",
116 "                   *>>       +66uv,zN666*       ",
117 "                              566,xxj669        ",
118 "                              4666FF666>        ",
119 "                               >966666M         ",
120 "                                oM6668+         ",
121 "                                  *4            ",
122 "                                                ",
123 "                                                "
124 ]
125
126 class WheelbarrowExample:
127     # When invoked (via signal delete_event), terminates the application
128     def close_application(self, widget, event, data=None):
129         gtk.main_quit()
130         return False
131
132     def __init__(self):
133         # Create the main window, and attach delete_event signal to ←
    terminate
134         # the application.  Note that the main window will not have a ←
    titlebar
135         # since we're making it a popup.
136         window = gtk.Window(gtk.WINDOW_POPUP)
137         window.connect("delete_event", self.close_application)
138         window.set_events(window.get_events() | gtk.gdk.BUTTON_PRESS_MASK)
139         window.connect("button_press_event", self.close_application)
140         window.show()
141
142         # Now for the pixmap and the image widget
143         pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(
144             window.window, None, WheelbarrowFull_xpm)
145         image = gtk.Image()
146         image.set_from_pixmap(pixmap, mask)
147         image.show()
148
149         # To display the image, we use a fixed widget to place the image
150         fixed = gtk.Fixed()
151         fixed.set_size_request(200, 200)
152         fixed.put(image, 0, 0)
153         window.add(fixed)
154         fixed.show()
155
156         # This masks out everything except for the image itself
157         window.shape_combine_mask(mask, 0, 0)
158
159         # show the window
160         window.set_position(gtk.WIN_POS_CENTER_ALWAYS)
161         window.show()
162
163 def main():
164     gtk.main()
165     return 0
166
167 if __name__ == "__main__":
168     WheelbarrowExample()
169     main()
```

To make the wheelbarrow image sensitive, we attached the "button_press_event" signal to make the program exit. Lines 138-139 make the picture sensitive to a mouse button being pressed and connect the close_application() method.

9.7 Rulers

`Ruler` widgets are used to indicate the location of the mouse pointer in a given window. A window can have a horizontal ruler spanning across the width and a vertical ruler spanning down the height. A small triangular indicator on the ruler shows the exact location of the pointer relative to the ruler.

A ruler must first be created. Horizontal and vertical rulers are created using the functions:

```
hruler = gtk.HRuler()     # horizontal ruler

vruler = gtk.VRuler()     # vertical ruler
```

Once a ruler is created, we can define the unit of measurement. Units of measure for rulers can be `PIXELS`, `INCHES` or `CENTIMETERS`. This is set using the method:

```
ruler.set_metric(metric)
```

The default measure is `PIXELS`.

```
ruler.set_metric(gtk.PIXELS)
```

Other important characteristics of a ruler are how to mark the units of scale and where the position indicator is initially placed. These are set for a ruler using the method:

```
ruler.set_range(lower, upper, position, max_size)
```

The *lower* and *upper* arguments define the extent of the ruler, and *max_size* is the largest possible number that will be displayed. *Position* defines the initial position of the pointer indicator within the ruler.

A vertical ruler can span an 800 pixel wide window thus:

```
vruler.set_range(0, 800, 0, 800)
```

The markings displayed on the ruler will be from 0 to 800, with a number for every 100 pixels. If instead we wanted the ruler to range from 7 to 16, we would code:

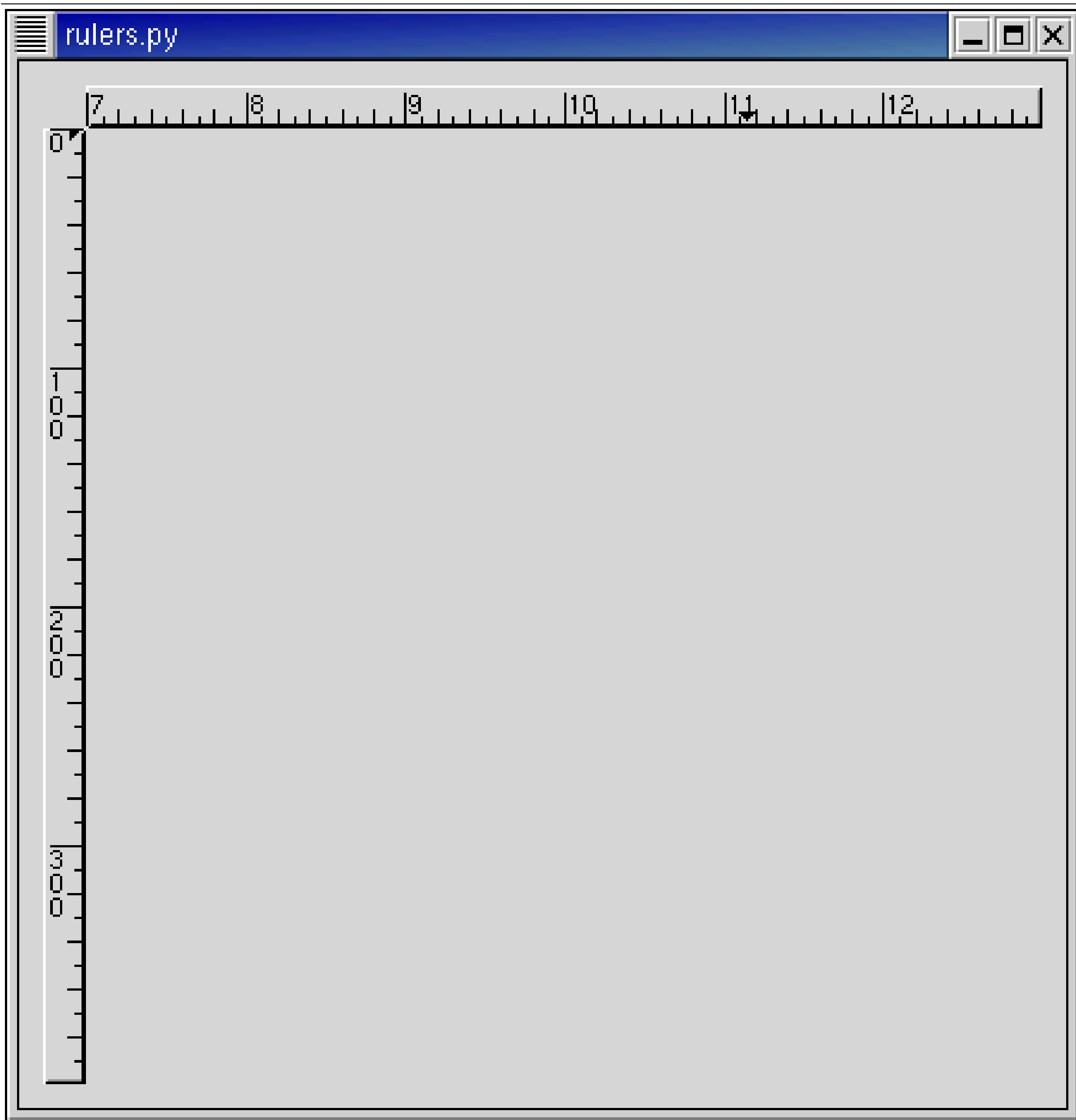
```
vruler.set_range(7, 16, 0, 20)
```

The indicator on the ruler is a small triangular mark that indicates the position of the pointer relative to the ruler. If the ruler is used to follow the mouse pointer, the "motion_notify_event" signal should be connected to the "motion_notify_event" method of the ruler. We need to setup a "motion_notify_event" callback for the area and use connect_object() to get the ruler to emit a "motion_notify_signal":

```
def motion_notify(ruler, event):
    return ruler.emit("motion_notify_event", event)

area.connect_object("motion_notify_event", motion_notify, ruler)
```

The rulers.py example program creates a drawing area with a horizontal ruler above it and a vertical ruler to the left of it. The size of the drawing area is 600 pixels wide by 400 pixels high. The horizontal ruler spans from 7 to 13 with a mark every 100 pixels, while the vertical ruler spans from 0 to 400 with a mark every 100 pixels. Placement of the drawing area and the rulers is done using a table. Figure 9.8 illustrates the result:

Figure 9.8 Rulers Example



The rulers.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example rulers.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class RulersExample:
10     XSIZE = 400
11     YSIZE = 400
12
13     # This routine gets control when the close button is clicked
14     def close_application(self, widget, event, data=None):
15         gtk.main_quit()
16         return False
17
```

```
18     def __init__(self):
19         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
20         window.connect("delete_event", self.close_application)
21         window.set_border_width(10)
22
23         # Create a table for placing the ruler and the drawing area
24         table = gtk.Table(3, 2, False)
25         window.add(table)
26
27         area = gtk.DrawingArea()
28         area.set_size_request(self.XSIZE, self.YSIZE)
29         table.attach(area, 1, 2, 1, 2,
30                      gtk.EXPAND|gtk.FILL, gtk.FILL, 0, 0 )
31         area.set_events(gtk.gdk.POINTER_MOTION_MASK |
32                         gtk.gdk.POINTER_MOTION_HINT_MASK )
33
34         # The horizontal ruler goes on top. As the mouse moves across the
35         # drawing area, a motion_notify_event is passed to the
36         # appropriate event handler for the ruler.
37         hrule = gtk.HRuler()
38         hrule.set_metric(gtk.PIXELS)
39         hrule.set_range(7, 13, 0, 20)
40         def motion_notify(ruler, event):
41             return ruler.emit("motion_notify_event", event)
42         area.connect_object("motion_notify_event", motion_notify, hrule)
43         table.attach(hrule, 1, 2, 0, 1,
44                      gtk.EXPAND|gtk.SHRINK|gtk.FILL, gtk.FILL, 0, 0 )
45
46         # The vertical ruler goes on the left. As the mouse moves across
47         # the drawing area, a motion_notify_event is passed to the
48         # appropriate event handler for the ruler.
49         vrule = gtk.VRuler()
50         vrule.set_metric(gtk.PIXELS)
51         vrule.set_range(0, self.YSIZE, 10, self.YSIZE)
52         area.connect_object("motion_notify_event", motion_notify, vrule)
53         table.attach(vrule, 0, 1, 1, 2,
54                      gtk.FILL, gtk.EXPAND|gtk.SHRINK|gtk.FILL, 0, 0 )
55
56         # Now show everything
57         area.show()
58         hrule.show()
59         vrule.show()
60         table.show()
61         window.show()
62
63 def main():
64     gtk.main()
65     return 0
66
67 if __name__ == "__main__":
68     RulersExample()
69     main()
```

Lines 42 and 52 connect the motion_notify() callback to the area but passing `hrule` in line 42 and `vrule` in line 52 as user data. The motion_notify() callback will be called twice each time the mouse moves - once with `hrule` and once with `vrule`.

9.8 Statusbars

`Statusbars` are simple widgets used to display a text message. They keep a stack of the messages pushed onto them, so that popping the current message will re-display the previous text message.

In order to allow different parts of an application to use the same statusbar to display messages, the statusbar widget issues Context Identifiers which are used to identify different "users". The message on

top of the stack is the one displayed, no matter what context it is in. Messages are stacked in last-in-first-out order, not context identifier order.

A statusbar is created with a call to:

```
statusbar = gtk.Statusbar()
```

A new Context Identifier is requested using a call to the following method with a short textual description of the context:

```
context_id = statusbar.get_context_id(context_description)
```

There are three additional methods that operate on statusbars:

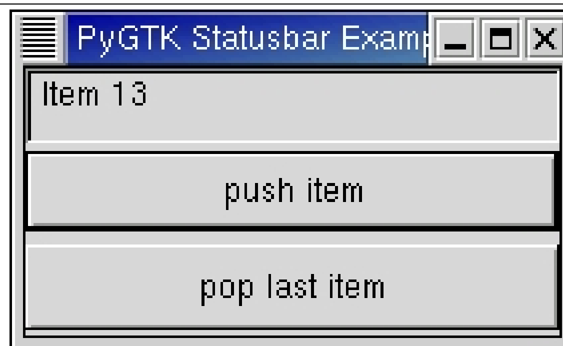
```
message_id = statusbar.push(context_id, text)

statusbar.pop(context_id)

statusbar.remove(context_id, message_id)
```

The first, push(), is used to add a new message to the `statusbar`. It returns a `message_id`, which can be passed later to the remove() method to remove the message with the combination `message_id` and `context_id` from the `statusbar`'s stack.

The pop() method removes the message highest in the stack with the given `context_id`.

The statusbar.py example program creates a statusbar and two buttons, one for pushing items onto the statusbar, and one for popping the last item back off. Figure 9.9 illustrates the result:

Figure 9.9 Statusbar Example



The statusbar.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example statusbar.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class StatusbarExample:
10     def push_item(self, widget, data):
11         buff = " Item %d" % self.count
12         self.count = self.count + 1
13         self.status_bar.push(data, buff)
14         return
15
16     def pop_item(self, widget, data):
17         self.status_bar.pop(data)
18         return
19
20     def __init__(self):
21         self.count = 1
22         # create a new window
```

```
23          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
24          window.set_size_request(200, 100)
25          window.set_title("PyGTK Statusbar Example")
26          window.connect("delete_event", lambda w,e: gtk.main_quit())
27
28          vbox = gtk.VBox(False, 1)
29          window.add(vbox)
30          vbox.show()
31
32          self.status_bar = gtk.Statusbar()
33          vbox.pack_start(self.status_bar, True, True, 0)
34          self.status_bar.show()
35
36          context_id = self.status_bar.get_context_id("Statusbar example")
37
38          button = gtk.Button("push item")
39          button.connect("clicked", self.push_item, context_id)
40          vbox.pack_start(button, True, True, 2)
41          button.show()
42
43          button = gtk.Button("pop last item")
44          button.connect("clicked", self.pop_item, context_id)
45          vbox.pack_start(button, True, True, 2)
46          button.show()
47
48          # always display the window as the last step so it all splashes on
49          # the screen at once.
50          window.show()
51
52  def main():
53      gtk.main()
54      return 0
55
56  if __name__ == "__main__":
57      StatusbarExample()
58      main()
```

9.9 Text Entries

The `Entry` widget allows text to be typed and displayed in a single line text box. The text may be set with method calls that allow new text to replace, prepend or append the current contents of the `Entry` widget.

The function for creating an `Entry` widget is:

```
entry = gtk.Entry(max=0)
```

If the *max* argument is given it sets a limit on the length of the text within the `Entry`. If *max* is 0 then there is no limit.

The maximum length of the entry can be changed using the method:

```
entry.set_max_length(max)
```

The next method alters the text which is currently within the `Entry` widget.

```
entry.set_text(text)
```

The set_text() method sets the contents of the `Entry` widget to *text*, replacing the current contents. Note that the class `Entry` implements the `Editable` interface (yes, `gobject` supports Java-like interfaces) which contains some more functions for manipulating the contents. For example, the method:

```
entry.insert_text(text, position=0)
```

inserts *text* at the given position within the *entry*.

The contents of the `Entry` can be retrieved by using a call to the following method. This is useful in the callback methods described below.

```
text = entry.get_text()
```

If we don't want the contents of the `Entry` to be changed by someone typing into it, we can change its editable state.

```
entry.set_editable(is_editable)
```

The above method allows us to toggle the editable state of the `Entry` widget by passing in a `TRUE` or `FALSE` value for the *`is_editable`* argument.

If we are using the `Entry` where we don't want the text entered to be visible, for example when a password is being entered, we can use the following method, which also takes a boolean flag.

```
entry.set_visibility(visible)
```

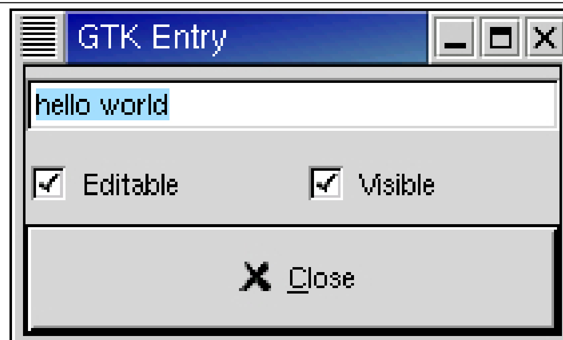
A region of the text may be set as selected by using the following method. This would most often be used after setting some default text in an `Entry`, making it easy for the user to remove it.

```
entry.select_region(start, end)
```

If we want to be notified when the user has entered text, we can connect to the "activate" or "changed" signal. Activate is raised when the user hits the enter key within the `Entry` widget. Changed is raised when the any change is made to the text, e.g. for every character entered or removed.

The entry.py example program illustrates the use of an `Entry` widget. Figure 9.10 shows the result of running the program:

Figure 9.10 Entry Example



The entry.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example entry.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class EntryExample:
10     def enter_callback(self, widget, entry):
11         entry_text = entry.get_text()
12         print "Entry contents: %s\n" % entry_text
13
14     def entry_toggle_editable(self, checkbutton, entry):
15         entry.set_editable(checkbutton.get_active())
16
17     def entry_toggle_visibility(self, checkbutton, entry):
18         entry.set_visibility(checkbutton.get_active())
19
20     def __init__(self):
```

```
21          # create a new window
22          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23          window.set_size_request(200, 100)
24          window.set_title("GTK Entry")
25          window.connect("delete_event", lambda w,e: gtk.main_quit())
26
27          vbox = gtk.VBox(False, 0)
28          window.add(vbox)
29          vbox.show()
30
31          entry = gtk.Entry()
32          entry.set_max_length(50)
33          entry.connect("activate", self.enter_callback, entry)
34          entry.set_text("hello")
35          entry.insert_text(" world", len(entry.get_text()))
36          entry.select_region(0, len(entry.get_text()))
37          vbox.pack_start(entry, True, True, 0)
38          entry.show()
39
40          hbox = gtk.HBox(False, 0)
41          vbox.add(hbox)
42          hbox.show()
43
44          check = gtk.CheckButton("Editable")
45          hbox.pack_start(check, True, True, 0)
46          check.connect("toggled", self.entry_toggle_editable, entry)
47          check.set_active(True)
48          check.show()
49
50          check = gtk.CheckButton("Visible")
51          hbox.pack_start(check, True, True, 0)
52          check.connect("toggled", self.entry_toggle_visibility, entry)
53          check.set_active(True)
54          check.show()
55
56          button = gtk.Button(stock=gtk.STOCK_CLOSE)
57          button.connect("clicked", lambda w: gtk.main_quit())
58          vbox.pack_start(button, True, True, 0)
59          button.set_flags(gtk.CAN_DEFAULT)
60          button.grab_default()
61          button.show()
62          window.show()
63
64  def main():
65      gtk.main()
66      return 0
67
68  if __name__ == "__main__":
69      EntryExample()
70      main()
```

9.10 Spin Buttons

The SpinButton widget is generally used to allow the user to select a value from a range of numeric values. It consists of a text entry box with up and down arrow buttons attached to the side. Selecting one of the buttons causes the value to "spin" up and down the range of possible values. The entry box may also be edited directly to enter a specific value.

The SpinButton allows the value to have zero or more decimal places and to be incremented/decremented in configurable steps. The action of holding down one of the buttons optionally results in an acceleration of change in the value according to how long it is depressed.

The SpinButton uses an Adjustment (see Chapter 7) object to hold information about the range

of values that the spin button can take. This makes for a powerful `SpinButton` widget.

Recall that an `Adjustment` widget is created with the following function, which illustrates the information that it holds:

```
adjustment = gtk.Adjustment(value=0, lower=0, upper=0, step_incr=0, page_incr ↩
    =0, page_size=0)
```

These attributes of an `Adjustment` are used by the `SpinButton` in the following way:

value	initial value for the Spin Button
lower	lower range value
upper	*upper* range value
step_increment	value to increment/decrement when pressing mouse button-1 on a button
page_increment	value to increment/decrement when pressing mouse button-2 on a button
page_size	unused

Additionally, mouse button-3 can be used to jump directly to the *upper* or *lower* values when used to select one of the buttons. Lets look at how to create a `SpinButton`:

```
spin_button = gtk.SpinButton(adjustment=None, climb_rate=0.0, digits=0)
```

The *climb_rate* argument take a value between 0.0 and 1.0 and indicates the amount of acceleration that the `SpinButton` has. The *digits* argument specifies the number of decimal places to which the value will be displayed.

A `SpinButton` can be reconfigured after creation using the following method:

```
spin_button.configure(adjustment, climb_rate, digits)
```

The *spin_button* argument specifies the `SpinButton` widget that is to be reconfigured. The other arguments are as specified above.

The *adjustment* can be set and retrieved independently using the following two methods:

```
spin_button.set_adjustment(adjustment)

adjustment = spin_button.get_adjustment()
```

The number of decimal places can also be altered using:

```
spin_button.set_digits(digits)
```

The value that a `SpinButton` is currently displaying can be changed using the following method:

```
spin_button.set_value(value)
```

The current value of a `SpinButton` can be retrieved as either a floating point or integer value with the following methods:

```
float_value = spin_button.get_value()

int_value = spin_button.get_value_as_int()
```

If you want to alter the value of a `SpinButton` relative to its current value, then the following method can be used:

```
spin_button.spin(direction, increment)
```

The *direction* parameter can take one of the following values:

```
SPIN_STEP_FORWARD
SPIN_STEP_BACKWARD
SPIN_PAGE_FORWARD
SPIN_PAGE_BACKWARD
SPIN_HOME
SPIN_END
SPIN_USER_DEFINED
```

This method packs in quite a bit of functionality, which I will attempt to clearly explain. Many of these settings use values from the `Adjustment` object that is associated with a `SpinButton`.

`SPIN_STEP_FORWARD` and `SPIN_STEP_BACKWARD` change the value of the `SpinButton` by the amount specified by *increment*, unless *increment* is equal to 0, in which case the value is changed by the value of *step_increment* in the `Adjustment`.

`SPIN_PAGE_FORWARD` and `SPIN_PAGE_BACKWARD` simply alter the value of the `SpinButton` by *increment*.

`SPIN_HOME` sets the value of the `SpinButton` to the bottom of the `Adjustment` range.

`SPIN_END` sets the value of the `SpinButton` to the top of the `Adjustment` range.

`SPIN_USER_DEFINED` simply alters the value of the `SpinButton` by the specified amount.

We move away from methods for setting and retrieving the range attributes of the `SpinButton` now, and move onto methods that effect the appearance and behavior of the `SpinButton` widget itself.

The first of these methods is used to constrain the text box of the `SpinButton` such that it may only contain a numeric value. This prevents a user from typing anything other than numeric values into the text box of a `SpinButton`:

```
spin_button.set_numeric(numeric)
```

numeric is `TRUE` to constrain the text entry to numeric values or `FALSE` to unconstrain the text entry.

You can set whether a `SpinButton` will wrap around between the upper and lower range values with the following method:

```
spin_button.set_wrap(wrap)
```

The `SpinButton` will wrap when *wrap* is set to `TRUE`.

You can set a `SpinButton` to round the value to the nearest *step_increment*, which is set within the `Adjustment` object used with the `SpinButton`. This is accomplished with the following method when *snap_to_ticks* is `TRUE`:

```
spin_button.set_snap_to_ticks(snap_to_ticks)
```

The update policy of a `SpinButton` can be changed with the following method:

```
spin_button.set_update_policy(policy)
```

The possible values of policy are:

```
UPDATE_ALWAYS

UPDATE_IF_VALID
```

These policies affect the behavior of a `SpinButton` when parsing inserted text and syncing its value with the values of the `Adjustment`.

In the case of `UPDATE_IF_VALID` the `SpinButton` value only gets changed if the text input is a numeric value that is within the range specified by the `Adjustment`. Otherwise the text is reset to the current value.

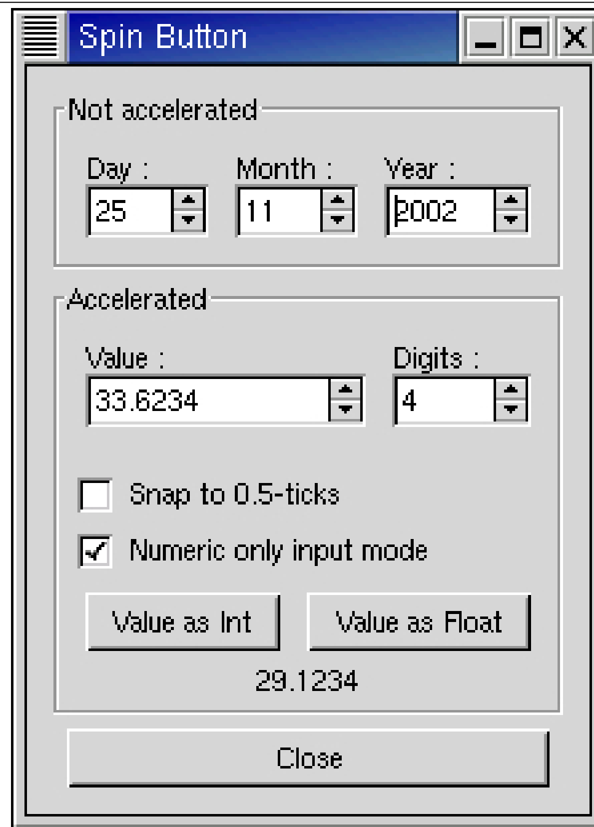
In case of `UPDATE_ALWAYS` we ignore errors while converting text into a numeric value.

Finally, you can explicitly request that a `SpinButton` update itself:

```
spin_button.update()
```

The spinbutton.py example program illustrates the use of spinbuttons including setting a number of characteristics. Figure 9.11 shows the result of running the example program:

Figure 9.11 Spin Button Example



The spinbutton.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example spinbutton.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class SpinButtonExample:
10     def toggle_snap(self, widget, spin):
11         spin.set_snap_to_ticks(widget.get_active())
12
13     def toggle_numeric(self, widget, spin):
14         spin.set_numeric(widget.get_active())
15
16     def change_digits(self, widget, spin, spin1):
17         spin1.set_digits(spin.get_value_as_int())
18
19     def get_value(self, widget, data, spin, spin2, label):
20         if data == 1:
21             buf = "%d" % spin.get_value_as_int()
22         else:
23             buf = "%0.*f" % (spin2.get_value_as_int(),
24                              spin.get_value())
25         label.set_text(buf)
26
27     def __init__(self):
28         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
29         window.connect("destroy", lambda w: gtk.main_quit())
30         window.set_title("Spin Button")
31
```

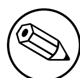
```
32          main_vbox = gtk.VBox(False, 5)
33          main_vbox.set_border_width(10)
34          window.add(main_vbox)
35
36          frame = gtk.Frame("Not accelerated")
37          main_vbox.pack_start(frame, True, True, 0)
38
39          vbox = gtk.VBox(False, 0)
40          vbox.set_border_width(5)
41          frame.add(vbox)
42
43          # Day, month, year spinners
44          hbox = gtk.HBox(False, 0)
45          vbox.pack_start(hbox, True, True, 5)
46
47          vbox2 = gtk.VBox(False, 0)
48          hbox.pack_start(vbox2, True, True, 5)
49
50          label = gtk.Label("Day :")
51          label.set_alignment(0, 0.5)
52          vbox2.pack_start(label, False, True, 0)
53
54          adj = gtk.Adjustment(1.0, 1.0, 31.0, 1.0, 5.0, 0.0)
55          spinner = gtk.SpinButton(adj, 0, 0)
56          spinner.set_wrap(True)
57          vbox2.pack_start(spinner, False, True, 0)
58
59          vbox2 = gtk.VBox(False, 0)
60          hbox.pack_start(vbox2, True, True, 5)
61
62          label = gtk.Label("Month :")
63          label.set_alignment(0, 0.5)
64          vbox2.pack_start(label, False, True, 0)
65
66          adj = gtk.Adjustment(1.0, 1.0, 12.0, 1.0, 5.0, 0.0)
67          spinner = gtk.SpinButton(adj, 0, 0)
68          spinner.set_wrap(True)
69          vbox2.pack_start(spinner, False, True, 0)
70
71          vbox2 = gtk.VBox(False, 0)
72          hbox.pack_start(vbox2, True, True, 5)
73
74          label = gtk.Label("Year :")
75          label.set_alignment(0, 0.5)
76          vbox2.pack_start(label, False, True, 0)
77
78          adj = gtk.Adjustment(1998.0, 0.0, 2100.0, 1.0, 100.0, 0.0)
79          spinner = gtk.SpinButton(adj, 0, 0)
80          spinner.set_wrap(False)
81          spinner.set_size_request(55, -1)
82          vbox2.pack_start(spinner, False, True, 0)
83
84          frame = gtk.Frame("Accelerated")
85          main_vbox.pack_start(frame, True, True, 0)
86
87          vbox = gtk.VBox(False, 0)
88          vbox.set_border_width(5)
89          frame.add(vbox)
90
91          hbox = gtk.HBox(False, 0)
92          vbox.pack_start(hbox, False, True, 5)
93
94          vbox2 = gtk.VBox(False, 0)
95          hbox.pack_start(vbox2, True, True, 5)
```

```
 96
 97         label = gtk.Label("Value :")
 98         label.set_alignment(0, 0.5)
 99         vbox2.pack_start(label, False, True, 0)
100
101         adj = gtk.Adjustment(0.0, -10000.0, 10000.0, 0.5, 100.0, 0.0)
102         spinner1 = gtk.SpinButton(adj, 1.0, 2)
103         spinner1.set_wrap(True)
104         spinner1.set_size_request(100, -1)
105         vbox2.pack_start(spinner1, False, True, 0)
106
107         vbox2 = gtk.VBox(False, 0)
108         hbox.pack_start(vbox2, True, True, 5)
109
110         label = gtk.Label("Digits :")
111         label.set_alignment(0, 0.5)
112         vbox2.pack_start(label, False, True, 0)
113
114         adj = gtk.Adjustment(2, 1, 5, 1, 1, 0)
115         spinner2 = gtk.SpinButton(adj, 0.0, 0)
116         spinner2.set_wrap(True)
117         adj.connect("value_changed", self.change_digits, spinner2, spinner1 ↩
    )
118         vbox2.pack_start(spinner2, False, True, 0)
119
120         hbox = gtk.HBox(False, 0)
121         vbox.pack_start(hbox, False, True, 5)
122
123         button = gtk.CheckButton("Snap to 0.5-ticks")
124         button.connect("clicked", self.toggle_snap, spinner1)
125         vbox.pack_start(button, True, True, 0)
126         button.set_active(True)
127
128         button = gtk.CheckButton("Numeric only input mode")
129         button.connect("clicked", self.toggle_numeric, spinner1)
130         vbox.pack_start(button, True, True, 0)
131         button.set_active(True)
132
133         val_label = gtk.Label("")
134
135         hbox = gtk.HBox(False, 0)
136         vbox.pack_start(hbox, False, True, 5)
137         button = gtk.Button("Value as Int")
138         button.connect("clicked", self.get_value, 1, spinner1, spinner2,
139                        val_label)
140         hbox.pack_start(button, True, True, 5)
141
142         button = gtk.Button("Value as Float")
143         button.connect("clicked", self.get_value, 2, spinner1, spinner2,
144                        val_label)
145         hbox.pack_start(button, True, True, 5)
146
147         vbox.pack_start(val_label, True, True, 0)
148         val_label.set_text("0")
149
150         hbox = gtk.HBox(False, 0)
151         main_vbox.pack_start(hbox, False, True, 0)
152
153         button = gtk.Button("Close")
154         button.connect("clicked", lambda w: gtk.main_quit())
155         hbox.pack_start(button, True, True, 5)
156         window.show_all()
157
158 def main():
```

```
159     gtk.main()
160     return 0
161
162 if __name__ == "__main__":
163     SpinButtonExample()
164     main()
```

9.11 Combo Widget

NOTE



The Combo widget is deprecated in PyGTK 2.4 and above.

The Combo widget is another fairly simple widget that is really just a collection of other widgets. From the user's point of view, the widget consists of a text entry box and a pull down menu from which the user can select one of a set of predefined entries. Alternatively, the user can type a different option directly into the text box.

The Combo has two principal parts that you really care about: an *entry* and a *list*. These are accessed using the attributes:

```
combo.entry

combo.list
```

First off, to create a Combo, use:

```
combo = gtk.Combo()
```

Now, if you want to set the string in the entry section of the combo, this is done by manipulating the entry widget directly:

```
combo.entry.set_text(text)
```

To set the values in the popdown *list*, one uses the method:

```
combo.set_popdown_strings(strings)
```

Before you can do this, you have to assemble a list of the strings that you want.

Here's a typical code segment for creating a set of options:

```
slist = [ "String 1", "String 2", "String 3", "String 4" ]

combo.set_popdown_strings(slist)
```

At this point you have set up a working Combo. There are a few aspects of its behavior that you can change. These are accomplished with the methods:

```
combo.set_use_arrows(val)

combo.set_use_arrows_always(val)

combo.set_case_sensitive(val)
```

The set_use_arrows() method lets the user change the value in the entry using the up/down arrow keys when *val* is set to TRUE. This doesn't bring up the list, but rather replaces the current text in the entry with the next list entry (up or down, as your key choice indicates). It does this by searching in the list for the item corresponding to the current value in the *entry* and selecting the previous/next item accordingly. Usually in an *entry* the arrow keys are used to change focus (you can do that anyway

using **Tab**). Note that when the current item is the last of the list and you press arrow-down it changes the focus (the same applies with the first item and arrow-up).

If the current value in the *entry* is not in the list, then the set_use_arrows() method is disabled.

The set_use_arrows_always() method, when *val* is TRUE, similarly allows the use of the up/down arrow keys to cycle through the choices in the dropdown list, except that it wraps around the values in the list, completely disabling the use of the up and down arrow keys for changing focus.

The set_case_sensitive() method toggles whether or not GTK+ searches for entries in a case sensitive manner. This is used when the `Combo` widget is asked to find a value from the list using the current entry in the text box. This completion can be performed in either a case sensitive or insensitive manner, depending upon the setting of this method. The `Combo` widget can also simply complete the current entry if the user presses the key combination MOD-1-Tab. MOD-1 is often mapped to the **Alt** key, by the **xmodmap** utility. Note, however that some window managers also use this key combination, which will override its use within GTK.

Now that we have a combo, tailored to look and act how we want it, all that remains is being able to get data from the combo. This is relatively straightforward. The majority of the time, all you are going to care about getting data from is the entry. The entry is accessed simply as *combo.entry*. The two principal things that you are going to want to do with it are attach to the "activate" signal, which indicates that the user has pressed the **Return** or **Enter** key, and read the text. The first is accomplished using something like:

```
combo.entry.connect("activate", my_callback, my_data)
```

Getting the text at any arbitrary time is accomplished by simply using the entry method:

```
string = combo.entry.get_text()
```

That's about all there is to it. There is a method:

```
combo.disable_activate()
```

that will disable the activate signal on the entry widget in the combo. Personally, I can't think of why you'd want to use it, but it does exist.

9.12 Calendar

The `Calendar` widget is an effective way to display and retrieve monthly date related information. It is a very simple widget to create and work with.

Creating a `gtk.Calendar` widget is as simple as:

```
calendar = gtk.Calendar()
```

The calendar will display the current month and year by default.

There might be times where you need to change a lot of information within this widget and the following methods allow you to make multiple changes to a `Calendar` widget without the user seeing multiple on-screen updates.

```
calendar.freeze()

calendar.thaw()
```

They work just like the freeze/thaw methods of every other widget.

The `Calendar` widget has a few options that allow you to change the way the widget both looks and operates by using the method:

```
calendar.display_options(flags)
```

The *flags* argument can be formed by combining either of the following five options using the logical bitwise OR (|) operation:

CALENDAR_SHOW_HEADING	this option specifies that the month and year should be shown when drawing the calendar.
CALENDAR_SHOW_DAY_NAMES	this option specifies that the three letter descriptions should be displayed for each day (e.g. Mon,Tue, etc.).

CALENDAR_NO_MONTH_CHANGE	this option states that the user should not and can not change the currently displayed month. This can be good if you only need to display a particular month such as if you are displaying 12 calendar widgets for every month in a particular year.
CALENDAR_SHOW_WEEK_NUMBERS	this option specifies that the number for each week should be displayed down the left side of the calendar. (e.g. Jan 1 = Week 1,Dec 31 = Week 52).
CALENDAR_WEEK_START_MONDAY	this option states that the calender week will start on Monday instead of Sunday which is the default. This only affects the order in which days are displayed from left to right. Note that in PyGTK 2.4 and above this option is deprecated.

The following methods are used to set the the currently displayed date:

```
result = calendar.select_month(month, year)

calendar.select_day(day)
```

The return value from the select_month() method is a boolean value indicating whether the selection was successful.

With the select_day() method the specified day number is selected within the current month, if that is possible. A day value of 0 will deselect any current selection.

In addition to having a day selected, any number of days in the month may be "marked". A marked day is highlighted within the calendar display. The following methods are provided to manipulate marked days:

```
result = calendar.mark_day(day)

result = calendar.unmark_day(day)

calendar.clear_marks()
```

mark_day() and unmark_day() return a boolean indicating whether the method was successful. Note that marks are persistent across month and year changes.

The final `Calendar` widget method is used to retrieve the currently selected date, month and/or year.

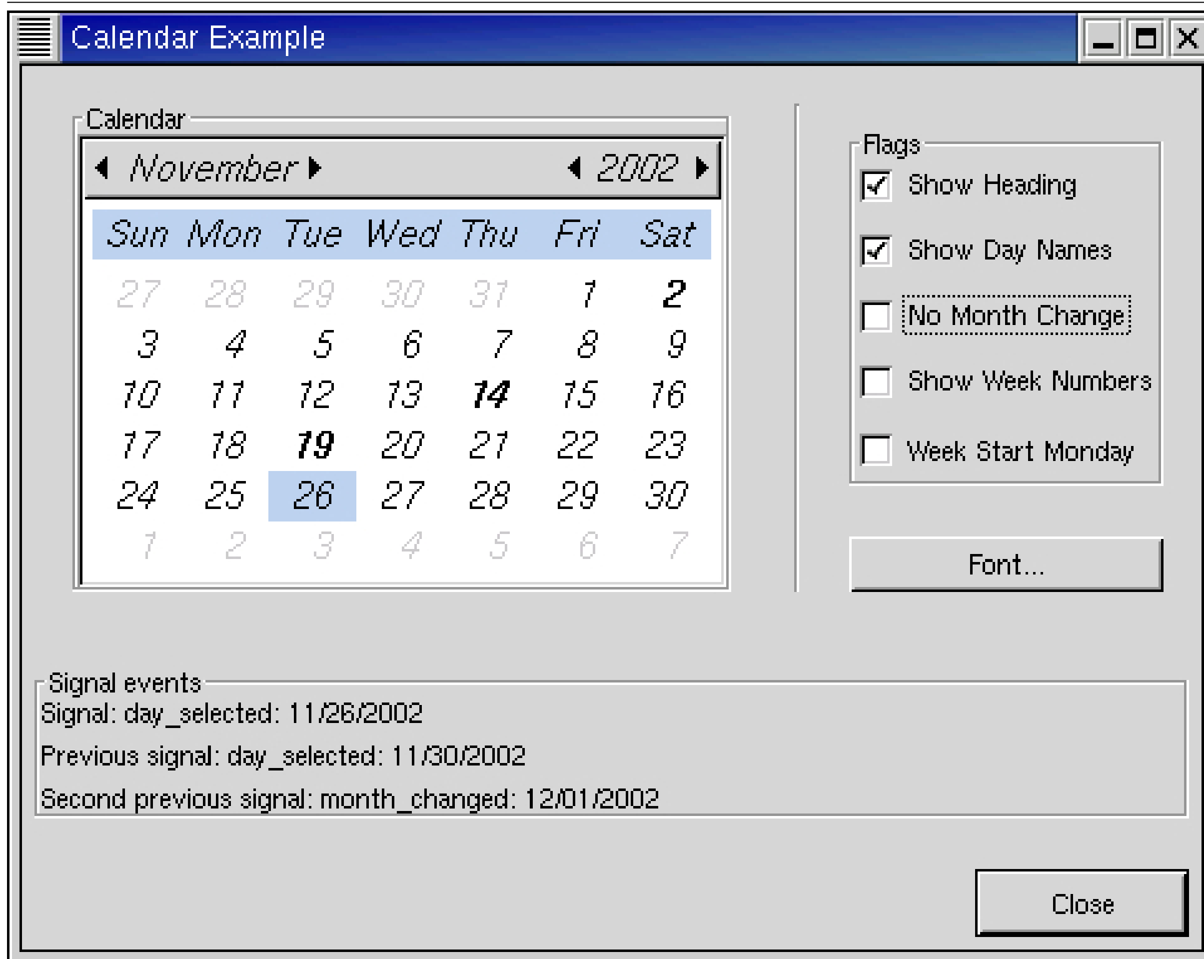
```
year, month, day = calendar.get_date()
```

The `Calendar` widget can generate a number of signals indicating date selection and change. The names of these signals are self explanatory, and are:

```
month_changed

day_selected

day_selected_double_click

prev_month

next_month

prev_year

next_year
```

That just leaves us with the need to put all of this together into the calendar.py example program. Figure 9.12 illustrates the program operation:

Figure 9.12 Calendar Example



The calendar.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example calendar.py
 4 #
 5 # Copyright (C) 1998 Cesar Miquel, Shawn T. Amundson, Mattias Gronlund
 6 # Copyright (C) 2000 Tony Gale
 7 # Copyright (C) 2001-2004 John Finlay
 8 #
 9 # This program is free software; you can redistribute it and/or modify
10 # it under the terms of the GNU General Public License as published by
11 # the Free Software Foundation; either version 2 of the License, or
12 # (at your option) any later version.
13 #
14 # This program is distributed in the hope that it will be useful,
15 # but WITHOUT ANY WARRANTY; without even the implied warranty of
16 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 # GNU General Public License for more details.
18 #
19 # You should have received a copy of the GNU General Public License
20 # along with this program; if not, write to the Free Software
21 # Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
22
23 import pygtk
24 pygtk.require('2.0')
25 import gtk, pango
26 import time
27
```

```
28 class CalendarExample:
29     DEF_PAD = 10
30     DEF_PAD_SMALL = 5
31     TM_YEAR_BASE = 1900
32 
33     calendar_show_header = 0
34     calendar_show_days = 1
35     calendar_month_change = 2
36     calendar_show_week = 3
37 
38     def calendar_date_to_string(self):
39         year, month, day = self.window.get_date()
40         mytime = time.mktime((year, month+1, day, 0, 0, 0, 0, 0, -1))
41         return time.strftime("%x", time.localtime(mytime))
42 
43     def calendar_set_signal_strings(self, sig_str):
44         prev_sig = self.prev_sig.get()
45         self.prev2_sig.set_text(prev_sig)
46 
47         prev_sig = self.last_sig.get()
48         self.prev_sig.set_text(prev_sig)
49         self.last_sig.set_text(sig_str)
50 
51     def calendar_month_changed(self, widget):
52         buffer = "month_changed: %s" % self.calendar_date_to_string()
53         self.calendar_set_signal_strings(buffer)
54 
55     def calendar_day_selected(self, widget):
56         buffer = "day_selected: %s" % self.calendar_date_to_string()
57         self.calendar_set_signal_strings(buffer)
58 
59     def calendar_day_selected_double_click(self, widget):
60         buffer = "day_selected_double_click: %s"
61         buffer = buffer % self.calendar_date_to_string()
62         self.calendar_set_signal_strings(buffer)
63 
64         year, month, day = self.window.get_date()
65 
66         if self.marked_date[day-1] == 0:
67             self.window.mark_day(day)
68             self.marked_date[day-1] = 1
69         else:
70             self.window.unmark_day(day)
71             self.marked_date[day-1] = 0
72 
73     def calendar_prev_month(self, widget):
74         buffer = "prev_month: %s" % self.calendar_date_to_string()
75         self.calendar_set_signal_strings(buffer)
76 
77     def calendar_next_month(self, widget):
78         buffer = "next_month: %s" % self.calendar_date_to_string()
79         self.calendar_set_signal_strings(buffer)
80 
81     def calendar_prev_year(self, widget):
82         buffer = "prev_year: %s" % self.calendar_date_to_string()
83         self.calendar_set_signal_strings(buffer)
84 
85     def calendar_next_year(self, widget):
86         buffer = "next_year: %s" % self.calendar_date_to_string()
87         self.calendar_set_signal_strings(buffer)
88 
89     def calendar_set_flags(self):
90         options = 0
91         for i in range(5):
```

```
 92                 if self.settings[i]:
 93                     options = options + (1<<i)
 94             if self.window:
 95                 self.window.display_options(options)
 96
 97         def calendar_toggle_flag(self, toggle):
 98             j = 0
 99             for i in range(5):
100                 if self.flag_checkboxes[i] == toggle:
101                     j = i
102
103             self.settings[j] = not self.settings[j]
104             self.calendar_set_flags()
105
106         def calendar_font_selection_ok(self, button):
107             self.font = self.font_dialog.get_font_name()
108             if self.window:
109                 font_desc = pango.FontDescription(self.font)
110                 if font_desc:
111                     self.window.modify_font(font_desc)
112
113         def calendar_select_font(self, button):
114             if not self.font_dialog:
115                 window = gtk.FontSelectionDialog("Font Selection Dialog")
116                 self.font_dialog = window
117
118                 window.set_position(gtk.WIN_POS_MOUSE)
119
120                 window.connect("destroy", self.font_dialog_destroyed)
121
122                 window.ok_button.connect("clicked",
123                                          self.calendar_font_selection_ok)
124                 window.cancel_button.connect_object("clicked",
125                                                     lambda wid: wid.destroy(),
126                                                     self.font_dialog)
127             window = self.font_dialog
128             if not (window.flags() & gtk.VISIBLE):
129                 window.show()
130             else:
131                 window.destroy()
132                 self.font_dialog = None
133
134         def font_dialog_destroyed(self, data=None):
135             self.font_dialog = None
136
137         def __init__(self):
138             flags = [
139                 "Show Heading",
140                 "Show Day Names",
141                 "No Month Change",
142                 "Show Week Numbers",
143                 ]
144             self.window = None
145             self.font = None
146             self.font_dialog = None
147             self.flag_checkboxes = 5*[None]
148             self.settings = 5*[0]
149             self.marked_date = 31*[0]
150
151             window = gtk.Window(gtk.WINDOW_TOPLEVEL)
152             window.set_title("Calendar Example")
153             window.set_border_width(5)
154             window.connect("destroy", lambda x: gtk.main_quit())
155
```

```
156         window.set_resizable(False)
157 
158         vbox = gtk.VBox(False, self.DEF_PAD)
159         window.add(vbox)
160 
161         # The top part of the window, Calendar, flags and fontsel.
162         hbox = gtk.HBox(False, self.DEF_PAD)
163         vbox.pack_start(hbox, True, True, self.DEF_PAD)
164         hbbox = gtk.HButtonBox()
165         hbox.pack_start(hbbox, False, False, self.DEF_PAD)
166         hbbox.set_layout(gtk.BUTTONBOX_SPREAD)
167         hbbox.set_spacing(5)
168 
169         # Calendar widget
170         frame = gtk.Frame("Calendar")
171         hbbox.pack_start(frame, False, True, self.DEF_PAD)
172         calendar = gtk.Calendar()
173         self.window = calendar
174         self.calendar_set_flags()
175         calendar.mark_day(19)
176         self.marked_date[19-1] = 1
177         frame.add(calendar)
178         calendar.connect("month_changed", self.calendar_month_changed)
179         calendar.connect("day_selected", self.calendar_day_selected)
180         calendar.connect("day_selected_double_click",
181                          self.calendar_day_selected_double_click)
182         calendar.connect("prev_month", self.calendar_prev_month)
183         calendar.connect("next_month", self.calendar_next_month)
184         calendar.connect("prev_year", self.calendar_prev_year)
185         calendar.connect("next_year", self.calendar_next_year)
186 
187         separator = gtk.VSeparator()
188         hbox.pack_start(separator, False, True, 0)
189 
190         vbox2 = gtk.VBox(False, self.DEF_PAD)
191         hbox.pack_start(vbox2, False, False, self.DEF_PAD)
192 
193         # Build the Right frame with the flags in
194         frame = gtk.Frame("Flags")
195         vbox2.pack_start(frame, True, True, self.DEF_PAD)
196         vbox3 = gtk.VBox(True, self.DEF_PAD_SMALL)
197         frame.add(vbox3)
198 
199         for i in range(len(flags)):
200             toggle = gtk.CheckButton(flags[i])
201             toggle.connect("toggled", self.calendar_toggle_flag)
202             vbox3.pack_start(toggle, True, True, 0)
203             self.flag_checkboxes[i] = toggle
204 
205         # Build the right font-button
206         button = gtk.Button("Font...")
207         button.connect("clicked", self.calendar_select_font)
208         vbox2.pack_start(button, False, False, 0)
209 
210         #  Build the Signal-event part.
211         frame = gtk.Frame("Signal events")
212         vbox.pack_start(frame, True, True, self.DEF_PAD)
213 
214         vbox2 = gtk.VBox(True, self.DEF_PAD_SMALL)
215         frame.add(vbox2)
216 
217         hbox = gtk.HBox (False, 3)
218         vbox2.pack_start(hbox, False, True, 0)
219         label = gtk.Label("Signal:")
```

```
220         hbox.pack_start(label, False, True, 0)
221         self.last_sig = gtk.Label("")
222         hbox.pack_start(self.last_sig, False, True, 0)
223 
224         hbox = gtk.HBox (False, 3)
225         vbox2.pack_start(hbox, False, True, 0)
226         label = gtk.Label("Previous signal:")
227         hbox.pack_start(label, False, True, 0)
228         self.prev_sig = gtk.Label("")
229         hbox.pack_start(self.prev_sig, False, True, 0)
230 
231         hbox = gtk.HBox (False, 3)
232         vbox2.pack_start(hbox, False, True, 0)
233         label = gtk.Label("Second previous signal:")
234         hbox.pack_start(label, False, True, 0)
235         self.prev2_sig = gtk.Label("")
236         hbox.pack_start(self.prev2_sig, False, True, 0)
237 
238         bbox = gtk.HButtonBox ()
239         vbox.pack_start(bbox, False, False, 0)
240         bbox.set_layout(gtk.BUTTONBOX_END)
241 
242         button = gtk.Button("Close")
243         button.connect("clicked", lambda w: gtk.main_quit())
244         bbox.add(button)
245         button.set_flags(gtk.CAN_DEFAULT)
246         button.grab_default()
247 
248         window.show_all()
249 
250 def main():
251     gtk.main()
252     return 0
253 
254 if __name__ == "__main__":
255     CalendarExample()
256     main()
```

9.13 Color Selection

The color selection widget is, not surprisingly, a widget for interactive selection of colors. This composite widget lets the user select a color by manipulating RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value) triples. This is done either by adjusting single values with sliders or entries, or by picking the desired color from a hue-saturation wheel/value bar. Optionally, the opacity of the color can also be set.

The color selection widget currently emits only one signal, "color_changed", which is emitted whenever the current color in the widget changes, either when the user changes it or if it's set explicitly through the set_color() method.

Lets have a look at what the color selection widget has to offer us. The widget comes in two flavors: `gtk.ColorSelection` and `gtk.ColorSelectionDialog`.

```
colorsel = gtk.ColorSelection()
```

You'll probably not be using this constructor directly. It creates an orphan `ColorSelection` widget which you'll have to parent yourself. The `ColorSelection` widget inherits from the `VBox` widget.

```
colorseldlg = gtk.ColorSelectionDialog(title)
```

where *`title`* is a string to be used in the titlebar of the dialog.

This is the most common color selection constructor. It creates a `ColorSelectionDialog`. It consists of a `Frame` containing a `ColorSelection` widget, an `HSeparator` and an `HBox` with three buttons, Ok, Cancel and Help. You can reach these buttons by accessing the *`ok_button`*, *`cancel_button`*

and *help_button* attributes of the ColorSelectionDialog, (i.e. *colorseldlg.ok_button*). The ColorSelection widget is accessed using the attribute *colorsel*:

```
colorsel = colorseldlg.colorsel
```

The ColorSelection widget has a number of methods that change its characteristics or provide access to the color selection.

```
colorsel.set_has_opacity_control(has_opacity)
```

The color selection widget supports adjusting the opacity of a color (also known as the alpha channel). This is disabled by default. Calling this method with *has_opacity* set to TRUE enables opacity. Likewise, *has_opacity* set to FALSE will disable opacity.

```
colorsel.set_current_color(color)
colorsel.set_current_alpha(alpha)
```

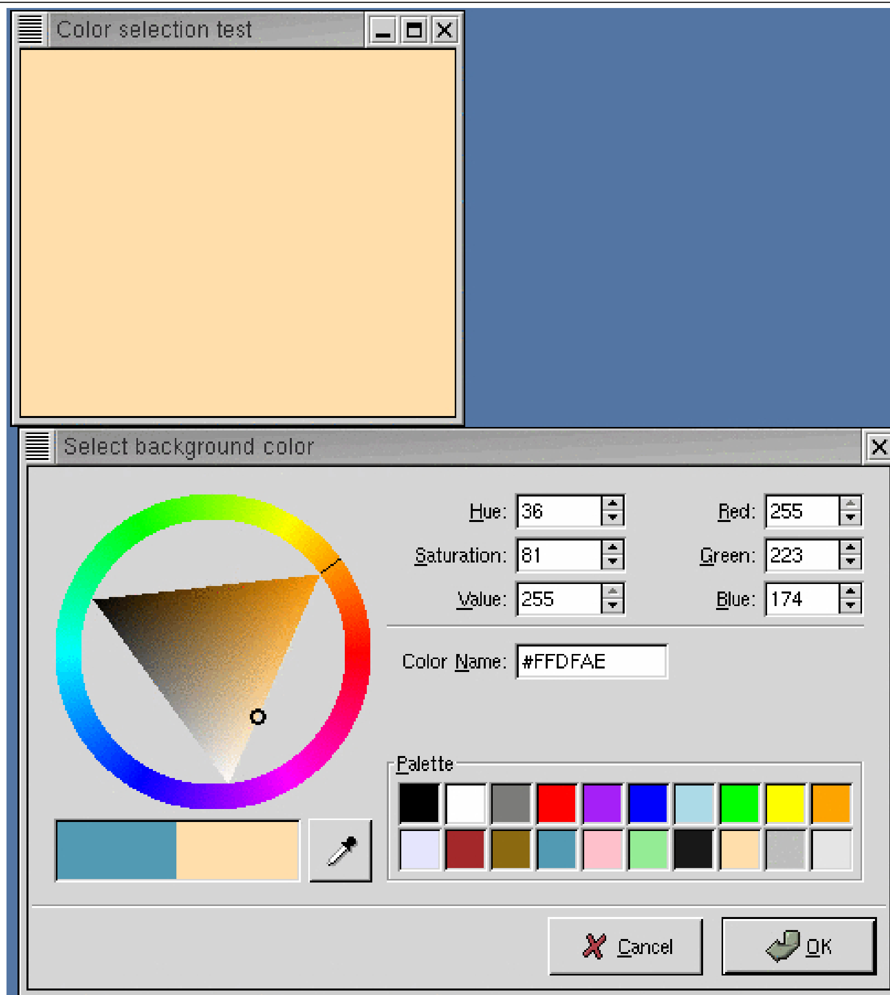
You can set the current color explicitly by calling the set_current_color() method with a gtk.gdk.Color. Setting the opacity (alpha channel) is done with the set_current_alpha() method. The *alpha* value should be between 0 (fully transparent) and 65636 (fully opaque).

```
color = colorsel.get_current_color()
alpha = colorsel.get_current_alpha()
```

When you need to query the current color, typically when you've received a "color_changed" signal, you use these methods.

The colorsel.py example program demonstrates the use of the ColorSelectionDialog. The program displays a window containing a drawing area. Clicking on it opens a color selection dialog, and changing the color in the color selection dialog changes the background color. Figure 9.13 illustrates this program in action:

Figure 9.13 Color Selection Dialog Example



The source code for colorsel.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example colorsel.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ColorSelectionExample:
10     # Color changed handler
11     def color_changed_cb(self, widget):
12         # Get drawingarea colormap
13         colormap = self.drawingarea.get_colormap()
14
15         # Get current color
16         color = self.colorseldlg.colorsel.get_current_color()
17
18         # Set window background color
19         self.drawingarea.modify_bg(gtk.STATE_NORMAL, color)
20
21     # Drawingarea event handler
22     def area_event(self, widget, event):
23         handled = False
24
25         # Check if we've received a button pressed event
```

```
26          if event.type == gtk.gdk.BUTTON_PRESS:
27              handled = True
28
29              # Create color selection dialog
30              if self.colorseldlg == None:
31                  self.colorseldlg = gtk.ColorSelectionDialog(
32                      "Select background color")
33
34              # Get the ColorSelection widget
35              colorsel = self.colorseldlg.colorsel
36
37              colorsel.set_previous_color(self.color)
38              colorsel.set_current_color(self.color)
39              colorsel.set_has_palette(True)
40
41              # Connect to the "color_changed" signal
42              colorsel.connect("color_changed", self.color_changed_cb)
43              # Show the dialog
44              response = self.colorseldlg.run()
45
46              if response -- gtk.RESPONSE_OK:
47                  self.color = colorsel.get_current_color()
48              else:
49                  self.drawingarea.modify_bg(gtk.STATE_NORMAL, self.color)
50
51              self.colorseldlg.hide()
52
53          return handled
54
55      # Close down and exit handler
56      def destroy_window(self, widget, event):
57          gtk.main_quit()
58          return True
59
60      def __init__(self):
61          self.colorseldlg = None
62          # Create toplevel window, set title and policies
63          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
64          window.set_title("Color selection test")
65          window.set_resizable(True)
66
67          # Attach to the "delete" and "destroy" events so we can exit
68          window.connect("delete_event", self.destroy_window)
69
70          # Create drawingarea, set size and catch button events
71          self.drawingarea = gtk.DrawingArea()
72
73          self.color = self.drawingarea.get_colormap().alloc_color(0, 65535,  ↩
    0)
74
75          self.drawingarea.set_size_request(200, 200)
76          self.drawingarea.set_events(gtk.gdk.BUTTON_PRESS_MASK)
77          self.drawingarea.connect("event",  self.area_event)
78
79          # Add drawingarea to window, then show them both
80          window.add(self.drawingarea)
81          self.drawingarea.show()
82          window.show()
83
84  def main():
85      gtk.main()
86      return 0
87
88  if __name__ == "__main__":
```

```
89    ColorSelectionExample()
90    main()
```

9.14 File Selections

The file selection widget is a quick and simple way to display a File dialog box. It comes complete with Ok, Cancel, and Help buttons, a great way to cut down on programming time.

To create a new file selection box use:

```
filesel = gtk.FileSelection(title=None)
```

To set the filename, for example to bring up a specific directory, or give a default filename, use this method:

```
filesel.set_filename(filename)
```

To grab the filename text that the user has entered or clicked on, use this method:

```
filename = filesel.get_filename()
```

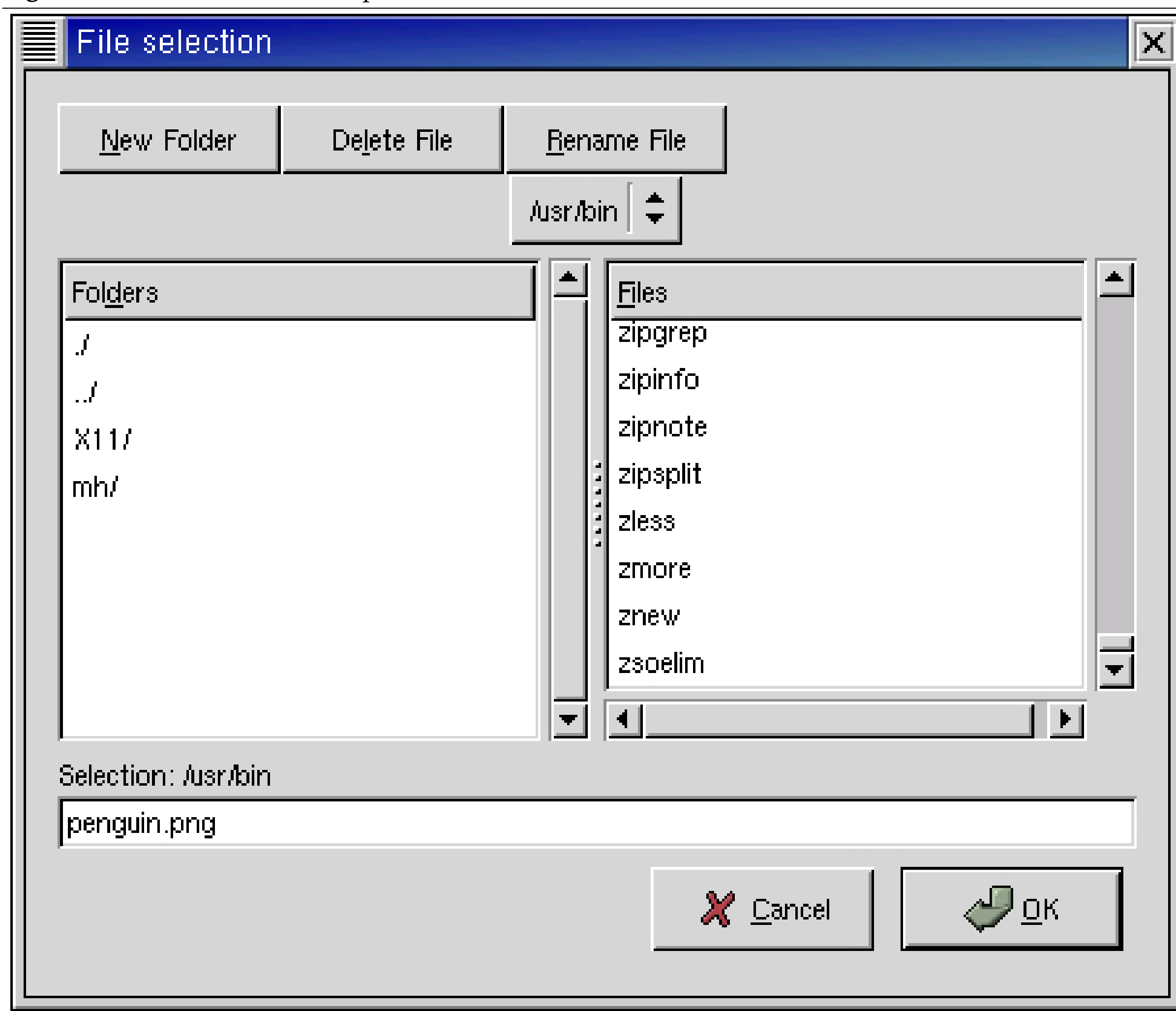
There are also references to the widgets contained within the file selection widget. These are the filesel attributes:

```
filesel.dir_list
filesel.file_list
filesel.selection_entry
filesel.selection_text
filesel.main_vbox
filesel.ok_button
filesel.cancel_button
filesel.help_button
filesel.history_pulldown
filesel.history_menu
filesel.fileop_dialog
filesel.fileop_entry
filesel.fileop_file
filesel.fileop_c_dir
filesel.fileop_del_file
filesel.fileop_ren_file
filesel.button_area
filesel.action_area
```

Most likely you will want to use the `ok_button`, `cancel_button`, and `help_button` attributes to connect their widget signals to callbacks.

The filesel.py example program illustrates the use of the FileSelection widget. As you will see, there is nothing much to creating a file selection widget. While in this example the Help button appears on the screen, it does nothing as there is not a signal attached to it. Figure 9.14 shows the resulting display:

Figure 9.14 File Selection Example



The source code for filesel.py is:

```
1 #!/usr/bin/env python
2
3 # example filesel.py
4
5 import pygtk
6 pygtk.require('2.0')
7 import gtk
8
9 class FileSelectionExample:
10     # Get the selected filename and print it to the console
11     def file_ok_sel(self, w):
12         print "%s" % self.filew.get_filename()
13
14     def destroy(self, widget):
15         gtk.main_quit()
16
17     def __init__(self):
18         # Create a new file selection widget
19         self.filew = gtk.FileSelection("File selection")
20
21         self.filew.connect("destroy", self.destroy)
22         # Connect the ok_button to file_ok_sel method
23         self.filew.ok_button.connect("clicked", self.file_ok_sel)
24
25         # Connect the cancel_button to destroy the widget
```

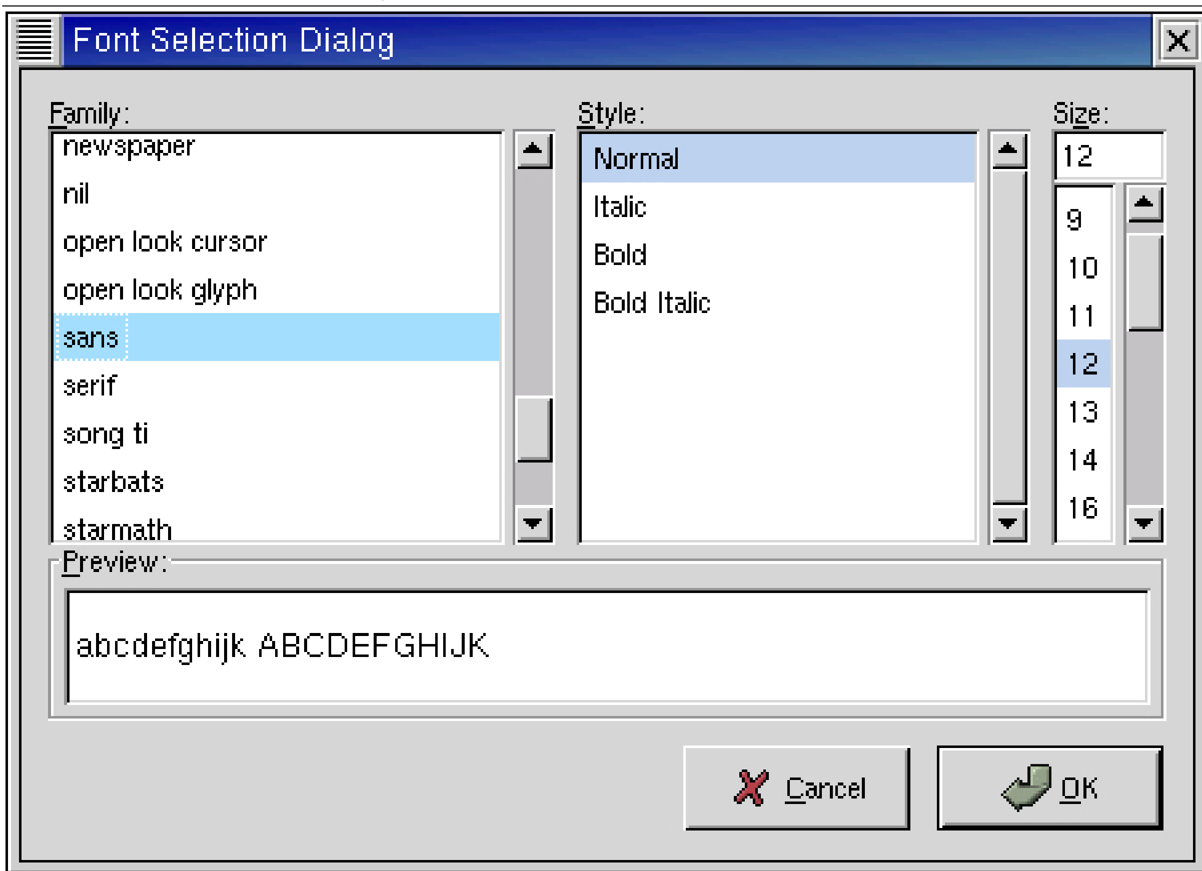
```
26          self.filew.cancel_button.connect("clicked",
27                                           lambda w: self.filew.destroy())
28
29          # Lets set the filename, as if this were a save dialog,
30          # and we are giving a default filename
31          self.filew.set_filename("penguin.png")
32
33          self.filew.show()
34
35 def main():
36      gtk.main()
37      return 0
38
39 if __name__ == "__main__":
40      FileSelectionExample()
41      main()
```

9.15 Font Selection Dialog

The Font Selection Dialog allows a user to interactively select a font for use within your program. The dialog contains a `FontSelection` widget and OK and Cancel buttons. An Apply button is also available in the dialog but is initially hidden. The Font Selection Dialog allows a user to select a font from the available system fonts (the same ones that can be retrieved using **xlsfonts**).

Figure 9.15 illustrates the `FontSelectionDialog` display:

Figure 9.15 Font Selection Dialog



The dialog contains a set of three notebook pages that provide:

- an interface to select the font, font style and font size

- detailed information about the currently selected font
- an interface to the font filter mechanism that restricts the fonts available for selection

The function to create a `FontSelectionDialog` is:

```
fontseldlg = gtk.FontSelectionDialog(title)
```

The *title* is a string that will be used to set the titlebar text.
A Font Selection Dialog instance has several attributes:

```
fontsel
main_vbox
action_area
ok_button
apply_button
cancel_button
```

The *fontsel* attribute provides a reference to the Font Selection widget. *main_vbox* is a reference to the `gtk.VBox` containing the *fontsel* and the *action_area* in the dialog. The *action_area* attribute is a reference to the `gtk.HButtonBox` that contains the OK, Apply and Cancel buttons. The *ok_button*, *cancel_button* and *apply_button* attributes provide references to the OK, Apply and Cancel buttons that can be used to set connections to the button signals. The *apply_button* reference can also be used to show() the Apply button.
You can set the initial font to be displayed in the *fontseldlg* by using the method:

```
fontseldlg.set_font_name(fontname)
```

The *fontname* argument is the name of a completely specified or partially specified system font. For example:

```
fontseldlg.set_font_name('-adobe-courier-bold-*-*-*-*-120-*-*-*-*-*-*')
```

partially specifies the initial font.
The font name of the currently selected font can be retrieved using the method:

```
font_name = fontseldlg.get_font_name()
```

The Font Selection Dialog has a Preview area that displays text using the currently selected font. The text that is used in the Preview area can be set with the method:

```
fontseldlg.set_preview_text(text)
```

The preview text can be retrieved with the method:

```
text = fontseldlg.get_preview_text()
```

The calendar.py example program uses a Font Selection Dialog to select the font to display the calendar information. Lines 105-110 define a callback that retrieves the font name from the Font Selection Dialog and uses it to set the font for the calendar widget. Lines 112-131 defines the method that creates the Font Selection Dialog, sets up the callbacks for the OK and Cancel buttons and displays the dialog.

Chapter 10

Container Widgets

10.1 The EventBox

Some GTK widgets don't have associated X windows, so they just draw on their parents. Because of this, they cannot receive events and if they are incorrectly sized, they don't clip so you can get messy overwriting, etc. If you require more from these widgets, the `EventBox` is for you.

At first glance, the `EventBox` widget might appear to be totally useless. It draws nothing on the screen and responds to no events. However, it does serve a function - it provides an X window for its child widget. This is important as many GTK widgets do not have an associated X window. Not having an X window saves memory and improves performance, but also has some drawbacks. A widget without an X window cannot receive events, does not perform any clipping on its contents and cannot set its background color. Although the name `EventBox` emphasizes the event-handling function, the widget can also be used for clipping. (and more, see the example below).

To create a new `EventBox` widget, use:

```
event_box = gtk.EventBox()
```

A child widget can then be added to this *event_box*:

```
event_box.add(widget)
```

The eventbox.py example program demonstrates both uses of an `EventBox` - a label is created that is clipped to a small box, has a green background and is set up so that a mouse-click on the label causes the program to exit. Resizing the window reveals varying amounts of the label. Figure 10.1 illustrates the programs display:

Figure 10.1 Event Box Example

The source code for eventbox.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example eventbox.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class EventBoxExample:
10     def __init__(self):
11         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

```
12          window.set_title("Event Box")
13          window.connect("destroy", lambda w: gtk.main_quit())
14          window.set_border_width(10)
15
16          # Create an EventBox and add it to our toplevel window
17          event_box = gtk.EventBox()
18          window.add(event_box)
19          event_box.show()
20
21          # Create a long label
22          label = gtk.Label("Click here to quit, quit, quit, quit, quit")
23          event_box.add(label)
24          label.show()
25
26          # Clip it short.
27          label.set_size_request(110, 20)
28
29          # And bind an action to it
30          event_box.set_events(gtk.gdk.BUTTON_PRESS_MASK)
31          event_box.connect("button_press_event", lambda w,e: gtk.main_quit() ↩
    )
32
33          # More things you need an X window for ...
34          event_box.realize()
35          event_box.window.set_cursor(gtk.gdk.Cursor(gtk.gdk.HAND1))
36
37          # Set background color to green
38          event_box.modify_bg(gtk.STATE_NORMAL,
39                              event_box.get_colormap().alloc_color("green"))
40
41          window.show()
42
43  def main():
44      gtk.main()
45      return 0
46
47  if __name__ == "__main__":
48      EventBoxExample()
49      main()
```

10.2 The Alignment widget

The `Alignment` widget allows you to place a widget within its window at a position and size relative to the size of the `Alignment` widget itself. For example, it can be very useful for centering a widget within the window.

There are only two calls associated with the `Alignment` widget:

```
alignment = gtk.Alignment(xalign=0.0, yalign=0.0, xscale=0.0, yscale=0.0)

alignment.set(xalign, yalign, xscale, yscale)
```

The `gtk.Alignment()` function creates a new `Alignment` widget with the specified parameters. The set() method allows the alignment parameters of an existing `Alignment` widget to be altered.

All four alignment parameters are floating point numbers which can range from 0.0 to 1.0. The *`xalign`* and *`yalign`* arguments affect the position of the widget placed within the gtk.Alignment widget. The align properties specify the fraction of *free* space that will be placed above or to the left of the child widget. The values range from 0.0 (no *free* space above or to the left of the child) to 1.0 (all *free* space above or to the left of the child). Of course, if the scale properties are both set to 1.0, the alignment properties have no effect since the child widget will expand to fill the available space.

The *`xscale`* and *`yscale`* arguments specify the fraction of *free* space absorbed by the child widget. The values can range from 0.0 (meaning the child absorbs none) to 1.0 (meaning the child absorbs all of the *free* space).

A child widget can be added to this `Alignment` widget using:

```
alignment.add(widget)
```

For an example of using an `Alignment` widget, refer to the progressbar.py example for the Progress Bar widget.

10.3 Fixed Container

The `Fixed` container allows you to place widgets at a fixed position within it's window, relative to it's upper left hand corner. The position of the widgets can be changed dynamically.

There are only three calls associated with the fixed widget:

```
fixed = gtk.Fixed()

fixed.put(widget, x, y)

fixed.move(widget, x, y)
```

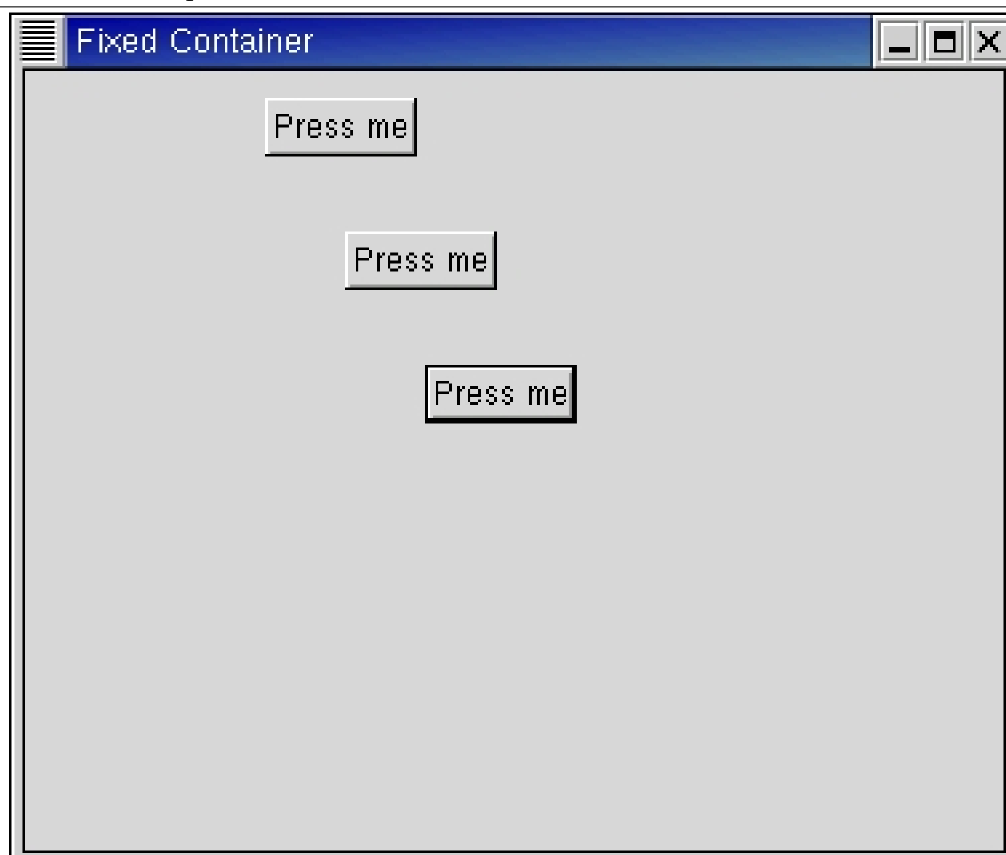
The function `gtk.Fixed()` allows you to create a new `Fixed` container.

The put() method places widget in the container fixed at the position specified by *x* and *y*.

The move() method allows the specified widget to be moved to a new position.

The fixed.py example illustrates how to use the `Fixed` container. Figure 10.2 shows the result:

Figure 10.2 Fixed Example



The source code for fixed.py:

```
1 #!/usr/bin/env python
2
3 # example fixed.py
4
5 import pygtk
```

```
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class FixedExample:
10     # This callback method moves the button to a new position
11     # in the Fixed container.
12     def move_button(self, widget):
13         self.x = (self.x+30)%300
14         self.y = (self.y+50)%300
15         self.fixed.move(widget, self.x, self.y)
16
17     def __init__(self):
18         self.x = 50
19         self.y = 50
20
21         # Create a new window
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23         window.set_title("Fixed Container")
24
25         # Here we connect the "destroy" event to a signal handler
26         window.connect("destroy", lambda w: gtk.main_quit())
27
28         # Sets the border width of the window.
29         window.set_border_width(10)
30
31         # Create a Fixed Container
32         self.fixed = gtk.Fixed()
33         window.add(self.fixed)
34         self.fixed.show()
35
36         for i in range(1, 4):
37             # Creates a new button with the label "Press me"
38             button = gtk.Button("Press me")
39
40             # When the button receives the "clicked" signal, it will call  ↩
   the
41             # method move_button().
42             button.connect("clicked", self.move_button)
43
44             # This packs the button into the fixed containers window.
45             self.fixed.put(button, i*50, i*50)
46
47             # The final step is to display this newly created widget.
48             button.show()
49
50         # Display the window
51         window.show()
52
53 def main():
54     # Enter the event loop
55     gtk.main()
56     return 0
57
58 if __name__ == "__main__":
59     FixedExample()
60     main()
```

10.4 Layout Container

The `Layout` container is similar to the `Fixed` container except that it implements an infinite (where infinity is less than 2^32) scrolling area. The X window system has a limitation where windows can be at most 32767 pixels wide or tall. The `Layout` container gets around this limitation by doing some exotic

stuff using window and bit gravities, so that you can have smooth scrolling even when you have many child widgets in your scrolling area.

A `Layout` container is created using:

```
layout = gtk.Layout(hadjustment=None, vadjustment=None)
```

As you can see, you can optionally specify the `Adjustment` objects (see Chapter 7) that the `Layout` widget will use for its scrolling. If you don't specify the `Adjustment` objects, new ones will be created.

You can add and move widgets in the `Layout` container using the following two methods:

```
layout.put(child_widget, x, y)

layout.move(child_widget, x, y)
```

The size of the `Layout` container can be set and retrieved using the next methods:

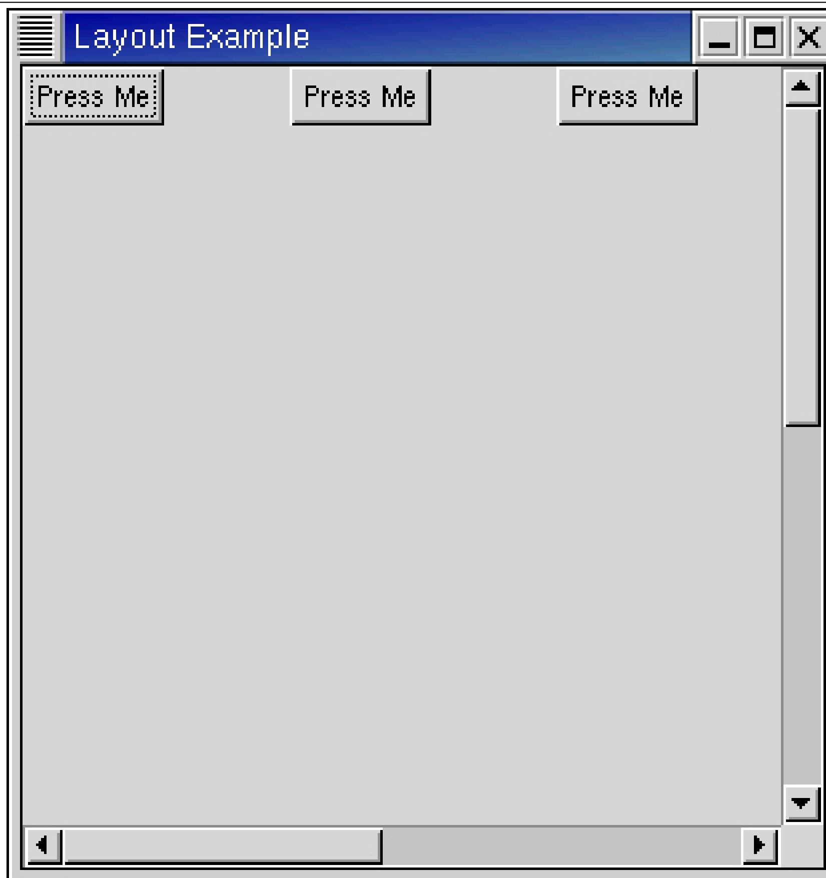
```
layout.set_size(width, height)

size = layout.get_size()
```

The final four methods for use with `Layout` widgets are for manipulating the horizontal and vertical adjustment widgets:

```
hadj = layout.get_hadjustment()

vadj = layout.get_vadjustment()

layout.set_hadjustment(adjustment)

layout.set_vadjustment(adjustment)
```

The layout.py example program creates three buttons and puts them in a layout widget. when a button is clicked, it is moved to a random location in the layout. Figure 10.3 illustrates the starting display of the program:

Figure 10.3 Layout Example



The layout.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example layout.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8 import random
 9
10 class LayoutExample:
11     def WindowDeleteEvent(self, widget, event):
12         # return false so that window will be destroyed
13         return False
14
15     def WindowDestroy(self, widget, *data):
16         # exit main loop
17         gtk.main_quit()
18
19     def ButtonClicked(self, button):
20         # move the button
21         self.layout.move(button, random.randint(0,500),
22                          random.randint(0,500))
23
24     def __init__(self):
25         # create the top level window
26         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
27         window.set_title("Layout Example")
28         window.set_default_size(300, 300)
29         window.connect("delete-event", self.WindowDeleteEvent)
```

```
30          window.connect("destroy", self.WindowDestroy)
31          # create the table and pack into the window
32          table = gtk.Table(2, 2, False)
33          window.add(table)
34          # create the layout widget and pack into the table
35          self.layout = gtk.Layout(None, None)
36          self.layout.set_size(600, 600)
37          table.attach(self.layout, 0, 1, 0, 1, gtk.FILL|gtk.EXPAND,
38                       gtk.FILL|gtk.EXPAND, 0, 0)
39          # create the scrollbars and pack into the table
40          vScrollbar = gtk.VScrollbar(None)
41          table.attach(vScrollbar, 1, 2, 0, 1, gtk.FILL|gtk.SHRINK,
42                       gtk.FILL|gtk.SHRINK, 0, 0)
43          hScrollbar = gtk.HScrollbar(None)
44          table.attach(hScrollbar, 0, 1, 1, 2, gtk.FILL|gtk.SHRINK,
45                       gtk.FILL|gtk.SHRINK, 0, 0)
46          # tell the scrollbars to use the layout widget's adjustments
47          vAdjust = self.layout.get_vadjustment()
48          vScrollbar.set_adjustment(vAdjust)
49          hAdjust = self.layout.get_hadjustment()
50          hScrollbar.set_adjustment(hAdjust)
51          # create 3 buttons and put them into the layout widget
52          button = gtk.Button("Press Me")
53          button.connect("clicked", self.ButtonClicked)
54          self.layout.put(button, 0, 0)
55          button = gtk.Button("Press Me")
56          button.connect("clicked", self.ButtonClicked)
57          self.layout.put(button, 100, 0)
58          button = gtk.Button("Press Me")
59          button.connect("clicked", self.ButtonClicked)
60          self.layout.put(button, 200, 0)
61          # show all the widgets
62          window.show_all()
63
64  def main():
65      # enter the main loop
66      gtk.main()
67      return 0
68
69  if __name__ == "__main__":
70      LayoutExample()
71      main()
```

10.5 Frames

Frames can be used to enclose one or a group of widgets with a box which can optionally be labeled. The position of the label and the style of the box can be altered to suit.

A `Frame` can be created with the following function:

```
frame = gtk.Frame(label=None)
```

The *label* is by default placed in the upper left hand corner of the frame. Specifying a value of None for the *label* argument or specifying no *label* argument will result in no label being displayed. The text of the label can be changed using the method.

```
frame.set_label(label)
```

The position of the label can be changed using this method:

```
frame.set_label_align(xalign, yalign)
```

xalign and *yalign* take values between 0.0 and 1.0. *xalign* indicates the position of the label along the top horizontal of the frame. *yalign* is not currently used. The default value of *xalign* is 0.0 which places the label at the left hand end of the frame.

The next method alters the style of the box that is used to outline the frame.

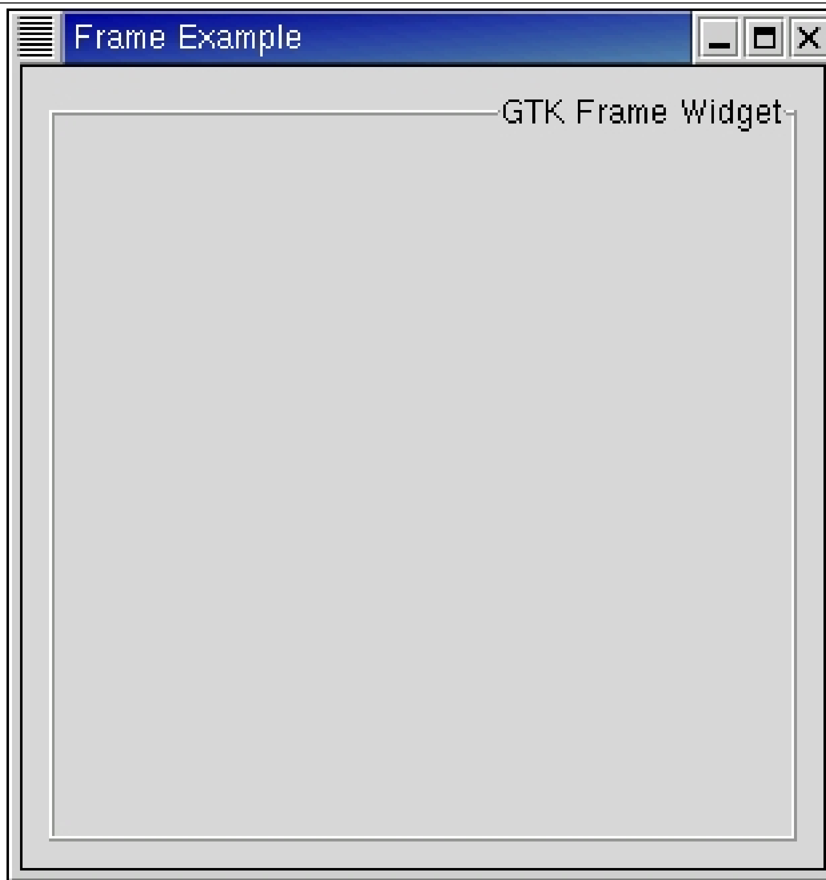
```
frame.set_shadow_type(type)
```

The *type* argument can take one of the following values:

```
SHADOW_NONE
SHADOW_IN
SHADOW_OUT
SHADOW_ETCHED_IN      # the default
SHADOW_ETCHED_OUT
```

The frame.py example illustrates the use of the Frame widget. Figure 10.4 shows the resulting display:

Figure 10.4 Frame Example



The source code of frame.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example frame.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class FrameExample:
10     def __init__(self):
11         # Create a new window
12         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
13         window.set_title("Frame Example")
14
15         # Here we connect the "destroy" event to a signal handler
```

```
16          window.connect("destroy", lambda w: gtk.main_quit())
17          window.set_size_request(300, 300)
18
19          # Sets the border width of the window.
20          window.set_border_width(10)
21
22          # Create a Frame
23          frame = gtk.Frame()
24          window.add(frame)
25
26          # Set the frame's label
27          frame.set_label("GTK Frame Widget")
28
29          # Align the label at the right of the frame
30          frame.set_label_align(1.0, 0.0)
31
32          # Set the style of the frame
33          frame.set_shadow_type(gtk.SHADOW_ETCHED_OUT)
34          frame.show()
35
36          # Display the window
37          window.show()
38
39  def main():
40      # Enter the event loop
41      gtk.main()
42      return 0
43
44  if __name__ == "__main__":
45      FrameExample()
46      main()
```

The calendar.py, label.py and spinbutton.py examples also use Frames.

10.6 Aspect Frames

The aspect frame widget is like a frame widget, except that it also enforces the aspect ratio (that is, the ratio of the width to the height) of the child widget to have a certain value, adding extra space if necessary. This is useful, for instance, if you want to preview a larger image. The size of the preview should vary when the user resizes the window, but the aspect ratio needs to always match the original image.

To create a new aspect frame use:

```
aspect_frame = gtk.AspectFrame(label=None, xalign=0.5, yalign=0.5, ratio=1.0, ↩
    obey_child=TRUE)
```

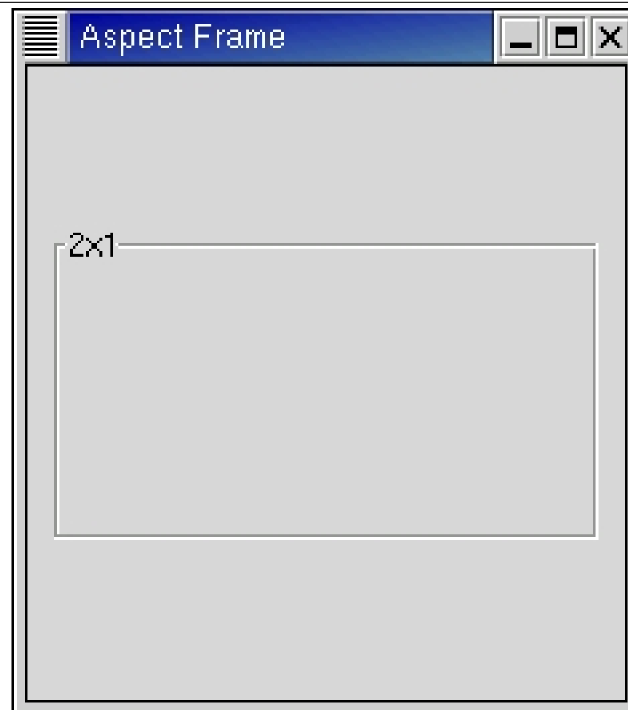
label specifies the text to be displayed as the label. *xalign* and *yalign* specify alignment as with gtk.Alignment widgets. If *obey_child* is TRUE, the aspect ratio of a child widget will match the aspect ratio of the ideal size it requests. Otherwise, it is given by *ratio*.

To change the options of an existing aspect frame, you can use:

```
aspect_frame.set(xalign=0.0, yalign=0.0, ratio=1.0, obey_child=TRUE)
```

As an example, the aspectframe.py program uses an AspectFrame to present a drawing area whose aspect ratio will always be 2:1, no matter how the user resizes the top-level window. Figure 10.5 illustrates the display of the program:

Figure 10.5 Aspect Frame Example



The source code for aspectframe.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example aspectframe.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class AspectFrameExample:
10     def __init__(self):
11         window = gtk.Window(gtk.WINDOW_TOPLEVEL);
12         window.set_title("Aspect Frame")
13         window.connect("destroy", lambda x: gtk.main_quit())
14         window.set_border_width(10)
15
16         # Create an aspect_frame and add it to our toplevel window
17         aspect_frame = gtk.AspectFrame("2x1", # label
18                                        0.5, # center x
19                                        0.5, # center y
20                                        2, # xsize/ysize = 2
21                                        False) # ignore child's aspect
22         window.add(aspect_frame)
23         aspect_frame.show()
24
25         # Now add a child widget to the aspect frame
26         drawing_area = gtk.DrawingArea()
27
28         # Ask for a 200x200 window, but the AspectFrame will give us a 200 ↩
   x100
29         # window since we are forcing a 2x1 aspect ratio
30         drawing_area.set_size_request(200, 200)
31         aspect_frame.add(drawing_area)
32         drawing_area.show()
33         window.show()
34
```

```
35 def main():
36     gtk.main()
37     return 0
38
39 if __name__ == "__main__":
40     AspectFrameExample()
41     main()
```

10.7 Paned Window Widgets

The paned window widgets are useful when you want to divide an area into two parts, with the relative size of the two parts controlled by the user. A groove is drawn between the two portions with a handle that the user can drag to change the ratio. The division can either be horizontal (`HPaned`) or vertical (`VPaned`).

To create a new paned window, call one of:

```
hpane = gtk.HPaned()

vpane = gtk.VPaned()
```

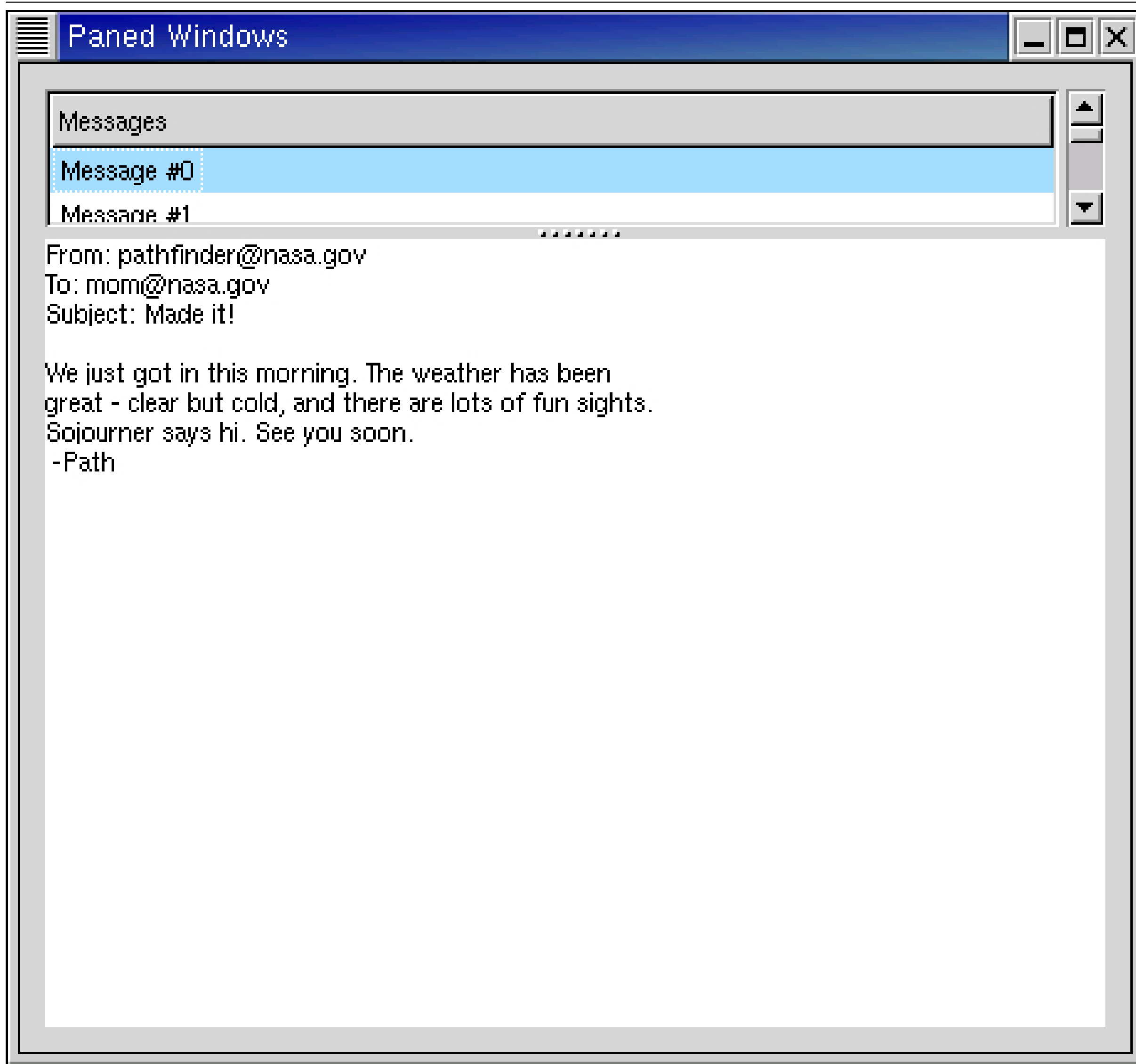
After creating the paned window widget, you need to add child widgets to its two halves. To do this, use the methods:

```
paned.add1(child)

paned.add2(child)
```

The add1() method adds the *child* widget to the left or top half of the paned window. The add2() method adds the *child* widget to the right or bottom half of the paned window.

The paned.py example program creates part of the user interface of an imaginary email program. A window is divided into two portions vertically, with the top portion being a list of email messages and the bottom portion the text of the email message. Most of the program is pretty straightforward. A couple of points to note: text can't be added to a Text widget until it is realized. This could be done by calling the realize() method, but as a demonstration of an alternate technique, we connect a handler to the "realize" signal to add the text. Also, we need to add the `SHRINK` option to some of the items in the table containing the text window and its scrollbars, so that when the bottom portion is made smaller, the correct portions shrink instead of being pushed off the bottom of the window. Figure 10.6 shows the result of running the program:

Figure 10.6 Paned Example



The source code of the paned.py program is:

```
 1 #!/usr/bin/env python
 2
 3 # example paned.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk, gobject
 8
 9 class PanedExample:
10     # Create the list of "messages"
11     def create_list(self):
12         # Create a new scrolled window, with scrollbars only if needed
13         scrolled_window = gtk.ScrolledWindow()
14         scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk. ↩
    POLICY_AUTOMATIC)
15
16         model = gtk.ListStore(gobject.TYPE_STRING)
17         tree_view = gtk.TreeView(model)
18         scrolled_window.add_with_viewport (tree_view)
19         tree_view.show()
20
```

```
21         # Add some messages to the window
22         for i in range(10):
23             msg = "Message #%d" % i
24             iter = model.append()
25             model.set(iter, 0, msg)
26
27         cell = gtk.CellRendererText()
28         column = gtk.TreeViewColumn("Messages", cell, text=0)
29         tree_view.append_column(column)
30
31         return scrolled_window
32
33     # Add some text to our text widget - this is a callback that is invoked
34     # when our window is realized. We could also force our window to be
35     # realized with gtk.Widget.realize(), but it would have to be part of a
36     # hierarchy first
37     def insert_text(self, buffer):
38         iter = buffer.get_iter_at_offset(0)
39         buffer.insert(iter,
40                       "From: pathfinder@nasa.gov\n"
41                       "To: mom@nasa.gov\n"
42                       "Subject: Made it!\n"
43                       "\n"
44                       "We just got in this morning. The weather has been\n"
45                       "great - clear but cold, and there are lots of fun ↩
   sights.\n"
46                       "Sojourner says hi. See you soon.\n"
47                       " -Path\n")
48
49     # Create a scrolled text area that displays a "message"
50     def create_text(self):
51         view = gtk.TextView()
52         buffer = view.get_buffer()
53         scrolled_window = gtk.ScrolledWindow()
54         scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk. ↩
   POLICY_AUTOMATIC)
55         scrolled_window.add(view)
56         self.insert_text(buffer)
57         scrolled_window.show_all()
58         return scrolled_window
59
60     def __init__(self):
61         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
62         window.set_title("Paned Windows")
63         window.connect("destroy", lambda w: gtk.main_quit())
64         window.set_border_width(10)
65         window.set_size_request(450, 400)
66
67         # create a vpaned widget and add it to our toplevel window
68         vpaned = gtk.VPaned()
69         window.add(vpaned)
70         vpaned.show()
71
72         # Now create the contents of the two halves of the window
73         list = self.create_list()
74         vpaned.add1(list)
75         list.show()
76
77         text = self.create_text()
78         vpaned.add2(text)
79         text.show()
80         window.show()
81
82 def main():
```

```
83      gtk.main()
84      return 0
85
86  if __name__ == "__main__":
87      PanedExample()
88      main()
```

10.8 Viewports

It is unlikely that you will ever need to use the `Viewport` widget directly. You are much more likely to use the `ScrolledWindow` widget *see Section 10.9) which in turn uses the `Viewport`.

A viewport widget allows you to place a larger widget within it such that you can view a part of it at a time. It uses `Adjustment` object (see Chapter 7) to define the area that is currently in view.

A `Viewport` is created with the function:

```
viewport = gtk.Viewport(hadjustment=None, vadjustment=None)
```

As you can see you can specify the horizontal and vertical `Adjustment` objects that the widget is to use when you create the widget. It will create its own if you pass `None` as the value of the arguments or pass no arguments.

You can get and set the adjustments after the widget has been created using the following four methods:

```
viewport.get_hadjustment()

viewport.get_vadjustment()

viewport.set_hadjustment(adjustment)

viewport.set_vadjustment(adjustment)
```

The only other viewport method is used to alter its appearance:

```
viewport.set_shadow_type(type)
```

Possible values for the *type* parameter are:

```
SHADOW_NONE
SHADOW_IN
SHADOW_OUT
SHADOW_ETCHED_IN
SHADOW_ETCHED_OUT
```

10.9 Scrolled Windows

Scrolled windows are used to create a scrollable area with another widget inside it. You may insert any type of widget into a scrolled window, and it will be accessible regardless of the size by using the scrollbars.

The following function is used to create a new scrolled window.

```
scrolled_window = gtk.ScrolledWindow(hadjustment=None, vadjustment=None)
```

Where the first argument is the adjustment for the horizontal direction, and the second, the adjustment for the vertical direction. These are almost always set to `None` or not specified.

```
scrolled_window.set_policy(hscrollbar_policy, vscrollbar_policy)
```

This method sets the policy to be used with respect to the scrollbars. The first argument sets the policy for the horizontal scrollbar, and the second, the policy for the vertical scrollbar.

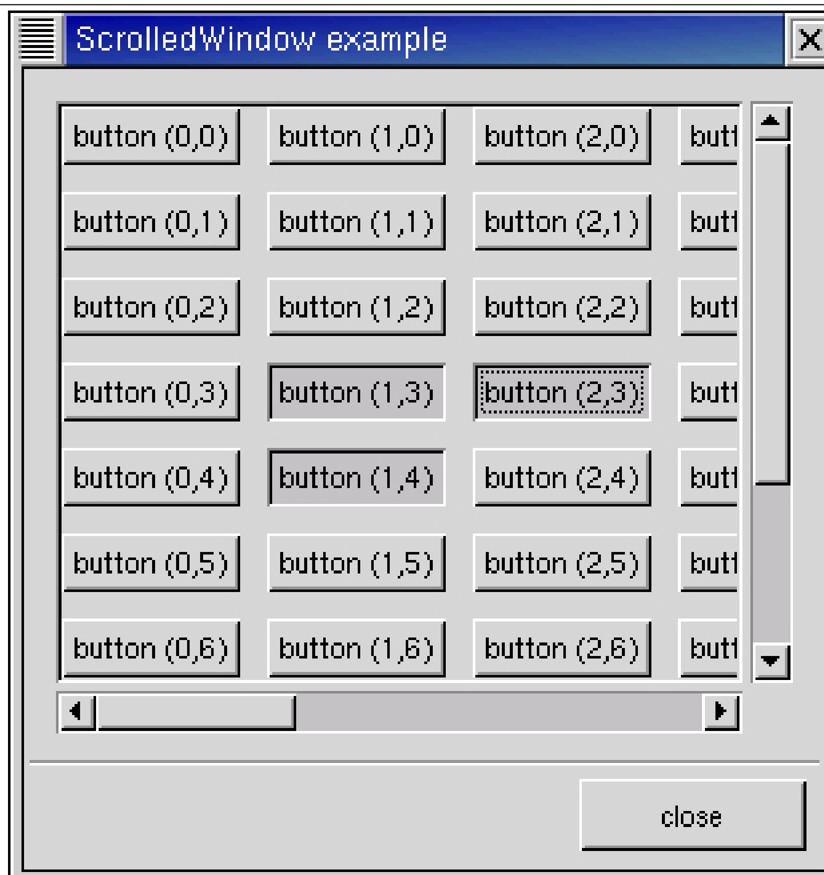
The policy may be one of `POLICY_AUTOMATIC` or `POLICY_ALWAYS`. `POLICY_AUTOMATIC` will automatically decide whether you need scrollbars, whereas `POLICY_ALWAYS` will always leave the scrollbars there.

You can then place your object into the scrolled window using the following method.

```
scrolled_window.add_with_viewport(child)
```

The scrolledwin.py example program packs a table with 100 toggle buttons into a scrolled window. I've only commented on the parts that may be new to you. Figure 10.7 illustrates the program display:

Figure 10.7 Scrolled Window Example



The source code for the scrolledwin.py program is:

```
 1 #!/usr/bin/env python
 2
 3 # example scrolledwin.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ScrolledWindowExample:
10     def destroy(self, widget):
11         gtk.main_quit()
12
13     def __init__(self):
14         # Create a new dialog window for the scrolled window to be
15         # packed into.
16         window = gtk.Dialog()
17         window.connect("destroy", self.destroy)
18         window.set_title("ScrolledWindow example")
19         window.set_border_width(0)
20         window.set_size_request(300, 300)
21
22         # create a new scrolled window.
23         scrolled_window = gtk.ScrolledWindow()
```

```
24         scrolled_window.set_border_width(10)
25
26         # the policy is one of POLICY AUTOMATIC, or POLICY_ALWAYS.
27         # POLICY_AUTOMATIC will automatically decide whether you need
28         # scrollbars, whereas POLICY_ALWAYS will always leave the  ↩
   scrollbars
29         # there. The first one is the horizontal scrollbar, the second, the
30         # vertical.
31         scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_ALWAYS)
32
33         # The dialog window is created with a vbox packed into it.
34         window.vbox.pack_start(scrolled_window, True, True, 0)
35         scrolled_window.show()
36
37         # create a table of 10 by 10 squares.
38         table = gtk.Table(10, 10, False)
39
40         # set the spacing to 10 on x and 10 on y
41         table.set_row_spacings(10)
42         table.set_col_spacings(10)
43
44         # pack the table into the scrolled window
45         scrolled_window.add_with_viewport(table)
46         table.show()
47
48         # this simply creates a grid of toggle buttons on the table
49         # to demonstrate the scrolled window.
50         for i in range(10):
51             for j in range(10):
52                 buffer = "button (%d,%d)" % (i, j)
53                 button = gtk.ToggleButton(buffer)
54                 table.attach(button, i, i+1, j, j+1)
55                 button.show()
56
57         # Add a "close" button to the bottom of the dialog
58         button = gtk.Button("close")
59         button.connect_object("clicked", self.destroy, window)
60
61         # this makes it so the button is the default.
62         button.set_flags(gtk.CAN_DEFAULT)
63         window.action_area.pack_start( button, True, True, 0)
64
65         # This grabs this button to be the default button. Simply hitting
66         # the "Enter" key will cause this button to activate.
67         button.grab_default()
68         button.show()
69         window.show()
70
71 def main():
72     gtk.main()
73     return 0
74
75 if __name__ == "__main__":
76     ScrolledWindowExample()
77     main()
```

Try resizing the window. You'll notice how the scrollbars react. You may also wish to use the set_size_request() method to set the default size of the window or other widgets.

10.10 Button Boxes

`ButtonBoxes` are a convenient way to quickly layout a group of buttons. They come in both horizontal and vertical flavors. You create a new `ButtonBox` with one of the following calls, which create a

horizontal or vertical box, respectively:

```
hbutton_box = gtk.HButtonBox()

vbutton_box = gtk.VButtonBox()
```

The only methods pertaining to button boxes effect how the buttons are laid out.

The layout of the buttons within the box is set using:

```
button_box.set_layout(layout_style)
```

The *layout_style* argument can take one of the following values:

```
BUTTONBOX_DEFAULT_STYLE
BUTTONBOX_SPREAD
BUTTONBOX_EDGE
BUTTONBOX_START
BUTTONBOX_END
```

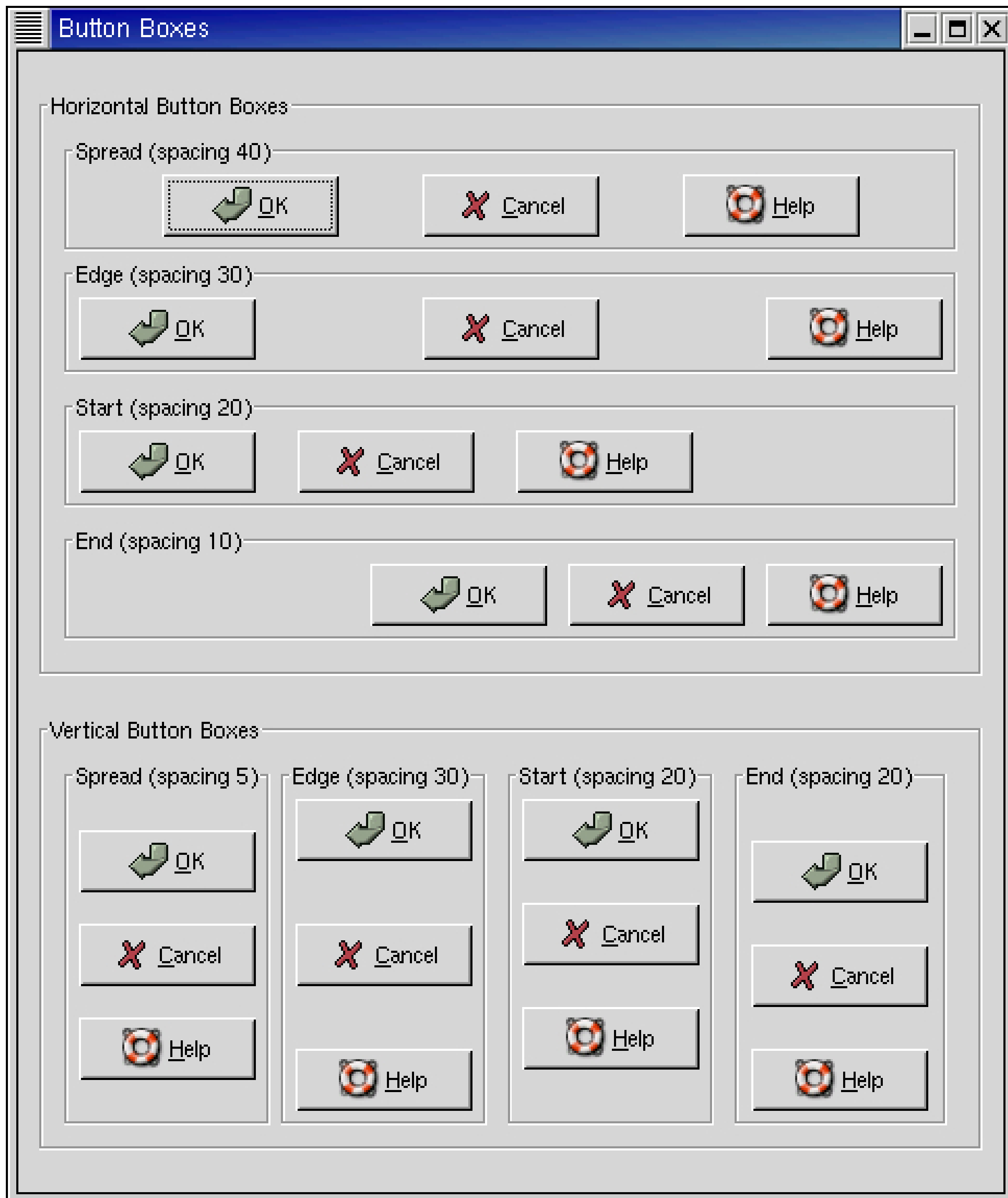
The current *layout_style* setting can be retrieved using:

```
layout_style = button_box.get_layout()
```

Buttons are added to a `ButtonBox` using the usual `Container` method:

```
button_box.add(widget)
```

The buttonbox.py example program illustrates all the different layout settings for `ButtonBoxes`. The resulting display is:



The source code for the buttonbox.py program is:

```
 1 #!/usr/bin/env python
 2
 3 # example buttonbox.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ButtonBoxExample:
10     # Create a Button Box with the specified parameters
11     def create_bbox(self, horizontal, title, spacing, layout):
12         frame = gtk.Frame(title)
13
```

```
14          if horizontal:
15              bbox = gtk.HButtonBox()
16          else:
17              bbox = gtk.VButtonBox()
18
19          bbox.set_border_width(5)
20          frame.add(bbox)
21
22          # Set the appearance of the Button Box
23          bbox.set_layout(layout)
24          bbox.set_spacing(spacing)
25
26          button = gtk.Button(stock=gtk.STOCK_OK)
27          bbox.add(button)
28
29          button = gtk.Button(stock=gtk.STOCK_CANCEL)
30          bbox.add(button)
31
32          button = gtk.Button(stock=gtk.STOCK_HELP)
33          bbox.add(button)
34
35          return frame
36
37      def __init__(self):
38          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
39          window.set_title("Button Boxes")
40
41          window.connect("destroy", lambda x: gtk.main_quit())
42
43          window.set_border_width(10)
44
45          main_vbox = gtk.VBox(False, 0)
46          window.add(main_vbox)
47
48          frame_horz = gtk.Frame("Horizontal Button Boxes")
49          main_vbox.pack_start(frame_horz, True, True, 10)
50
51          vbox = gtk.VBox(False, 0)
52          vbox.set_border_width(10)
53          frame_horz.add(vbox)
54
55          vbox.pack_start(self.create_bbox(True, "Spread (spacing 40)",
56                                           40, gtk.BUTTONBOX_SPREAD),
57                          True, True, 0)
58
59          vbox.pack_start(self.create_bbox(True, "Edge (spacing 30)",
60                                           30, gtk.BUTTONBOX_EDGE),
61                          True, True, 5)
62
63          vbox.pack_start(self.create_bbox(True, "Start (spacing 20)",
64                                           20, gtk.BUTTONBOX_START),
65                          True, True, 5)
66
67          vbox.pack_start(self.create_bbox(True, "End (spacing 10)",
68                                           10, gtk.BUTTONBOX_END),
69                          True, True, 5)
70
71          frame_vert = gtk.Frame("Vertical Button Boxes")
72          main_vbox.pack_start(frame_vert, True, True, 10)
73
74          hbox = gtk.HBox(False, 0)
75          hbox.set_border_width(10)
76          frame_vert.add(hbox)
77
```

```
 78         hbox.pack_start(self.create_bbox(False, "Spread (spacing 5)",
 79                                          5, gtk.BUTTONBOX_SPREAD),
 80                         True, True, 0)
 81
 82         hbox.pack_start(self.create_bbox(False, "Edge (spacing 30)",
 83                                          30, gtk.BUTTONBOX_EDGE),
 84                         True, True, 5)
 85
 86         hbox.pack_start(self.create_bbox(False, "Start (spacing 20)",
 87                                          20, gtk.BUTTONBOX_START),
 88                         True, True, 5)
 89
 90         hbox.pack_start(self.create_bbox(False, "End (spacing 20)",
 91                                          20, gtk.BUTTONBOX_END),
 92                         True, True, 5)
 93
 94         window.show_all()
 95
 96 def main():
 97     # Enter the event loop
 98     gtk.main()
 99     return 0
100
101 if __name__ == "__main__":
102     ButtonBoxExample()
103     main()
```

10.11 Toolbar

`Toolbars` are usually used to group some number of widgets in order to simplify customization of their look and layout. Typically a toolbar consists of buttons with icons, labels and tooltips, but any other widget can also be put inside a toolbar. Finally, items can be arranged horizontally or vertically and buttons can be displayed with icons, labels, or both.

Creating a toolbar is (as one may already suspect) done with the following function:

```
toolbar = gtk.Toolbar()
```

After creating a toolbar one can append, prepend and insert items (that means simple text strings) or elements (that means any widget types) into the toolbar. To describe an item we need a label text, a tooltip text, a private tooltip text, an icon for the button and a callback for it. For example, to append or prepend an item you may use the following methods:

```
toolbar.append_item(text, tooltip_text, tooltip_private_text, icon, callback, ↩
    user_data=None)

toolbar.prepend_item(text, tooltip_text, tooltip_private_text, icon, callback, ↩
    user_data)
```

If you want to use the insert_item() method, the only additional parameter which must be specified is the position in which the item should be inserted, thus:

```
toolbar.insert_item(text, tooltip_text, tooltip_private_text, icon, callback,
                    user_data, position)
```

To simplify adding spaces between toolbar items, you may use the following methods:

```
toolbar.append_space()

toolbar.prepend_space()

toolbar.insert_space(position)
```

If it's required, the orientation of a toolbar, its style and whether tooltips are available can be changed "on the fly" using the following methods:

```
toolbar.set_orientation(orientation)

toolbar.set_style(style)

toolbar.set_tooltips(enable)
```

Where *orientation* is one of ORIENTATION_HORIZONTAL or ORIENTATION_VERTICAL. The *style* is used to set appearance of the toolbar items by using one of TOOLBAR_ICONS, TOOLBAR_TEXT, or TOOLBAR_BOTH. The *enable* argument is either TRUE or FALSE.

To show some other things that can be done with a toolbar, let's take the toolbar.py example program (we'll interrupt the listing with some additional explanations):

```
 1  #!/usr/bin/env python
 2
 3  # example toolbar.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class ToolbarExample:
10      # This method is connected to the Close button or
11      # closing the window from the WM
12      def delete_event(self, widget, event=None):
13          gtk.main_quit()
14          return False
15
```

The above beginning seems should be familiar to you if it's not your first PyGTK program. There is one additional thing though, we import a nice XPM picture (gtk.xpm)to serve as an icon for all of the buttons. Line 10 starts the ToolbarExample class and lines 12-14 define the callback method which will terminate the program.

```
16      # that's easy... when one of the buttons is toggled, we just
17      # check which one is active and set the style of the toolbar
18      # accordingly
19      def radio_event(self, widget, toolbar):
20          if self.text_button.get_active():
21              toolbar.set_style(gtk.TOOLBAR_TEXT)
22          elif self.icon_button.get_active():
23              toolbar.set_style(gtk.TOOLBAR_ICONS)
24          elif self.both_button.get_active():
25              toolbar.set_style(gtk.TOOLBAR_BOTH)
26
27      # even easier, just check given toggle button and enable/disable
28      # tooltips
29      def toggle_event(self, widget, toolbar):
30          toolbar.set_tooltips(widget.get_active())
31
```

Lines 19-30 are two callback methods that will be called when one of the buttons on a toolbar is pressed. You should already be familiar with things like this if you've already used toggle buttons (and radio buttons).

```
32      def __init__(self):
33          # Here is our main window (a dialog) and a handle for the ↩
    handlebox
34          # Ok, we need a toolbar, an icon with a mask (one for all of
35          # the buttons) and an icon widget to put this icon in (but
36          # we'll create a separate widget for each button)
37          # create a new window with a given title, and nice size
38          dialog = gtk.Dialog()
39          dialog.set_title("GTKToolbar Tutorial")
40          dialog.set_size_request(450, 250)
```

```
41          dialog.set_resizable(True)
42
43          # typically we quit if someone tries to close us
44          dialog.connect("delete_event", self.delete_event)
45
46          # to make it nice we'll put the toolbar into the handle box,
47          # so that it can be detached from the main window
48          handlebox = gtk.HandleBox()
49          dialog.vbox.pack_start(handlebox, False, False, 5)
50
```

The above should be similar to any other PyGTK application. Just initialization of a `ToolbarExample` object instance creating the window, etc. There is only one thing that probably needs some explanation: a handle box. A handle box is just another box that can be used to pack widgets in to. The difference between it and typical boxes is that it can be detached from a parent window (or, in fact, the handle box remains in the parent, but it is reduced to a very small rectangle, while all of its contents are reparented to a new freely floating window). It is usually nice to have a detachable toolbar, so these two widgets occur together quite often.

```
51          # toolbar will be horizontal, with both icons and text, and
52          # with 5pxl spaces between items and finally,
53          # we'll also put it into our handlebox
54          toolbar = gtk.Toolbar()
55          toolbar.set_orientation(gtk.ORIENTATION_HORIZONTAL)
56          toolbar.set_style(gtk.TOOLBAR_BOTH)
57          toolbar.set_border_width(5)
58          handlebox.add(toolbar)
59
```

Well, what we do above is just a straightforward initialization of the toolbar widget.

```
60          # our first item is <close> button
61          iconw = gtk.Image() # icon widget
62          iconw.set_from_file("gtk.xpm")
63          close_button = toolbar.append_item(
64              "Close",           # button label
65              "Closes this app", # this button's tooltip
66              "Private",         # tooltip private info
67              iconw,             # icon widget
68              self.delete_event) # a signal
69          toolbar.append_space() # space after item
```

In the above code you see the simplest case: adding a button to toolbar. Just before appending a new item, we have to construct an image widget to serve as an icon for this item; this step will have to be repeated for each new item. Just after the item we also add a space, so the following items will not touch each other. As you see the append_item() method returns a reference to our newly created button widget, so that we can work with it in the normal way.

```
71          # now, let's make our radio buttons group...
72          iconw = gtk.Image() # icon widget
73          iconw.set_from_file("gtk.xpm")
74          icon_button = toolbar.append_element(
75              gtk.TOOLBAR_CHILD_RADIOBUTTON, # type of element
76              None,                          # widget
77              "Icon",                        # label
78              "Only icons in toolbar",       # tooltip
79              "Private",                     # tooltip private string
80              iconw,                         # icon
81              self.radio_event,              # signal
82              toolbar)                       # data for signal
83          toolbar.append_space()
84          self.icon_button = icon_button
85
```

Here we begin creating a radio buttons group. To do this we use the append_element() method. In fact, using this method one can also add simple items or even spaces (*type* = `gtk.TOOLBAR_CHILD_SPACE` or `gtk.TOOLBAR_CHILD_BUTTON`). In the above case we start creating a radio group. In creating other radio buttons for this group a reference to the previous button in the group is required, so that a list of buttons can be easily constructed (see Section 6.4 earlier in this tutorial). We also save a reference to the button in the `ToolbarExample` instance for later access.

```
 86            # following radio buttons refer to previous ones
 87            iconw = gtk.Image() # icon widget
 88            iconw.set_from_file("gtk.xpm")
 89            text_button = toolbar.append_element(
 90                gtk.TOOLBAR_CHILD_RADIOBUTTON,
 91                icon_button,
 92                "Text",
 93                "Only texts in toolbar",
 94                "Private",
 95                iconw,
 96                self.radio_event,
 97                toolbar)
 98            toolbar.append_space()
 99            self.text_button = text_button
100
101            iconw = gtk.Image() # icon widget
102            iconw.set_from_file("gtk.xpm")
103            both_button = toolbar.append_element(
104                gtk.TOOLBAR_CHILD_RADIOBUTTON,
105                text_button,
106                "Both",
107                "Icons and text in toolbar",
108                "Private",
109                iconw,
110                self.radio_event,
111                toolbar)
112            toolbar.append_space()
113            self.both_button = both_button
114            both_button.set_active(True)
115
```

We create the other radiobuttons the same way except we pass one of the created radio group buttons to the append_element() method to specify the radio group.

In the end we have to set the state of one of the buttons manually (otherwise they all stay in active state, preventing us from switching between them).

```
116            # here we have just a simple toggle button
117            iconw = gtk.Image() # icon widget
118            iconw.set_from_file("gtk.xpm")
119            tooltips_button = toolbar.append_element(
120                gtk.TOOLBAR_CHILD_TOGGLEBUTTON,
121                None,
122                "Tooltips",
123                "Toolbar with or without tips",
124                "Private",
125                iconw,
126                self.toggle_event,
127                toolbar)
128            toolbar.append_space()
129            tooltips_button.set_active(True)
130
```

A toggle button can be created in the obvious way (if one knows how to create radio buttons already).

```
131            # to pack a widget into toolbar, we only have to
132            # create it and append it with an appropriate tooltip
133            entry = gtk.Entry()
134            toolbar.append_widget(entry,  "This is just an entry", "Private")
```

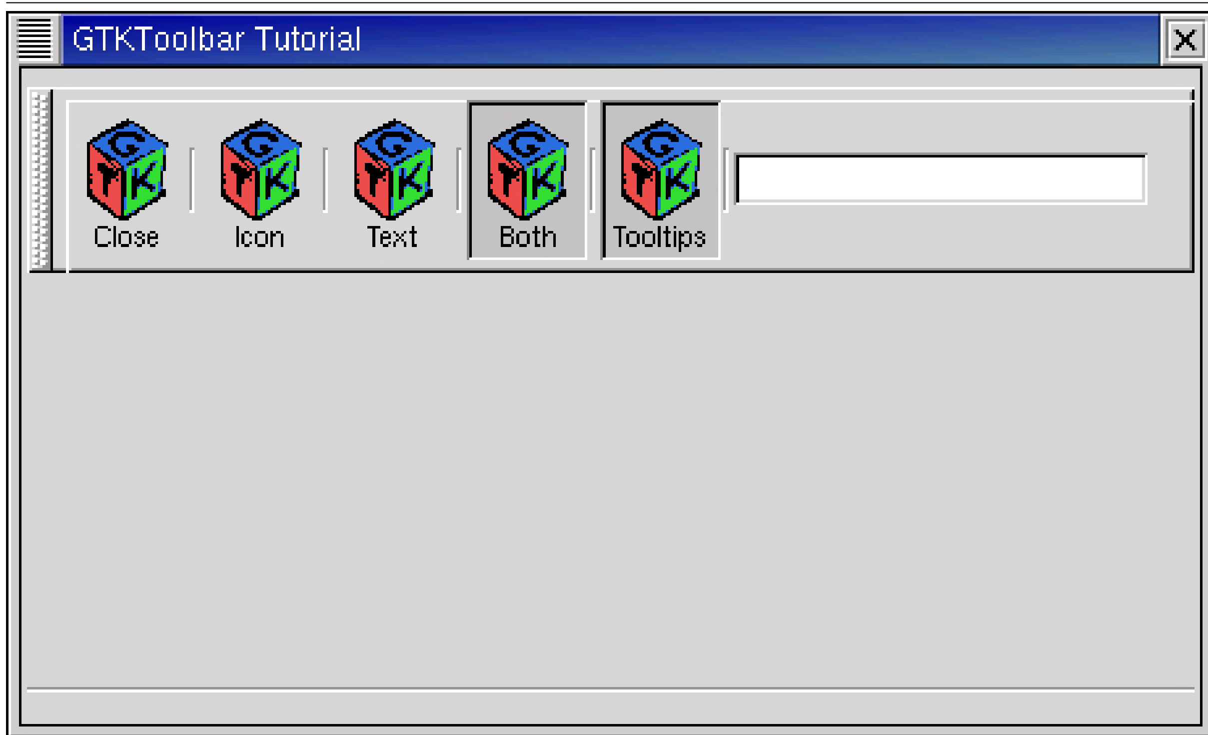
```
135
136             # well, it isn't created within the toolbar, so we must still ↩
    show it
137             entry.show()
138
```

As you see, adding any kind of widget to a toolbar is simple. The one thing you have to remember is that this widget must be shown manually (contrary to items which will be shown together with the toolbar).

```
139             # that's it ! let's show everything.
140             toolbar.show()
141             handlebox.show()
142             dialog.show()
143
144     def main():
145         # rest in gtk_main and wait for the fun to begin!
146         gtk.main()
147         return 0
148
149     if __name__ == "__main__":
150         ToolbarExample()
151         main()
```

Line 142 ends the `ToolbarExample` class definition. Lines 144-147 define the `main`() function which just calls the `gtk.main`() function to start the event processing loop. Lines 149-151 arrange to create a `ToolbarExample` instance and then enter the event processing loop. So, here we are at the end of toolbar tutorial. Of course, to appreciate it in full you need also this nice XPM icon, gtk.xpm. Figure 10.8 illustrates the resulting display:

Figure 10.8 Toolbar Example



10.12 Notebooks

The `NoteBook` Widget is a collection of "pages" that overlap each other; each page contains different information with only one page visible at a time. This widget has become more common lately in GUI

programming, and it is a good way to show blocks of similar information that warrant separation in their display.

The first function call you will need to know, as you can probably guess by now, is used to create a new notebook widget.

```
notebook = gtk.Notebook()
```

Once the notebook has been created, there are a number of methods that operate on the notebook widget. Let's look at them individually.

The first one we will look at is how to position the page indicators. These page indicators or "tabs" as they are referred to, can be positioned in four ways: top, bottom, left, or right.

```
notebook.set_tab_pos(pos)
```

pos will be one of the following, which are pretty self explanatory:

```
POS_LEFT
POS_RIGHT
POS_TOP
POS_BOTTOM
```

`POS_TOP` is the default.

Next we will look at how to add pages to the notebook. There are three ways to add pages to a `NoteBook`. Let's look at the first two together as they are quite similar.

```
notebook.append_page(child, tab_label)

notebook.prepend_page(child, tab_label)
```

These methods add pages to the notebook by inserting them from the back of the notebook (append), or the front of the notebook (prepend). *child* is the widget that is placed within the notebook page, and *tab_label* is the label for the page being added. The *child* widget must be created separately, and is typically a set of options setup within one of the other container widgets, such as a table.

The final method for adding a page to the notebook contains all of the properties of the previous two, but it allows you to specify what position you want the page to be in the notebook.

```
notebook.insert_page(child, tab_label, position)
```

The parameters are the same as append() and prepend() except it contains an extra parameter, *position*. This parameter is used to specify what place this page will be inserted into; the first page having position zero.

Now that we know how to add a page, lets see how we can remove a page from the notebook.

```
notebook.remove_page(page_num)
```

This method takes the page specified by *page_num* and removes it from the widget pointed to by *notebook*.

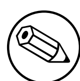
To find out what the current page is in a notebook use the method:

```
page = notebook.get_current_page()
```

These next two methods are simple calls to move the notebook page forward or backward. Simply provide the respective method call with the notebook widget you wish to operate on.

```
notebook.next_page()

notebook.prev_page()
```

NOTE



When the *notebook* is currently on the last page, and next_page() is called, nothing happens. Likewise, if the *notebook* is on the first page, and prev_page() is called, nothing happens.

This next method sets the "active" page. If you wish the notebook to be opened to page 5 for example, you would use this method. Without using this method, the notebook defaults to displaying the first page.

```
notebook.set_current_page(page_num)
```

The next two methods add or remove the notebook page tabs and the notebook border respectively.

```
notebook.set_show_tabs(show_tabs)

notebook.set_show_border(show_border)
```

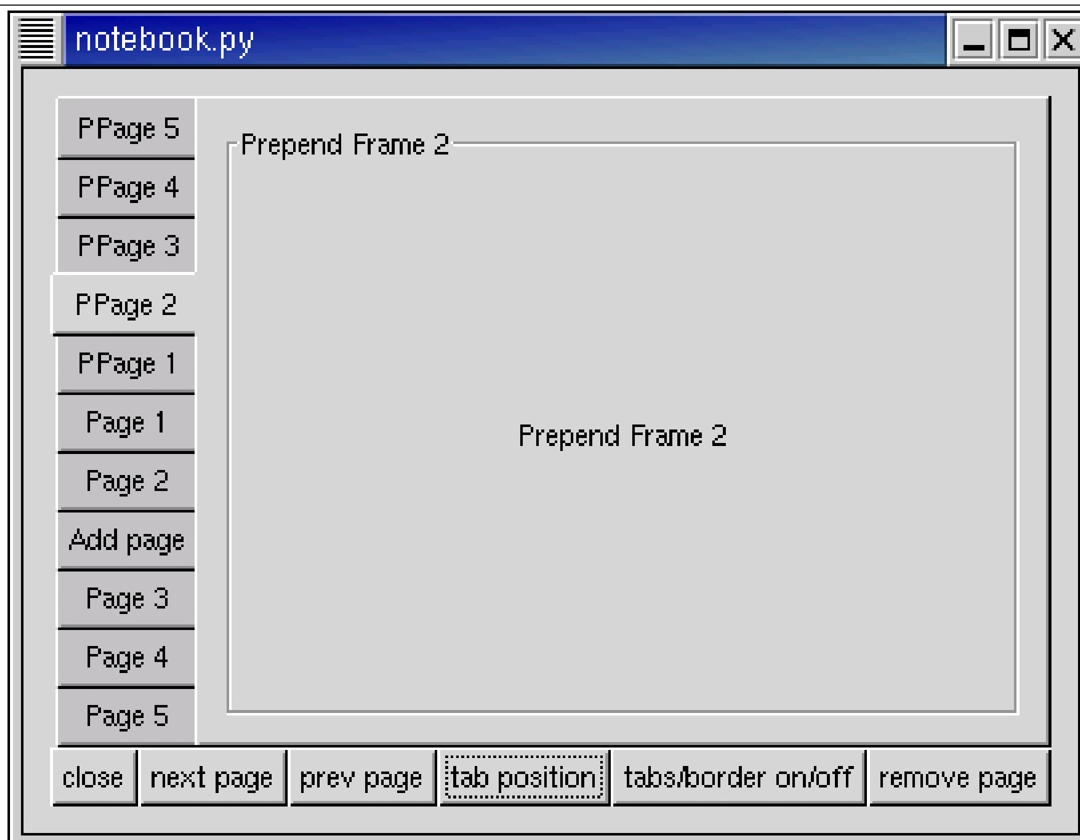
The next method is useful when the you have a large number of pages, and the tabs don't fit on the page. It allows the tabs to be scrolled through using two arrow buttons.

```
notebook.set_scrollable(scrollable)
```

`show_tabs`, `show_border` and `scrollable` can be either `TRUE` or `FALSE`.

Now let's look at an example. The notebook.py program creates a window with a notebook and six buttons. The notebook contains 11 pages, added in three different ways, appended, inserted, and prepended. The buttons allow you rotate the tab positions, add or remove the tabs and border, remove a page, change pages in both a forward and backward manner, and exit the program. Figure 10.9 illustrates the program display:

Figure 10.9 Notebook Example



The source code for notebook.py is:

```
1 #!/usr/bin/env python
2
3 # example notebook.py
4
5 import pygtk
6 pygtk.require('2.0')
7 import gtk
```

```
 8
 9 class NotebookExample:
10     # This method rotates the position of the tabs
11     def rotate_book(self, button, notebook):
12         notebook.set_tab_pos((notebook.get_tab_pos()+1) %4)
13
14     # Add/Remove the page tabs and the borders
15     def tabsborder_book(self, button, notebook):
16         tval = False
17         bval = False
18         if self.show_tabs == False:
19       tval = True
20         if self.show_border == False:
21       bval = True
22
23         notebook.set_show_tabs(tval)
24         self.show_tabs = tval
25         notebook.set_show_border(bval)
26         self.show_border = bval
27
28     # Remove a page from the notebook
29     def remove_book(self, button, notebook):
30         page = notebook.get_current_page()
31         notebook.remove_page(page)
32         # Need to refresh the widget --
33         # This forces the widget to redraw itself.
34         notebook.queue_draw_area(0,0,-1,-1)
35
36     def delete(self, widget, event=None):
37         gtk.main_quit()
38         return False
39
40     def __init__(self):
41         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
42         window.connect("delete_event", self.delete)
43         window.set_border_width(10)
44
45         table = gtk.Table(3,6,False)
46         window.add(table)
47
48         # Create a new notebook, place the position of the tabs
49         notebook = gtk.Notebook()
50         notebook.set_tab_pos(gtk.POS_TOP)
51         table.attach(notebook, 0,6,0,1)
52         notebook.show()
53         self.show_tabs = True
54         self.show_border = True
55
56         # Let's append a bunch of pages to the notebook
57         for i in range(5):
58             bufferf = "Append Frame %d" % (i+1)
59             bufferl = "Page %d" % (i+1)
60
61             frame = gtk.Frame(bufferf)
62             frame.set_border_width(10)
63             frame.set_size_request(100, 75)
64             frame.show()
65
66             label = gtk.Label(bufferf)
67             frame.add(label)
68             label.show()
69
70             label = gtk.Label(bufferl)
71             notebook.append_page(frame, label)
```

```
 72
 73         # Now let’s add a page to a specific spot
 74         checkbutton = gtk.CheckButton("Check me please!")
 75         checkbutton.set_size_request(100, 75)
 76         checkbutton.show ()
 77
 78         label = gtk.Label("Add page")
 79         notebook.insert_page(checkbutton, label, 2)
 80
 81         # Now finally let’s prepend pages to the notebook
 82         for i in range(5):
 83             bufferf = "Prepend Frame %d" % (i+1)
 84             bufferl = "PPage %d" % (i+1)
 85
 86             frame = gtk.Frame(bufferf)
 87             frame.set_border_width(10)
 88             frame.set_size_request(100, 75)
 89             frame.show()
 90
 91             label = gtk.Label(bufferf)
 92             frame.add(label)
 93             label.show()
 94
 95             label = gtk.Label(bufferl)
 96             notebook.prepend_page(frame, label)
 97
 98         # Set what page to start at (page 4)
 99         notebook.set_current_page(3)
100
101         # Create a bunch of buttons
102         button = gtk.Button("close")
103         button.connect("clicked", self.delete)
104         table.attach(button, 0,1,1,2)
105         button.show()
106
107         button = gtk.Button("next page")
108         button.connect("clicked", lambda w: notebook.next_page())
109         table.attach(button, 1,2,1,2)
110         button.show()
111
112         button = gtk.Button("prev page")
113         button.connect("clicked", lambda w: notebook.prev_page())
114         table.attach(button, 2,3,1,2)
115         button.show()
116
117         button = gtk.Button("tab position")
118         button.connect("clicked", self.rotate_book, notebook)
119         table.attach(button, 3,4,1,2)
120         button.show()
121
122         button = gtk.Button("tabs/border on/off")
123         button.connect("clicked", self.tabsborder_book, notebook)
124         table.attach(button, 4,5,1,2)
125         button.show()
126
127         button = gtk.Button("remove page")
128         button.connect("clicked", self.remove_book, notebook)
129         table.attach(button, 5,6,1,2)
130         button.show()
131
132         table.show()
133         window.show()
134
135 def main():
```

```
136     gtk.main()
137     return 0
138
139 if __name__ == "__main__":
140     NotebookExample()
141     main()
```

I hope this helps you on your way with creating notebooks for your PyGTK applications.

10.13 Plugs and Sockets

`Plugs` and `Sockets` cooperate to embed the user interface from one process into another process. This can also be accomplished using Bonobo.

10.13.1 Plugs

A `Plug` encapsulates a user interface provided by one application so that it can be embedded in another application's user interface. The "embedded" signal alerts the plug application that the plug has been embedded in the other application's user interface.

A `Plug` is created using the following function:

```
plug = gtk.Plug(socket_id)
```

which creates a new `Plug` and embeds it in the `Socket` identified by *socket_id*. If *socket_id* is 0L, the plug is left "unplugged" and can later be plugged into a `Socket` using the `Socket` add_id() method.

The `Plug` method:

```
id = plug.get_id()
```

returns the window ID of a `Plug`, that can be used to embed it inside a `Socket` using the `Socket` add_id() method.

The plug.py example program illustrates the use of a Plug:

```
 1 #!/usr/bin/python
 2
 3 import pygtk
 4 pygtk.require('2.0')
 5 import gtk,sys
 6
 7 Wid = 0L
 8 if len(sys.argv) == 2:
 9     Wid = long(sys.argv[1])
10
11 plug = gtk.Plug(Wid)
12 print "Plug ID=", plug.get_id()
13
14 def embed_event(widget):
15     print "I (",widget,") have just been embedded!"
16
17 plug.connect("embedded", embed_event)
18
19 entry = gtk.Entry()
20 entry.set_text("hello")
21 def entry_point(widget):
22     print "You've changed my text to '%s'" % widget.get_text()
23
24 entry.connect("changed", entry_point)
25 plug.connect("destroy", gtk.mainquit)
26
27 plug.add(entry)
28 plug.show_all()
29
```

```
30
31 gtk.mainloop()
```

The program is invoked like:

```
plug.py [windowID]
```

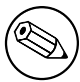
where *windowID* is the ID of a Socket to connect the Plug to.

10.13.2 Sockets

A Socket provides the widget to embed a Plug widget from another application into your GUI transparently. An application creates a Socket widget and, passes that widget's window ID to another application, which then creates a Plug using that window ID as a parameter. Any widgets contained in the Plug appear inside the first application's window.

The Socket window ID is obtained by using the Socket method get_id(). Before using this method, the Socket must be realized, and added to its parent.

NOTE



If you pass the window ID of the Socket to another process that will create a Plug in the Socket, you must make sure that the Socket widget is not destroyed until that Plug is created.

When GTK+ is notified that the embedded window has been destroyed, then it will destroy the Socket as well. You should always, therefore, be prepared for your sockets to be destroyed at any time when the main event loop is running. Destroying a Socket will cause an embedded Plug to be destroyed as well.

The communication between a Socket and a Plug follows the XEmbed protocol. This protocol has also been implemented in other toolkits, e.g. Qt, allowing the same level of integration when embedding a Qt widget in GTK or vice versa.

Create a new empty Socket:

```
socket = gtk.Socket()
```

The Socket must be contained in a toplevel window before you invoke the add_id() method:

```
socket.add_id(window_id)
```

which adds an XEMBED client, such as a Plug, to the Socket. The client may be in the same process or in a different process.

To embed a Plug in a Socket, you can either create the Plug with:

```
plug = gtk.Plug(0L)
```

and then pass the number returned by the Plug get_id() method to the Socket add_id() method:

```
socket.add_id(plug)
```

or you can invoke the Socket get_id() method:

```
window_id = socket.get_id()
```

to get the window ID for the socket, and then create the plug with:

```
plug = gtk.Plug(window_id)
```

The Socket must have already be added into a toplevel window before you can make this call.

The socket.py example program illustrates the use of a Socket:

```
1 #!/usr/bin/python
2
3 import string
4
```

```
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk,sys
 8
 9 window = gtk.Window()
10 window.show()
11
12 socket = gtk.Socket()
13 socket.show()
14 window.add(socket)
15
16 print "Socket ID=", socket.get_id()
17 window.connect("destroy", gtk.mainquit)
18
19 def plugged_event(widget):
20     print "I (",widget,") have just had a plug inserted!"
21
22 socket.connect("plug-added", plugged_event)
23
24 if len(sys.argv) == 2:
25     socket.add_id(long(sys.argv[1]))
26
27 gtk.mainloop()
```

To run the example you can either run plug.py first:

```
$ python plug.py
Plug ID= 20971522
```

and copy the output ID to the first arg of socket.py:

```
$ python socket.py 20971522
Socket ID= 48234523
I ( <gtk.Plug object (GtkPlug) at 0x3008dd78> ) have just been embedded!
I ( <gtk.Socket object (GtkSocket) at 0x3008ddf0> ) have just had a plug ↩
    inserted!
```

Or you can run socket.py:

```
$ python socket.py
Socket ID= 20971547
```

and then run plug.py, copying across the window ID:

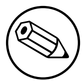
```
$ python plug.py
20971547
I ( <gtk.Socket object (GtkSocket) at 0x3008ddf0> ) have just had a plug ↩
    inserted!
Plug ID= 48234498
```

Chapter 11

Menu Widget

There are two ways to create menus: there's the easy way, and there's the hard way. Both have their uses, but you can usually use the ItemFactory (the easy way). The "hard" way is to create all the menus using the calls directly. The easy way is to use the gtk.ItemFactory calls. This is much simpler, but there are advantages and disadvantages to each approach.

NOTE



In PyGTK 2.4 ItemFactory is deprecated - use the UIManager instead.

The ItemFactory is much easier to use, and to add new menus to, although writing a few wrapper functions to create menus using the manual method could go a long way towards usability. With the Itemfactory, it is not possible to add images or the character '/' to the menus.

11.1 Manual Menu Creation

In the true tradition of teaching, we'll show you the hard way first. :)

There are three widgets that go into making a menubar and submenus:

- a menu item, which is what the user wants to select, e.g., "Save"
- a menu, which acts as a container for the menu items, and
- a menubar, which is a container for each of the individual menus.

This is slightly complicated by the fact that menu item widgets are used for two different things. They are both the widgets that are packed into the menu, and the widget that is packed into the menubar, which, when selected, activates the menu.

Let's look at the functions that are used to create menus and menubars. This first function is used to create a new menubar:

```
menu_bar = gtk.MenuBar()
```

This rather self explanatory function creates a new menubar. You use the gtk.Container add() method to pack this into a window, or the gtk.Box pack methods to pack it into a box - the same as buttons.

```
menu = gtk.Menu()
```

This function returns a reference to a new menu; it is never actually shown (with the show() method), it is just a container for the menu items. I hope this will become more clear when you look at the example below.

The next function is used to create menu items that are packed into the menu (and menubar):

```
menu_item = gtk.MenuItem(label=None)
```

The *label*, if any, will be parsed for mnemonic characters. This call is used to create the menu items that are to be displayed. Remember to differentiate between a "menu" as created with gtk.Menu() and a "menu item" as created by the gtk.MenuItem() functions. The menu item will be an actual button with an associated action, whereas a menu will be a container holding menu items.

Once you've created a menu item you have to put it into a menu. This is done using the append() method. In order to capture when the item is selected by the user, we need to connect to the "activate" signal in the usual way. So, if we wanted to create a standard File menu, with the options Open, Save, and Quit, the code would look something like:

```
file_menu = gtk.Menu()    # Don't need to show menus

# Create the menu items
open_item = gtk.MenuItem("Open")
save_item = gtk.MenuItem("Save")
quit_item = gtk.MenuItem("Quit")

# Add them to the menu
file_menu.append(open_item)
file_menu.append(save_item)
file_menu.append(quit_item)

# Attach the callback functions to the activate signal
open_item.connect_object("activate", menuitem_response, "file.open")
save_item.connect_object("activate", menuitem_response, "file.save")

# We can attach the Quit menu item to our exit function
quit_item.connect_object ("activate", destroy, "file.quit")

# We do need to show menu items
open_item.show()
save_item.show()
quit_item.show()
```

At this point we have our menu. Now we need to create a menubar and a menu item for the File entry, to which we add our menu. The code looks like this:

```
menu_bar = gtk.MenuBar()
window.add(menu_bar)
menu_bar.show()

file_item = gtk.MenuItem("File")
file_item.show()
```

Now we need to associate the menu with *file_item*. This is done with the method:

```
menu_item.set_submenu(submenu)
```

So, our example would continue with:

```
file_item.set_submenu(file_menu)
```

All that is left to do is to add the menu to the menubar, which is accomplished using the method:

```
menu_bar.append(child)
```

which in our case looks like this:

```
menu_bar.append(file_item)
```

If we wanted the menu right justified on the menubar, such as help menus often are, we can use the following method (again on *file_item* in the current example) before attaching it to the menubar.

```
menu_item.set_right_justified(right_justified)
```

Here is a summary of the steps needed to create a menu bar with menus attached:

- Create a new menu using gtk.Menu()

- Use multiple calls to `gtk.MenuItem()` for each item you wish to have on your menu. And use the append() method to put each of these new items on to the menu.
- Create a menu item using `gtk.MenuItem()`. This will be the root of the menu, the text appearing here will be on the menubar itself.
- Use the set_submenu() method to attach the menu to the root menu item (the one created in the above step).
- Create a new menubar using `gtk.MenuBar()`. This step only needs to be done once when creating a series of menus on one menu bar.
- Use the append() method to put the root menu onto the menubar.

Creating a popup menu is nearly the same. The difference is that the menu is not posted "automatically" by a menubar, but explicitly by calling the popup() method from a button-press event, for example. Take these steps:

- Create an event handling callback. It needs to have the format:

```
def handler(widget, event):
```

- and it will use the event to find out where to pop up the menu.
- In the event handler, if the event is a mouse button press, treat event as a button event (which it is) and use it as shown in the sample code to pass information to the popup() method.
- Bind that event handler to a widget with:

```
widget.connect_object("event", handler, menu)
```

- where widget is the widget you are binding to, handler is the handling function, and menu is a menu created with `gtk.Menu()`. This can be a menu which is also posted by a menu bar, as shown in the sample code.

11.2 Manual Menu Example

That should about do it. Let's take a look at the menu.py example program to help clarify the concepts. Figure 11.1 illustrates the program display:

Figure 11.1 Menu Example

The menu.py program source code is:

```
1 #!/usr/bin/env python
2
3 # example menu.py
4
5 import pygtk
6 pygtk.require('2.0')
```

```
 7 import gtk
 8
 9 class MenuExample:
10     def __init__(self):
11         # create a new window
12         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
13         window.set_size_request(200, 100)
14         window.set_title("GTK Menu Test")
15         window.connect("delete_event", lambda w,e: gtk.main_quit())
16
17         # Init the menu-widget, and remember -- never
18         # show() the menu widget!!
19         # This is the menu that holds the menu items, the one that
20         # will pop up when you click on the "Root Menu" in the app
21         menu = gtk.Menu()
22
23         # Next we make a little loop that makes three menu-entries for
24         # "test-menu".  Notice the call to gtk_menu_append.  Here we are
25         # adding a list of menu items to our menu.  Normally, we'd also
26         # catch the "clicked" signal on each of the menu items and setup a
27         # callback for it, but it's omitted here to save space.
28         for i in range(3):
29             # Copy the names to the buf.
30             buf = "Test-undermenu - %d" % i
31
32             # Create a new menu-item with a name...
33             menu_items = gtk.MenuItem(buf)
34
35             # ...and add it to the menu.
36             menu.append(menu_items)
37
38 	    # Do something interesting when the menuitem is selected
39 	    menu_items.connect("activate", self.menuitem_response, buf)
40
41             # Show the widget
42             menu_items.show()
43
44         # This is the root menu, and will be the label
45         # displayed on the menu bar.  There won't be a signal handler  ←
    attached,
46         # as it only pops up the rest of the menu when pressed.
47         root_menu = gtk.MenuItem("Root Menu")
48
49         root_menu.show()
50
51         # Now we specify that we want our newly created "menu" to be the
52         # menu for the "root menu"
53         root_menu.set_submenu(menu)
54
55         # A vbox to put a menu and a button in:
56         vbox = gtk.VBox(False, 0)
57         window.add(vbox)
58         vbox.show()
59
60         # Create a menu-bar to hold the menus and add it to our main window
61         menu_bar = gtk.MenuBar()
62         vbox.pack_start(menu_bar, False, False, 2)
63         menu_bar.show()
64
65         # Create a button to which to attach menu as a popup
66         button = gtk.Button("press me")
67         button.connect_object("event", self.button_press, menu)
68         vbox.pack_end(button, True, True, 2)
69         button.show()
```

```
 70
 71         # And finally we append the menu-item to the menu-bar -- this is ↩
    the
 72         # "root" menu-item I have been raving about =)
 73         menu_bar.append (root_menu)
 74
 75         # always display the window as the last step so it all splashes on
 76         # the screen at once.
 77         window.show()
 78
 79     # Respond to a button-press by posting a menu passed in as widget.
 80     #
 81     # Note that the "widget" argument is the menu being posted, NOT
 82     # the button that was pressed.
 83     def button_press(self, widget, event):
 84         if event.type == gtk.gdk.BUTTON_PRESS:
 85             widget.popup(None, None, None, event.button, event.time)
 86             # Tell calling code that we have handled this event the buck
 87             # stops here.
 88             return True
 89         # Tell calling code that we have not handled this event pass it on.
 90         return False
 91
 92     # Print a string when a menu item is selected
 93     def menuitem_response(self, widget, string):
 94         print "%s" % string
 95
 96 def main():
 97     gtk.main()
 98     return 0
 99
100 if __name__ == "__main__":
101     MenuExample()
102     main()
```

You may also set a menu item to be insensitive and, using an accelerator table, bind keys to menu callbacks.

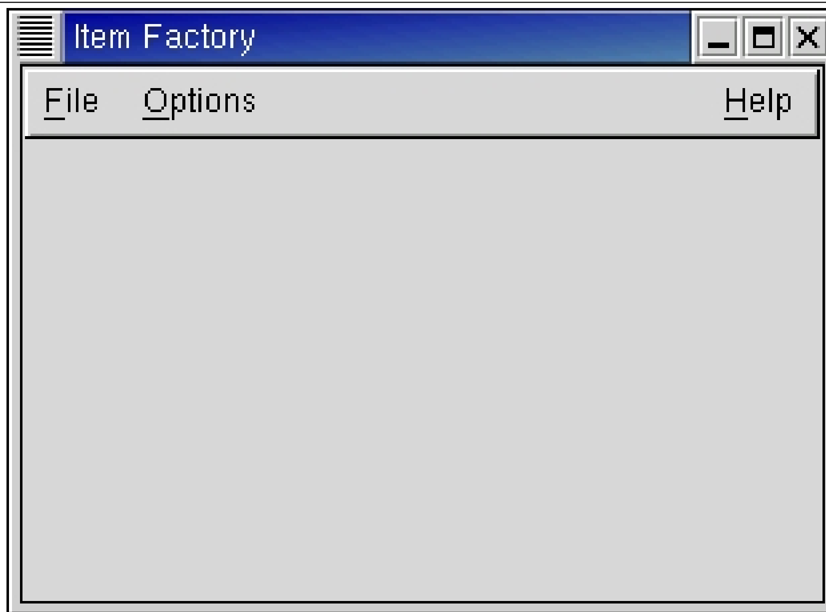
11.3 Using ItemFactory

Now that we've shown you the hard way, here's how you do it using the `gtk.ItemFactory` calls.

11.4 Item Factory Example

The itemfactory.py example program uses the `gtk.ItemFactory`. Figure 11.2 illustrates the program display:

Figure 11.2 Item Factory Example



The source code for itemfactory.py is:

```
 1 #!/usr/bin/env python
 2
 3 # example itemfactory.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ItemFactoryExample:
10     # Obligatory basic callback
11     def print_hello(self, w, data):
12         print "Hello, World!"
13
14     # This is the ItemFactoryEntry structure used to generate new menus.
15     # Item 1: The menu path. The letter after the underscore indicates an
16     #           accelerator key once the menu is open.
17     # Item 2: The accelerator key for the entry
18     # Item 3: The callback.
19     # Item 4: The callback action.  This changes the parameters with
20     #           which the callback is called.  The default is 0.
21     # Item 5: The item type, used to define what kind of an item it is.
22     #         Here are the possible values:
23
24     #         NULL               -> "<Item>"
25     #         ""                 -> "<Item>"
26     #         "<Title>"          -> create a title item
27     #         "<Item>"           -> create a simple item
28     #         "<CheckItem>"      -> create a check item
29     #         "<ToggleItem>"     -> create a toggle item
30     #         "<RadioItem>"      -> create a radio item
31     #         <path>             -> path of a radio item to link against
32     #         "<Separator>"      -> create a separator
33     #         "<Branch>"         -> create an item to hold sub items ( ↩
   optional)
34     #         "<LastBranch>"     -> create a right justified branch
35
36     def get_main_menu(self, window):
37         accel_group = gtk.AccelGroup()
38
```

```
39         # This function initializes the item factory.
40         # Param 1: The type of menu - can be MenuBar, Menu,
41         #          or OptionMenu.
42         # Param 2: The path of the menu.
43         # Param 3: A reference to an AccelGroup. The item factory sets up
44         #          the accelerator table while generating menus.
45         item_factory = gtk.ItemFactory(gtk.MenuBar, "<main>", accel_group)
46 
47         # This method generates the menu items. Pass to the item factory
48         #  the list of menu items
49         item_factory.create_items(self.menu_items)
50 
51         # Attach the new accelerator group to the window.
52         window.add_accel_group(accel_group)
53 
54         # need to keep a reference to item_factory to prevent its  ←
   destruction
55         self.item_factory = item_factory
56         # Finally, return the actual menu bar created by the item factory.
57         return item_factory.get_widget("<main>")
58 
59     def __init__(self):
60         self.menu_items = (
61             ( "/_File",         None,         None, 0, "<Branch>" ),
62             ( "/File/_New",     "<control>N", self.print_hello, 0, None ),
63             ( "/File/_Open",    "<control>O", self.print_hello, 0, None ),
64             ( "/File/_Save",    "<control>S", self.print_hello, 0, None ),
65             ( "/File/Save _As", None,         None, 0, None ),
66             ( "/File/sep1",     None,         None, 0, "<Separator>" ),
67             ( "/File/Quit",     "<control>Q", gtk.main_quit, 0, None ),
68             ( "/_Options",      None,         None, 0, "<Branch>" ),
69             ( "/Options/Test",  None,         None, 0, None ),
70             ( "/_Help",         None,         None, 0, "<LastBranch>" ),
71             ( "/_Help/About",   None,         None, 0, None ),
72             )
73         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
74         window.connect("destroy", lambda w: gtk.main_quit(), "WM destroy")
75         window.set_title("Item Factory")
76         window.set_size_request(300, 200)
77 
78         main_vbox = gtk.VBox(False, 1)
79         main_vbox.set_border_width(1)
80         window.add(main_vbox)
81         main_vbox.show()
82 
83         menubar = self.get_main_menu(window)
84 
85         main_vbox.pack_start(menubar, False, True, 0)
86         menubar.show()
87         window.show()
88 
89 def main():
90     gtk.main()
91     return 0
92 
93 if __name__ == "__main__":
94     ItemFactoryExample()
95     main()
```

For now, there's only this example. An explanation and lots 'o' comments will follow later.

Chapter 12

Drawing Area

The `DrawingArea` widget wraps a `gtk.gdk.Window` which is a subclass of `gtk.gdk.Drawable` (as is a `gtk.gdk.Pixmap`). In effect the `DrawingArea` provides a simple 'canvas' area (the wrapped `gtk.gdk.Window`) that can be drawn on using the methods of the `gtk.gdk.Drawable` class.

A `DrawingArea` is created using the constructor:

```
drawing_area = gtk.DrawingArea()
```

A `DrawingArea` is initially created with a size of (0, 0) so you should use the following method to make the *drawing_area* visible by setting its width and height to useful values greater than zero:

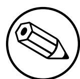
```
drawing_area.set_size_request(width, height)
```

To draw on a `DrawingArea` you must retrieve the wrapped `gtk.gdk.Window` using the *window* attribute of the `DrawingArea` as follows:

```
drawable = drawing_area.window
```

Then you can draw on *drawable* using the `gtk.gdk.Drawable` methods described in Section 12.2.

NOTE



The `DrawingArea` must be realized (i.e. the Widget methods realize() or show() have been called) to have an associated `gtk.gdk.Window` that can be used for drawing.

12.1 Graphics Context

A variety of methods are available to draw onto the `gtk.gdk.Window` of a `DrawingArea`. All these methods require a graphics context (`gtk.gdk.GC`) to encapsulate, as attributes, the information required for drawing. The attributes of a `gtk.gdk.GC` are:

```
background
cap_style
clip_mask
clip_x_origin
clip_y_origin
fill
font
foreground
function
graphics_exposures
join_style
line_style
line_width
stipple
sub_window
```

```
tile
ts_x_origin
ts_y_origin
```

background specifies an allocated `gtk.gdk.Color` that is used to draw the background color.
foreground specifies an allocated `gtk.gdk.Color` that is used to draw the foreground color.
A `gtk.gdk.Color` represents a color that may be allocated or unallocated. An unallocated color can be created using the constructor:

```
color = gtk.gdk.Color(red=0, green=0, blue=0, pixel=0)
```

where *red*, *green* and *blue* are integers in the range of 0 to 65535. *pixel* is not usually specified because it is overwritten when the color is allocated.
Alternatively, an unallocated `gtk.gdk.Color` can be created using the function:

```
color = gtk.gdk.color_parse(spec)
```

where *spec* is a color specification string that can be either:

- a color name (e.g. "red", "orange", "navajo white" as defined in the X Window file `rgb.txt`), or
- a hexadecimal string starting with '#' and containing three sets of hex digits of the same length (1, 2, 3 or 4 digits). For example, "#F0A", "#FF00AA", "#FFF000AAA" and "#FFFF0000AAAA" all represent the same color.

A `gtk.gdk.Color` representing an allocated color is created using the `gtk.gdk.Colormap` alloc_color() method which has three signatures:

```
color = colormap.alloc_color(color, writeable=FALSE, best_match=TRUE)

color = colormap.alloc_color(spec, writeable=FALSE, best_match=TRUE)

color = colormap.alloc_color(red, green, blue, writeable=FALSE, best_match=TRUE ↩
    )
```

color is an unallocated `gtk.gdk.Color`. *spec* is a color specification string as described above for the `gtk.gdk.color_parse`() function. *red*, *green* and *blue* are integer color values as described for the `gtk.gdk.Color`() constructor. You can optionally specify whether the allocated color should be writeable (i.e. can be changed later but cannot be shared) or whether a best match with existing colors should be made if the exact color is not available.
For example:

```
navajowhite = colormap.alloc('navajo white')

cyan = colormap.alloc(0, 65535, 65535)

red = colormap.alloc_color('#FF0000', True, True)
```

The colormap associated with a widget can be retrieved using the method:

```
colormap = widget.get_colormap()
```

cap_style specifies the line ending style that is used when drawing the end of a line that is not joined to another line. The available cap styles are:

`CAP_NOT_LAST`	draws line ends the same as `CAP_BUTT` for lines of non-zero width. For zero width lines, the final point on the line will not be drawn.
`CAP_BUTT`	the ends of the lines are drawn squared off and extending to the coordinates of the end point.
`CAP_ROUND`	the ends of the lines are drawn as semicircles with the diameter equal to the line width and centered at the end point.
`CAP_PROJECTING`	the ends of the lines are drawn squared off and extending half the width of the line beyond the end point.

clip_mask specifies a `gtk.gdk.Pixmap` that is used to clip the drawing in the *drawing_area*.

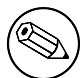
clip_x_origin and *clip_y_origin* specify the origin x and y values relative to the upper left corner of the *drawing_area* for clipping.

fill specifies the fill style to be used when drawing. The available fill styles are:

SOLID	draw with the foreground color.
TILED	draw with a tiled pixmap.
STIPPLED	draw using the stipple bitmap. Pixels corresponding to bits in the stipple bitmap that are set will be drawn in the foreground color; pixels corresponding to bits that are not set will be left untouched.
OPAQUE_STIPPLED	draw using the stipple bitmap. Pixels corresponding to bits in the stipple bitmap that are set will be drawn in the foreground color; pixels corresponding to bits that are not set will be drawn with the background color.

font is a `gtk.gdk.Font` that is used as the default font for drawing text.

NOTE



The use of the *font* attribute is deprecated.

function specifies how the bit values for the source pixels are combined with the bit values for destination pixels to produce the resulting pixels bits. The sixteen values here correspond to the 16 different possible 2x2 truth tables but only a couple of these values are usually useful. For color images, only COPY, XOR and INVERT are generally useful while for bitmaps, AND and OR are also useful. The function values are:

```
COPY
INVERT
XOR
CLEAR
AND
AND_REVERSE
AND_INVERT
NOOP
OR
EQUIV
OR_REVERSE
COPY_INVERT
OR_INVERT
NAND
SET
```

graphics_exposures specifies whether graphics exposures are enabled (`TRUE`) or disabled (`FALSE`). When *graphics_exposures* is `TRUE`, a failure when copy pixels in a drawing operation will cause an expose event to be issued. If the copy succeeds, a noexpose event is issued.

join_style specifies the style of joint to be used when lines meet at an angle. The available styles are:

JOIN_MITER	the sides of each line are extended to meet at an angle.
JOIN_ROUND	the sides of the two lines are joined by a circular arc.
JOIN_BEVEL	the sides of the two lines are joined by a straight line which makes an equal angle with each line.

line_style specifies the style that a line will be drawn with. The available styles are:

LINE_SOLID	lines are drawn as continuous segments.
LINE_ON_OFF_DASH	even segments are drawn; odd segments are not drawn.

LINE_DOUBLE_DASH	even segments are normally. Odd segments are drawn in the background color if the fill style is SOLID, or in the background color masked by the stipple if the fill style is STIPPLED.

line_width specifies the width that lines will be drawn with.

stipple specifies the gtk.gdk.Pixmap that will be used for stippled drawing when the *fill* is set to either STIPPLED or OPAQUE_STIPPLED.

sub_window specifies the mode of drawing into a gtk.gdk.Window that has child gtk.gdk.Windows. The possible values of *sub_window* are:

CLIP_BY_CHILDREN	only draw onto the window itself but not its child windows
INCLUDE_INFERIORS	draw onto the window and its child windows.

tile specifies the gtk.gdk.Pixmap to used for tiled drawing when the *fill* is set to TILED.

ts_x_origin and *ts_y_origin* specify the tiling/stippling origin (the starting position for the stippling bitmap or tiling pixmap).

A new Graphics Context is created by a call to the gtk.gdk.Drawable.new_gc() method:

```
gc = drawable.new_gc(foreground=None, background=None, font=None,
                     function=-1, fill=-1, tile=None,
                     stipple=None, clip_mask=None, subwindow_mode=-1,
                     ts_x_origin=-1, ts_y_origin=-1, clip_x_origin=-1,
                     clip_y_origin=-1, graphics_exposures=-1,
                     line_width=-1, line_style=-1, cap_style=-1
                     join_style=-1)
```

In order for a new Graphics Context to be created with this method, the drawable must be:

- a gtk.gdk.Window which has been realized (created), or;
- a gtk.gdk.Pixmap associated with a realized gtk.gdk.Window.

The various attributes of the Graphics Context have default values if not set in the new_gc() method. If you want to set the GC attributes using the new_gc() method, it's much easier to use the Python keyword arguments.

The individual attributes of a gtk.gdk.GC can also be set by assigning a value to the GC object attribute. For example:

```
  gc.cap_style = CAP_BUTT
  gc.line_width = 10
  gc.fill = SOLD
  gc.foreground = mycolor
```

or by using the following methods:

```
  gc.set_foreground(color)
  gc.set_background(color)
  gc.set_function(function)
  gc.set_fill(fill)
  gc.set_tile(tile)
  gc.set_stipple(stipple)
  gc.set_ts_origin(x, y)
  gc.set_clip_origin(x, y)
  gc.set_clip_mask(mask)
  gc.set_clip_rectangle(rectangle)
  gc.set_subwindow(mode)
  gc.set_exposures(exposures)
  gc.set_line_attributes(line_width, line_style, cap_style, join_style)
```

The dash pattern to be used when the *line_style* is LINE_ON_OFF_DASH or LINE_DOUBLE_DASH can be set using the following method:

```
  gc.set_dashes(offset, dash_list)
```

where *offset* is the index of the starting dash value in *dash_list* and *dash_list* is a list or tuple containing numbers of pixels to be drawn or skipped to form the dashes. The dashes are drawn starting with the number of pixels at the offset position; then the next number of pixels is skipped; and then the next number is drawn; and so on rotating through all the dash_list numbers and starting over when the end is reached. For example, if the dash_list is (2, 4, 8, 16) and the offset is 1, the dashes will be drawn as: draw 4 pixels, skip 8 pixels, draw 16 pixels, skip 2 pixels, draw 4 pixels and so on.

A copy of an existing `gtk.gdk.GC` can be made using the method:

```
gc.copy(src_gc)
```

The attributes of *gc* will then be the same as *src_gc*.

12.2 Drawing Methods

There are a general set of methods that can be used to draw onto the drawing area 'canvas'. These drawing methods can be used for any `gtk.gdk.Drawable` subclass (either a `gtk.gdk.Window` or a `gtk.gdk.Pixmap`). The drawing methods are:

```
drawable.draw_point(gc, x, y)
```

gc is the Graphics Context to be used to do the drawing.
x and *y* are the coordinates of the point.

```
drawable.draw_line(gc, x1, y1, x2, y2)
```

gc is the Graphics Context.
x1 and *y1* specify the starting point of the line. *x2* and *y2* specify the ending point of the line.

```
drawable.draw_rectangle(gc, filled, x, y, width, height)
```

where *gc* is the Graphics Context.
filled is a boolean indicating the rectangle should be filled with the foreground color if `TRUE` or not filled, if `FALSE`.
x and *y* are the top left corner of the rectangle.
width and *height* are the width and height of the rectangle.

```
drawable.draw_arc(gc, filled, x, y, width, height, angle1, angle2)
```

gc is the Graphics Context.
filled is a boolean indicating the arc should be filled with the foreground color if `TRUE` or not filled, if `FALSE`.
x and *y* are the top left corner of the bounding rectangle. *width* and *height* are the width and height of the bounding rectangle.
angle1 is the start angle of the arc, relative to the 3 o'clock position, counter-clockwise, in 1/64ths of a degree.
angle2 is the end angle of the arc, relative to *angle1*, in 1/64ths of a degree counter clockwise.

```
drawable.draw_polygon(gc, filled, points)
```

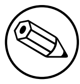
gc is the Graphics Context.
filled is a boolean indicating the polygon should be filled with the foreground color if `TRUE` or not filled, if `FALSE`.
points is a list of coordinate pairs in tuples e.g. [(0,0), (2,5), (3,7), (4,11)] of the points to be drawn as a connected polygon.

```
drawable.draw_string(font, gc, x, y, string)

drawable.draw_text(font, gc, x, y, string)
```

font is the `gtk.gdk.Font` to use to render the string.
gc is the Graphics Context.
x and *y* are the coordinates of the point to start rendering the string i.e the left baseline.

string is the string of characters to render.

NOTE



Both the draw_string() and draw_text() methods are deprecated - use a `pango.Layout` instead with the draw_layout() method.

```
drawable.draw_layout(gc, x, y, layout)
```

gc is the Graphics Context.
x and *y* are the coordinates of the point to start rendering the layout.
layout is the `pango.Layout` that is to be rendered.

```
drawable.draw_drawable(gc, src, xsrc, ysrc, xdest, ydest, width, height)
```

gc is the Graphics Context.
src is the source drawable.
xsrc and *ysrc* are the coordinates of the top left rectangle in the source drawable.
xdest and *ydest* are the coordinates of the top left corner in the drawing area.
width and *height* are the width and height of the source drawable area to be copied to the *drawable*. If *width* or *height* is -1 then the full width or height of the *drawable* is used.

```
drawable.draw_image(gc, image, xsrc, ysrc, xdest, ydest, width, height)
```

gc is the Graphics Context.
image is the source image.
xsrc and *ysrc* are the coordinates of the top left rectangle in the source drawable.
xdest and *ydest* are the coordinates of the top left corner in the drawing area.
width and *height* are the width and height of the source drawable area to be copied to the *drawable*. If *width* or *height* is -1 then the full width or height of the *image* is used.

```
drawable.draw_points(gc, points)
```

gc is the Graphics Context.
points is a list or tuple of coordinate pairs in tuples e.g. [(0,0), (2,5), (3,7), (4,11)] of the points to be drawn.

```
drawable.draw_segments(gc, segs)
```

gc is the Graphics Context.
segs is a list or tuple of start and end coordinate pairs in tuples e.g. [(0,0, 1,5), (2,5, 1,7), (3,7, 1,11), (4,11, 1,13)] of the line segments to be drawn.

```
drawable.draw_lines(gc, points)
```

gc is the Graphics Context.
points is a list or tuple of coordinate pairs in tuples e.g. [(0,0), (2,5), (3,7), (4,11)] of the points to be connected with lines.

```
drawable.draw_rgb_image(gc, x, y, width, height, dith, rgb_buf, rowstride)

drawable.draw_rgb_32_image(gc, x, y, width, height, dith, buf, rowstride)

drawable.draw_gray_image(gc, x, y, width, height, dith, buf, rowstride)
```

gc is the Graphics Context.
x and *y* are the top left corner of the image bounding rectangle.
width and *height* are the width and height of the image bounding rectangle.
dith is the dither mode as described below
For the draw_rgb_image() method, *rgb_buf* is the RGB Image data packed in a string as a sequence of 8-bit RGB pixel triplets. For the draw_rgb_32_image() method, *buf* is the RGB Image data packed in

a string as a sequence of 8-bit RGB pixel triplets with 8-bit padding (4 characters per RGB pixel). For the draw_gray_image() method, *buf* is the gray image data packed in a string as 8-bit pixel data.

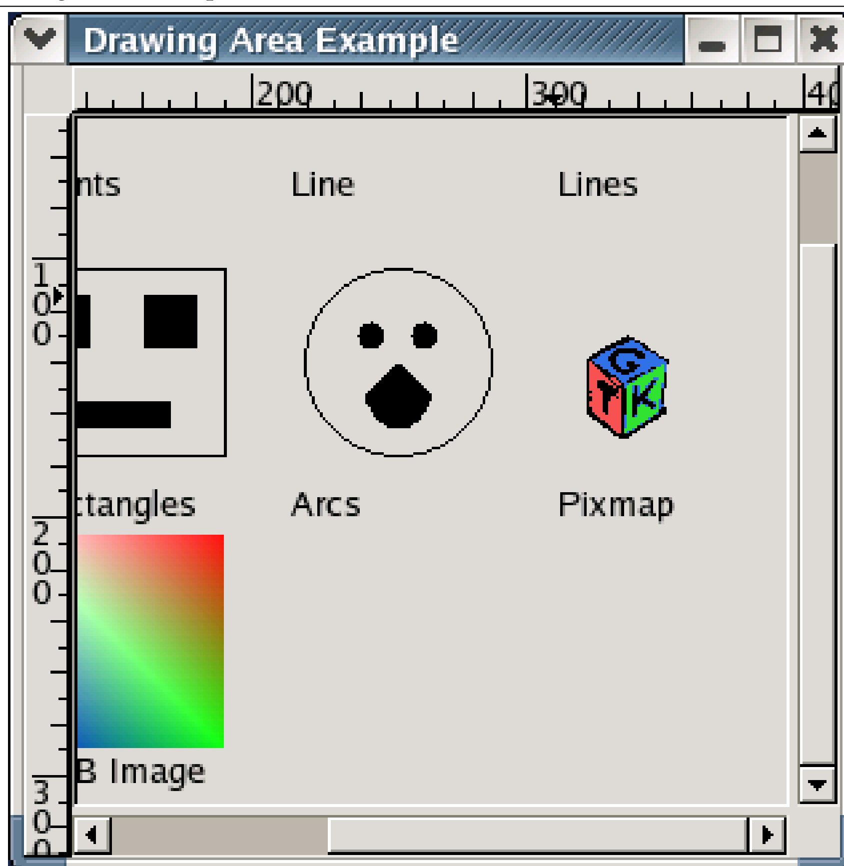
rowstride is the number of characters from the start of one row to the start of the next row of the image. *rowstride* usually defaults to: 3 * *width* for the draw_rgb_image() method; 4 * *width* for the draw_rgb_32_image(); and, *width* for the draw_gray_image() method. If *rowstride* is 0 the line will be replicated *height* times.

The *dither* modes are:

```
RGB_DITHER_NONE    # Never use dithering.

RGB_DITHER_NORMAL  # Use dithering in 8 bits per pixel (and below) only.

RGB_DITHER_MAX     # Use dithering in 16 bits per pixel and below.
```

The drawingarea.py example program demonstrates the use of most of the `DrawingArea` methods. It also puts the `DrawingArea` inside a `ScrolledWindow` and adds horizontal and vertical `Ruler` widgets. Figure 12.1 shows the program in operation:

Figure 12.1 Drawing Area Example



The drawingarea.py source code is below and uses the gtk.xpm pixmap:

```
1   #!/usr/bin/env python
2
3   # example drawingarea.py
4
5   import pygtk
6   pygtk.require('2.0')
7   import gtk
8   import operator
9   import time
10  import string
11
```

```
12  class DrawingAreaExample:
13      def __init__(self):
14          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
15          window.set_title("Drawing Area Example")
16          window.connect("destroy", lambda w: gtk.main_quit())
17          self.area = gtk.DrawingArea()
18          self.area.set_size_request(400, 300)
19          self.pangolayout = self.area.create_pango_layout("")
20          self.sw = gtk.ScrolledWindow()
21          self.sw.add_with_viewport(self.area)
22          self.table = gtk.Table(2,2)
23          self.hruler = gtk.HRuler()
24          self.vruler = gtk.VRuler()
25          self.hruler.set_range(0, 400, 0, 400)
26          self.vruler.set_range(0, 300, 0, 300)
27          self.table.attach(self.hruler, 1, 2, 0, 1, yoptions=0)
28          self.table.attach(self.vruler, 0, 1, 1, 2, xoptions=0)
29          self.table.attach(self.sw, 1, 2, 1, 2)
30          window.add(self.table)
31          self.area.set_events(gtk.gdk.POINTER_MOTION_MASK |
32                               gtk.gdk.POINTER_MOTION_HINT_MASK )
33          self.area.connect("expose-event", self.area_expose_cb)
34          def motion_notify(ruler, event):
35              return ruler.emit("motion_notify_event", event)
36          self.area.connect_object("motion_notify_event", motion_notify,
37                                   self.hruler)
38          self.area.connect_object("motion_notify_event", motion_notify,
39                                   self.vruler)
40          self.hadj = self.sw.get_hadjustment()
41          self.vadj = self.sw.get_vadjustment()
42          def val_cb(adj, ruler, horiz):
43              if horiz:
44                  span = self.sw.get_allocation()[3]
45              else:
46                  span = self.sw.get_allocation()[2]
47              l,u,p,m = ruler.get_range()
48              v = adj.value
49              ruler.set_range(v, v+span, p, m)
50              while gtk.events_pending():
51                  gtk.main_iteration()
52          self.hadj.connect('value-changed', val_cb, self.hruler, True)
53          self.vadj.connect('value-changed', val_cb, self.vruler, False)
54          def size_allocate_cb(wid, allocation):
55              x, y, w, h = allocation
56              l,u,p,m = self.hruler.get_range()
57              m = max(m, w)
58              self.hruler.set_range(l, l+w, p, m)
59              l,u,p,m = self.vruler.get_range()
60              m = max(m, h)
61              self.vruler.set_range(l, l+h, p, m)
62          self.sw.connect('size-allocate', size_allocate_cb)
63          self.area.show()
64          self.hruler.show()
65          self.vruler.show()
66          self.sw.show()
67          self.table.show()
68          window.show()
69
70      def area_expose_cb(self, area, event):
71          self.style = self.area.get_style()
72          self.gc = self.style.fg_gc[gtk.STATE_NORMAL]
73          self.draw_point(10,10)
74          self.draw_points(110, 10)
75          self.draw_line(210, 10)
```

```
 76            self.draw_lines(310, 10)
 77            self.draw_segments(10, 100)
 78            self.draw_rectangles(110, 100)
 79            self.draw_arcs(210, 100)
 80            self.draw_pixmap(310, 100)
 81            self.draw_polygon(10, 200)
 82            self.draw_rgb_image(110, 200)
 83            return True
 84
 85        def draw_point(self, x, y):
 86            self.area.window.draw_point(self.gc, x+30, y+30)
 87            self.pangolayout.set_text("Point")
 88            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ←
    )
 89            return
 90
 91        def draw_points(self, x, y):
 92            points = [(x+10,y+10), (x+10,y), (x+40,y+30),
 93                      (x+30,y+10), (x+50,y+10)]
 94            self.area.window.draw_points(self.gc, points)
 95            self.pangolayout.set_text("Points")
 96            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ←
    )
 97            return
 98
 99        def draw_line(self, x, y):
100            self.area.window.draw_line(self.gc, x+10, y+10, x+20, y+30)
101            self.pangolayout.set_text("Line")
102            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ←
    )
103            return
104
105        def draw_lines(self, x, y):
106            points = [(x+10,y+10), (x+10,y), (x+40,y+30),
107                      (x+30,y+10), (x+50,y+10)]
108            self.area.window.draw_lines(self.gc, points)
109            self.pangolayout.set_text("Lines")
110            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ←
    )
111            return
112
113        def draw_segments(self, x, y):
114            segments = ((x+20,y+10, x+20,y+70), (x+60,y+10, x+60,y+70),
115                (x+10,y+30 , x+70,y+30), (x+10, y+50 , x+70, y+50))
116            self.area.window.draw_segments(self.gc, segments)
117            self.pangolayout.set_text("Segments")
118            self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ←
    )
119            return
120
121        def draw_rectangles(self, x, y):
122            self.area.window.draw_rectangle(self.gc, False, x, y, 80, 70)
123            self.area.window.draw_rectangle(self.gc, True, x+10, y+10, 20,  ←
    20)
124            self.area.window.draw_rectangle(self.gc, True, x+50, y+10, 20,  ←
    20)
125            self.area.window.draw_rectangle(self.gc, True, x+20, y+50, 40,  ←
    10)
126            self.pangolayout.set_text("Rectangles")
127            self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ←
    )
128            return
129
130        def draw_arcs(self, x, y):
```

```
131             self.area.window.draw_arc(self.gc, False, x+10, y, 70, 70,
132                                       0, 360*64)
133             self.area.window.draw_arc(self.gc, True, x+30, y+20, 10, 10,
134                                       0, 360*64)
135             self.area.window.draw_arc(self.gc, True, x+50, y+20, 10, 10,
136                                       0, 360*64)
137             self.area.window.draw_arc(self.gc, True, x+30, y+10, 30, 50,
138                                       210*64, 120*64)
139             self.pangolayout.set_text("Arcs")
140             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ←
    )
141             return
142 
143         def draw_pixmap(self, x, y):
144             pixmap, mask = gtk.gdk.pixmap_create_from_xpm(
145                 self.area.window, self.style.bg[gtk.STATE_NORMAL], "gtk.xpm")
146 
147             self.area.window.draw_drawable(self.gc, pixmap, 0, 0, x+15, y+25,
148                                            -1, -1)
149             self.pangolayout.set_text("Pixmap")
150             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ←
    )
151             return
152 
153         def draw_polygon(self, x, y):
154             points = [(x+10,y+60), (x+10,y+20), (x+40,y+70),
155                       (x+30,y+30), (x+50,y+40)]
156             self.area.window.draw_polygon(self.gc, True, points)
157             self.pangolayout.set_text("Polygon")
158             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ←
    )
159             return
160 
161         def draw_rgb_image(self, x, y):
162             b = 80*3*80*['\0']
163             for i in range(80):
164                 for j in range(80):
165                     b[3*80*i+3*j] = chr(255-3*i)
166                     b[3*80*i+3*j+1] = chr(255-3*abs(i-j))
167                     b[3*80*i+3*j+2] = chr(255-3*j)
168             buff = string.join(b, '')
169             self.area.window.draw_rgb_image(self.gc, x, y, 80, 80,
170                                      gtk.gdk.RGB_DITHER_NONE, buff, 80*3)
171             self.pangolayout.set_text("RGB Image")
172             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ←
    )
173             return
174 
175     def main():
176         gtk.main()
177         return 0
178 
179     if __name__ == "__main__":
180         DrawingAreaExample()
181         main()
```

Chapter 13

TextView Widget

13.1 TextView Overview

TextView widgets and their associated objects (TextBuffers, TextMarks, TextIters, TextTags and TextTagTables) provide a powerful framework for multiline text editing.

A TextBuffer (see Section 13.3) contains the text which is displayed by one or more TextView widgets.

Within GTK+ 2.0, text is encoded in UTF-8 which means that one character may be encoded as multiple bytes. Within a TextBuffer it is necessary to differentiate between the character counts (called offsets) and the byte counts (called indexes).

TextIters provide a volatile representation of the position in a TextBuffer between two characters. TextIters are valid until the number of characters in the TextBuffer changes; i.e. any time characters are inserted or deleted from a TextBuffer all TextIters will become invalid. TextIters are the primary way to specify locations in a TextBuffer for manipulating text.

TextMarks are provided to allow preservation of TextBuffer positions across buffer modifications. A mark is like a TextIter (see Section 13.4) in that it represents a position between two characters in a TextBuffer) but if the text surrounding the mark is deleted the mark remains where the deleted text once was. Likewise, if text is inserted at the mark the mark ends up either to the left or right of the inserted text depending on the gravity of the mark - right gravity leaves the mark to the right of the inserted text while left gravity leaves it to the left. TextMarks (see Section 13.5) may be named or anonymous if not given a name. Each TextBuffer has two predefined named TextMarks (see Section 13.5) called *insert* and *selection_bound*. These refer to the insertion point and the boundary of the selection (the selection is between the *insert* and the *selection_bound* marks).

TextTags (see Section 13.6.1) are objects that specify a set of attributes that can be applied to a range of text in a TextBuffer. Each TextBuffer has a TextTagTable (see Section 13.6.2) which contains the tags that are available in that buffer. TextTagTables can be shared between TextBuffers to provide commonality. TextTags are generally used to change the appearance of a range of text but can also be used to prevent a range of text from being edited.

13.2 TextViews

There is only one function for creating a new TextView widget.

```
textview = gtk.TextView(buffer=None)
```

When a TextView is created it will create an associated TextBuffer and TextTagTable by default. If you want to use an existing TextBuffer in a TextView specify it in the above method. To change the TextBuffer used by a TextView use the following method:

```
textview.set_buffer(buffer)
```

Use the following method to retrieve a reference to the TextBuffer from a TextView:

```
buffer = textview.get_buffer()
```

A TextView widget doesn't have scrollbars to adjust the view in case the text is larger than the window. To provide scrollbars, you add the TextView to a ScrolledWindow (see Section 10.9).

A TextView can be used to allow the user to edit a body of text, or to display multiple lines of read-only text to the user. To switch between these modes of operation, the use the following method:

```
textview.set_editable(setting)
```

The *setting* argument is a TRUE or FALSE value that specifies whether the user is permitted to edit the contents of the TextView widget. The editable mode of the TextView can be overridden in text ranges within the TextBuffer by TextTags.

You can retrieve the current editable setting using the method:

```
setting = textview.get_editable()
```

When the TextView is not editable, you probably should hide the cursor using the method:

```
textview.set_cursor_visible(setting)
```

The *setting* argument is a TRUE or FALSE value that specifies whether the cursor should be visible The TextView can wrap lines of text that are too long to fit onto a single line of the display window. Its default behavior is to not wrap lines. This can be changed using the next method:

```
textview.set_wrap_mode(wrap_mode)
```

This method allows you to specify that the text widget should wrap long lines on word or character boundaries. The *word_wrap* argument is one of:

```
gtk.WRAP_NONE
gtk.WRAP_CHAR
gtk.WRAP_WORD
```

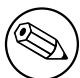
The default justification of the text in a TextView can be set and retrieved using the methods:

```
textview.set_justification(justification)
justification = textview.get_justification()
```

where *justification* is one of:

```
gtk.JUSTIFY_LEFT
gtk.JUSTIFY_RIGHT
gtk.JUSTIFY_CENTER
```

NOTE



The *justification* will be JUSTIFY_LEFT if the *wrap_mode* is WRAP_NONE. Tags in the associated TextBuffer may override the default justification.

Other default attributes that can be set and retrieved in a TextView are: left margin, right margin, tabs, and paragraph indentation using the following methods:

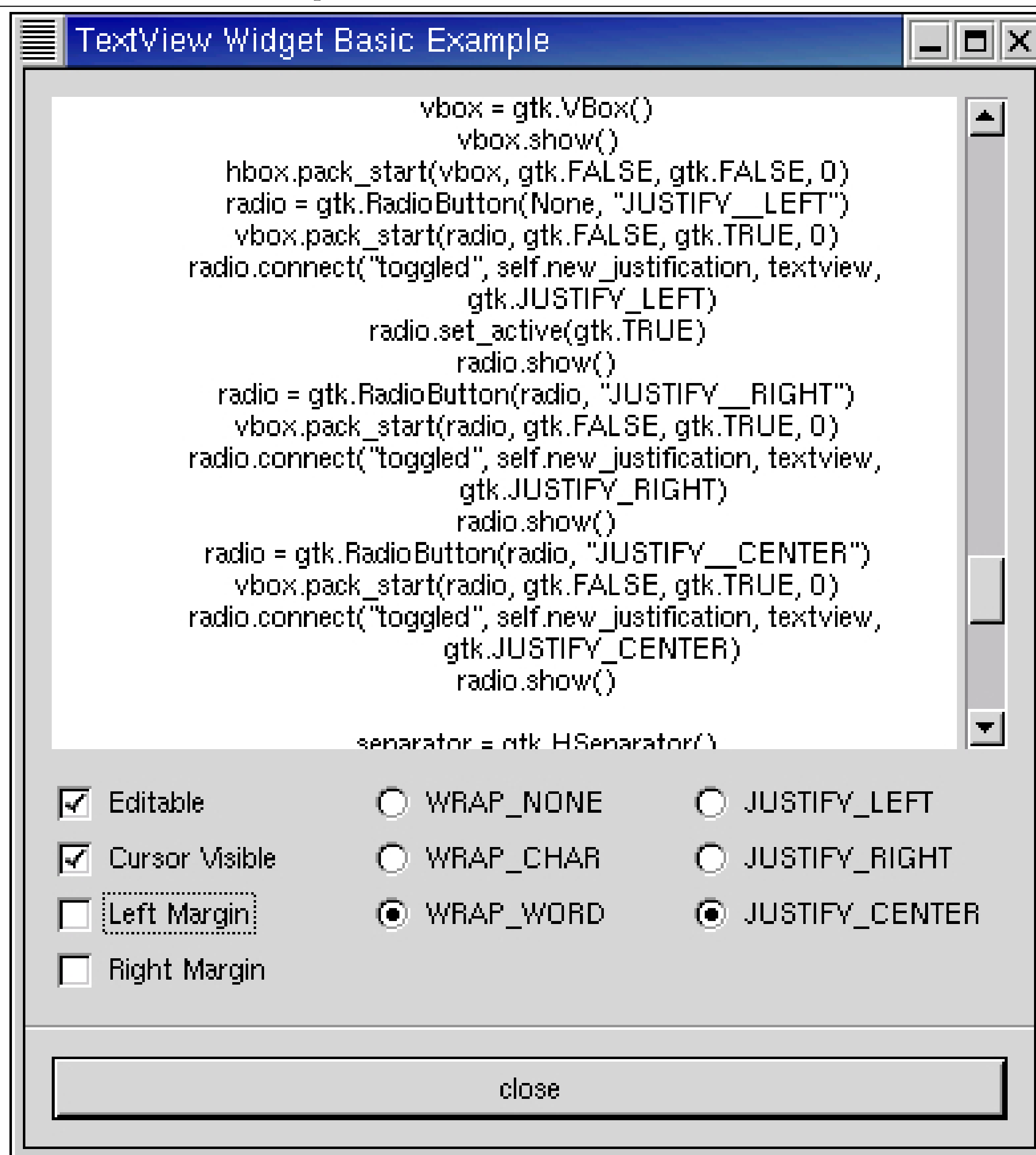
```
textview.set_left_margin(left_margin)
left_margin = textview.get_left_margin()

textview.set_right_margin(right_margin)
right_margin = textview.get_right_margin()

textview.set_indent(indent)
indent = textview.get_indent()

textview.set_pixels_above_lines(pixels_above_line)
pixels_above_line = textview.get_pixels_above_lines()
```

```
textview.set_pixels_below_lines(pixels_below_line)
pixels_below_line = textview.get_pixels_below_lines()

textview.set_pixels_inside_wrap(pixels_inside_wrap)
pixels_inside_wrap = textview.get_pixels_inside_wrap()

textview.set_tabs(tabs)
tabs = textview.get_tabs()
```

left_margin, *right_margin*, *indent*, *pixels_above_lines*, *pixels_below_lines* and *pixels_-inside_wrap* are specified in pixels. These default values may be overridden by tags in the associated `TextBuffer`. *tabs* is a `pango.TabArray`.

The textview-basic.py example program illustrates basic use of the `TextView` widget:

Figure 13.1 Basic TextView Example



The source code for the program is:

```
1 #!/usr/bin/env python
2
3 # example textview-basic.py
4
5 import pygtk
```

```
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class TextViewExample:
10     def toggle_editable(self, checkbutton, textview):
11         textview.set_editable(checkbutton.get_active())
12
13     def toggle_cursor_visible(self, checkbutton, textview):
14         textview.set_cursor_visible(checkbutton.get_active())
15
16     def toggle_left_margin(self, checkbutton, textview):
17         if checkbutton.get_active():
18             textview.set_left_margin(50)
19         else:
20             textview.set_left_margin(0)
21
22     def toggle_right_margin(self, checkbutton, textview):
23         if checkbutton.get_active():
24             textview.set_right_margin(50)
25         else:
26             textview.set_right_margin(0)
27
28     def new_wrap_mode(self, radiobutton, textview, val):
29         if radiobutton.get_active():
30             textview.set_wrap_mode(val)
31
32     def new_justification(self, radiobutton, textview, val):
33         if radiobutton.get_active():
34             textview.set_justification(val)
35
36     def close_application(self, widget):
37         gtk.main_quit()
38
39     def __init__(self):
40         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
41         window.set_resizable(True)
42         window.connect("destroy", self.close_application)
43         window.set_title("TextView Widget Basic Example")
44         window.set_border_width(0)
45
46         box1 = gtk.VBox(False, 0)
47         window.add(box1)
48         box1.show()
49
50         box2 = gtk.VBox(False, 10)
51         box2.set_border_width(10)
52         box1.pack_start(box2, True, True, 0)
53         box2.show()
54
55         sw = gtk.ScrolledWindow()
56         sw.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
57         textview = gtk.TextView()
58         textbuffer = textview.get_buffer()
59         sw.add(textview)
60         sw.show()
61         textview.show()
62
63         box2.pack_start(sw)
64         # Load the file textview-basic.py into the text window
65         infile = open("textview-basic.py", "r")
66
67         if infile:
68             string = infile.read()
69             infile.close()
```

```
 70             textbuffer.set_text(string)
 71 
 72         hbox = gtk.HButtonBox()
 73         box2.pack_start(hbox, False, False, 0)
 74         hbox.show()
 75 
 76         vbox = gtk.VBox()
 77         vbox.show()
 78         hbox.pack_start(vbox, False, False, 0)
 79         # check button to toggle editable mode
 80         check = gtk.CheckButton("Editable")
 81         vbox.pack_start(check, False, False, 0)
 82         check.connect("toggled", self.toggle_editable, textview)
 83         check.set_active(True)
 84         check.show()
 85         # check button to toggle cursor visiblity
 86         check = gtk.CheckButton("Cursor Visible")
 87         vbox.pack_start(check, False, False, 0)
 88         check.connect("toggled", self.toggle_cursor_visible, textview)
 89         check.set_active(True)
 90         check.show()
 91         # check button to toggle left margin
 92         check = gtk.CheckButton("Left Margin")
 93         vbox.pack_start(check, False, False, 0)
 94         check.connect("toggled", self.toggle_left_margin, textview)
 95         check.set_active(False)
 96         check.show()
 97         # check button to toggle right margin
 98         check = gtk.CheckButton("Right Margin")
 99         vbox.pack_start(check, False, False, 0)
100         check.connect("toggled", self.toggle_right_margin, textview)
101         check.set_active(False)
102         check.show()
103         # radio buttons to specify wrap mode
104         vbox = gtk.VBox()
105         vbox.show()
106         hbox.pack_start(vbox, False, False, 0)
107         radio = gtk.RadioButton(None, "WRAP__NONE")
108         vbox.pack_start(radio, False, True, 0)
109         radio.connect("toggled", self.new_wrap_mode, textview, gtk. ←
    WRAP_NONE)
110         radio.set_active(True)
111         radio.show()
112         radio = gtk.RadioButton(radio, "WRAP__CHAR")
113         vbox.pack_start(radio, False, True, 0)
114         radio.connect("toggled", self.new_wrap_mode, textview, gtk. ←
    WRAP_CHAR)
115         radio.show()
116         radio = gtk.RadioButton(radio, "WRAP__WORD")
117         vbox.pack_start(radio, False, True, 0)
118         radio.connect("toggled", self.new_wrap_mode, textview, gtk. ←
    WRAP_WORD)
119         radio.show()
120 
121         # radio buttons to specify justification
122         vbox = gtk.VBox()
123         vbox.show()
124         hbox.pack_start(vbox, False, False, 0)
125         radio = gtk.RadioButton(None, "JUSTIFY__LEFT")
126         vbox.pack_start(radio, False, True, 0)
127         radio.connect("toggled", self.new_justification, textview,
128                       gtk.JUSTIFY_LEFT)
129         radio.set_active(True)
130         radio.show()
```

```
131         radio = gtk.RadioButton(radio, "JUSTIFY__RIGHT")
132         vbox.pack_start(radio, False, True, 0)
133         radio.connect("toggled", self.new_justification, textview,
134                       gtk.JUSTIFY_RIGHT)
135         radio.show()
136         radio = gtk.RadioButton(radio, "JUSTIFY__CENTER")
137         vbox.pack_start(radio, False, True, 0)
138         radio.connect("toggled", self.new_justification, textview,
139                       gtk.JUSTIFY_CENTER)
140         radio.show()
141
142         separator = gtk.HSeparator()
143         box1.pack_start(separator, False, True, 0)
144         separator.show()
145
146         box2 = gtk.VBox(False, 10)
147         box2.set_border_width(10)
148         box1.pack_start(box2, False, True, 0)
149         box2.show()
150
151         button = gtk.Button("close")
152         button.connect("clicked", self.close_application)
153         box2.pack_start(button, True, True, 0)
154         button.set_flags(gtk.CAN_DEFAULT)
155         button.grab_default()
156         button.show()
157         window.show()
158
159 def main():
160     gtk.main()
161     return 0
162
163 if __name__ == "__main__":
164     TextViewExample()
165     main()
```

Lines 10-34 define the callbacks for the radio and check buttons used to change the default attributes of the `TextView`. Lines 55-63 create a `ScrolledWindow` to contain the `TextView`. The `ScrolledWindow` is packed into a `VBox` with the check and radio buttons created in lines 72-140. The `TextBuffer` associated with the `TextView` is loaded with the contents of the source file in lines 64-70.

13.3 Text Buffers

A `TextBuffer` is the core component of the PyGTK text editing system. It contains the text, the `TextTags` in a `TextTagTable` and the `TextMarks` which together describe how the text is to be displayed and allow a user to interactively modify the text and text display. As noted in the previous section a `TextBuffer` is associated with one or more `TextViews` which display the `TextBuffer` contents.

A `TextBuffer` can be created automatically when a `TextView` is created or it can be created with the function:

```
textbuffer = TextBuffer(table=None)
```

where *table* is a `TextTagTable`. If *table* is not specified (or is `None`) a `TextTagTable` will be created for the `TextBuffer`.

There are a large number of methods that can be used to:

- insert and remove text from a buffer
- create, delete and manipulate marks
- manipulate the cursor and the selection
- create, apply and remove tags

- specify and manipulate TextIters
- get status information

13.3.1 TextBuffer Status Information

You can retrieve the number of lines in a *textbuffer* by using the method:

```
line_count = textbuffer.get_line_count()
```

Likewise you can get the number of characters in the *textbuffer* using:

```
char_count = textbuffer.get_char_count()
```

When the *textbuffer* contents are changed the modified flag in the textbuffer is set. The status of the modified flag can be retrieved using the method:

```
modified = textbuffer.get_modified()
```

If the program saves the contents of the textbuffer the following method can be used to reset the modified flag:

```
textbuffer.set_modified(setting)
```

13.3.2 Creating TextIters

A TextIter is used to specify a location within a TextBuffer between two characters. TextBuffer methods that manipulate text use TextIters to specify where the method is to be applied. TextIters have a large number of methods that will be described in the >TextIters section.

The basic TextBuffer methods used to create TextIters are:

```
iter = textbuffer.get_iter_at_offset(char_offset)

iter = textbuffer_get_iter_at_line(line_number)

iter = textbuffer.get_iter_at_line_offset(line_number, line_offset)

iter = textbuffer.get_iter_at_mark(mark)
```

get_iter_at_offset() creates an iter that is just after *char_offset* chars from the start of the textbuffer.

get_iter_at_line() creates an iter that is just before the first character in *line_number*.

get_iter_at_line_offset() creates an iter that is just after the *line_offset* character in *line_number*.

get_iter_at_mark() creates an iter that is at the same position as the given *mark*.

The following methods create one or more TextIters at specific buffer locations:

```
startiter = textbuffer.get_start_iter()

enditer = textbuffer_get_end_iter()

startiter, enditer = textbuffer.get_bounds()

start, end = textbuffer.get_selection_bounds()
```

get_start_iter() creates an iter that is just before the first character in the textbuffer.

get_end_iter() creates an iter that is just after the last character in the textbuffer.

get_bounds() creates a tuple of two iters that are just before the first character and just after the last character in the textbuffer respectively.

get_selection_bounds() creates a tuple of two iters that have the same location as the *insert* and *selection_bound* marks in the textbuffer.

13.3.3 Text Insertion, Retrieval and Deletion

The text in a `TextBuffer` can be set using the method:

```
textbuffer.set_text(text)
```

This method replaces the current contents of textbuffer with *text*.

The most general method to insert characters in a textbuffer is:

```
textbuffer.insert(iter, text)
```

which inserts *text* at the textbuffer location specified by *iter*.

If you want to simulate the insertion of text by an interactive user use the method:

```
result = textbuffer.insert_interactive(iter, text, default_editable)
```

which inserts *text* in the textbuffer at the location specified by *iter* but only if the location is editable (i.e. does not have a tag that specifies the text is non-editable) and the *default_editable* value is `TRUE`. The result indicates whether the text was inserted.

default_editable indicates the editability of text that doesn't have a tag affecting editability; *default_editable* is usually determined by a call to the `TextView` get_editable() method.

Other methods that insert text are:

```
textbuffer.insert_at_cursor(text)

result = textbuffer.insert_at_cursor_interactive(text, default_editable)

textbuffer.insert_range(iter, start, end)

result = textbuffer.insert_range_interactive(iter, start, end, default_editable ↩
    )
```

insert_at_cursor() is a convenience method that inserts text at the current cursor (*insert*) location.

insert_range() copies text, pixbufs and tags between *start* and *end* from a `TextBuffer` (if different from textbuffer the tag table must be the same) and inserts the copy into textbuffer at *iter*'s location.

The interactive versions of these methods operate the same way except they will only insert if the location is editable.

Finally, text can be inserted and have tags applied at the same time using the methods:

```
textbuffer.insert_with_tags(iter, text, tag1, tag2, ...)

textbuffer.insert_with_tags_by_name(iter, text, tagname1, tagname2, ...)
```

insert_with_tags() inserts the *text* in the textbuffer at the location specified by *iter* and applies the given tags.

insert_with_tags_by_name() does that same thing but allows you to specify the tags using the tag name.

The text in a textbuffer can be deleted by using the methods:

```
textbuffer.delete(start, end)

result = textbuffer.delete_interactive(start, end, default_editable)
```

delete() removes the text between the *start* and *end* `TextIter` locations in textbuffer.

delete_interactive() removes all the editable (as determined by the applicable text tags and the *default_editable* argument) text between *start* and *end*.

You can retrieve a copy of the text from a textbuffer by using the methods:

```
text = textbuffer.get_text(start, end, include_hidden_chars=TRUE)

text = textbuffer.get_slice(start, end, include_hidden_chars=TRUE)
```

get_text() returns a copy of the *text* in textbuffer between *start* and *end*; undisplayed text is excluded if *include_hidden_chars* is `FALSE`. Characters which represent embedded images or widgets are excluded.

get_slice() is the same as get_text() except that the returned *text* includes a 0xFFFC character for each embedded image or widget.

13.3.4 TextMarks

`TextMarks` are similar to `TextIters` in that they specify a location in a `TextBuffer` between two characters. However, `TextMarks` maintain their location information across buffer modifications. The `TextMark` methods will be described in the `TextMarks` section.

A textbuffer contains two built-in marks: the *`insert`* (cursor) mark and the *`selection_bound`* mark. The *`insert`* mark is the default location for the insertion of text and the *`selection_bound`* mark combines with the *`insert`* mark to define a selection range.

The built-in marks can be retrieved by using the methods:

```
insertmark = textbuffer.get_insert()

selection_boundmark = textbuffer.get_selection_bound()
```

The *`insert`* and *`selection_bound`* marks can be placed simultaneously at a location by using the method:

```
textbuffer.place_cursor(where)
```

`where` is a textiter specifying the location. The place_cursor() method is needed to avoid temporarily creating a selection if the marks were moved individually.

`TextMarks` are created by using the method:

```
mark = textbuffer.create_mark(mark_name, where, left_gravity=FALSE)
```

where *`mark_name`* is the name assigned to the created mark (can be `None` to create an anonymous mark), *`where`* is the textiter specifying the location of the mark in textbuffer and *`left_gravity`* indicates where the mark will be located after text is inserted at the mark (left if `TRUE` or right if `FALSE`).

A mark can be moved in the textbuffer by using the methods:

```
textbuffer.move_mark(mark, where)

textbuffer.move_mark_by_name(name, where)
```

`mark` specifies the mark to be moved. *`name`* specifies the name of the mark to be moved. *`where`* is a textiter specifying the new location.

A mark can be deleted from a textbuffer by using the methods:

```
textbuffer.delete_mark(mark)

textbuffer.delete_mark_by_name(name)
```

A mark can be retrieved by name using the method:

```
mark = textbuffer.get_mark(name)
```

13.3.5 Creating and Applying TextTags

`TextTags` contain one or more attributes (e.g. foreground and background colors, font, editability) that can be applied to one or more ranges of text in a `TextBuffer`. The attributes that can be specified by `TextTag` properties will be described in Section 13.6.1.

A `TextTag` can be created with attributes and installed in the `TextTagTable` of a `TextBuffer` by using the convenience method:

```
tag = textbuffer.create_tag(name=None, attr1=val1, attr2=val2, ...)
```

where *`name`* is a string specifying the name of the tag or `None` if the tag is an anonymous tag and the keyword-value pairs specify the attributes that the tag will have. See the `TextTag`> section for information on what attributes can be set by the `TextTag` properties.

A tag can be applied to a range of text in a textbuffer by using the methods:

```
textbuffer.apply_tag(tag, start, end)

textbuffer.apply_tag_by_name(name, start, end)
```

tag is the tag to be applied to the text. *name* is the name of the tag to be applied. *start* and *end* are textiters that specify the range of text that the *tag* is to be applied to.

A tag can be removed from a range of text by using the methods:

```
textbuffer.remove_tag(tag, start, end)

textbuffer.remove_tag_by_name(name, start, end)
```

All tags for a range of text can be removed by using the method:

```
textbuffer.remove_all_tags(start, end)
```

13.3.6 Inserting Images and Widgets

In addition to text a `TextBuffer` can contain pixbuf images and an anchor location for widgets. A widget can be added to a `TextView` at an anchor location. A different widget can be added in each `TextView` which displays a buffer with an anchor.

A pixbuf can be inserted by using the method:

```
textbuffer.insert_pixbuf(iter, pixbuf)
```

where *iter* specifies the location in the *textbuffer* to insert the *pixbuf*. The image will be counted as one character and will be represented in a get_slice() return (but left out of a get_text() return) as the Unicode character "0xFFFC".

A GTK+ widget can be inserted in a `TextView` at a buffer location specified with a `TextChildAnchor`. The `TextChildAnchor` will be counted as one character and represented as "0xFFFC" similar to a pixbuf.

The `TextChildAnchor` can be created and inserted in the buffer by using the convenience method:

```
anchor = text_buffer.create_child_anchor(iter)
```

where *iter* is the location for the child_anchor.

A `TextChildAnchor` can also be created and inserted in two operations as:

```
anchor = gtk.TextChildAnchor()

text_buffer.insert_child_anchor(iter, anchor)
```

Then the widget can be added to the `TextView` at an anchor location using the method:

```
text_view.add_child_at_anchor(child, anchor)
```

The list of widgets at a particular buffer anchor can be retrieved using the method:

```
widget_list = anchor.get_widgets()
```

A widget can also be added to a `TextView` using the method:

```
text_view.add_child_in_window(child, which_window, xpos, ypos)
```

where the *child* widget is placed in *which_window* at the location specified by *xpos* and *ypos*. *which_window* indicates in which of the windows that make up the `TextView` the widget is to be placed:

```
gtk.TEXT_WINDOW_TOP
gtk.TEXT_WINDOW_BOTTOM
gtk.TEXT_WINDOW_LEFT
gtk.TEXT_WINDOW_RIGHT
gtk.TEXT_WINDOW_TEXT
gtk.TEXT_WINDOW_WIDGET
```

13.4 Text Iters

TextIters represent a position between two characters in a TextBuffer. TextIters are usually created by using a TextBuffer method. TextIters are invalidated when the number of characters in a TextBuffer is changed (except for the TextIter that is used for the insertion or deletion). Inserting or deleting pixbufs or anchors also counts as a TextIter invalidating change.

There are a large number of methods associated with a TextIter object. They are grouped together in the following sections by similar function.

13.4.1 TextIter Attributes

The TextBuffer that contains the TextIter can be retrieved using the method:

```
buffer = iter.get_buffer()
```

The following methods can be used to get the location of the TextIter in the TextBuffer:

```
offset = iter.get_offset()      # returns offset in buffer of iter

line_number = iter.get_line()     # returns number of line at iter

line_offset = iter.get_line_offset()    # returns iter offset in line

numchars = iter.get_chars_in_line()   # returns number of chars in line
```

13.4.2 Text Attributes at a TextIter

The PangoLanguage used at a given iter location in the TextBuffer is obtained by calling the method:

```
language = iter.get_language()
```

The more general method used to get the text attributes at a TextIter's location is:

```
result = iter.get_attributes(values)
```

where *result* indicates whether the given *values* (TextAttributes object) were modified. The given *values* are obtained by using the TextView method:

```
values = textview.get_default_attributes()
```

The following attributes are accessible from a TextAttributes object (not implemented in PyGTK <= 1.99.15):

bg_color	background color
fg_color	foreground color
bg_stipple	background stipple bitmap
fg_stipple	foreground stipple bitmap
rise	offset of text above baseline
underline	style of underline
strikethrough	whether text is strikethrough
draw_bg	TRUE if some tags affect the drawing of the background
justification	style of justification
direction	which direction the text runs
font	PangoFontDescription in use
font_scale	scale of the font in use
left_margin	location of left margin
right_margin	location of right margin
pixels_above_lines	pixels spacing above a line
pixels_below_lines	pixel spacing below a line
pixels_inside_wrap	pixel spacing between wrapped lines
tabs	PangoTabArray in use
wrap_mode	mode of wrap in use

language	PangoLanguage in use
invisible	whether text is invisible (not implemented in GTK+ 2.0)
bg_full_height	whether background is fit to full line height
editable	whether the text is editable
realized	text is realized
pad1	
pad2	
pad3	
pad4	

13.4.3 Copying a TextIter

A `TextIter` can be duplicated using the method:

```
iter_copy = iter.copy()
```

13.4.4 Retrieving Text and Objects

Various amounts of text and `TextBuffer` objects can be retrieved from a `TextBuffer` using the following methods:

```
char = iter.get_char()    # returns char or 0 if at end of buffer

text = start.get_slice(end)   # returns the text between start and end iters

text = start.get_text(end)    # returns the text between start and end iters

pixbuf = iter.get_pixbuf()    # returns the pixbuf at the location (or None)

anchor = iter.get_child_anchor()  # returns the child anchor (or None)

mark_list = iter.get_marks()    # returns a list of marks

tag_list = iter.get_toggled_tags()  # returns a list of tags that are toggled ↩
    on or off

tag_list = iter.get_tags()    # returns a prioritized list of tags
```

13.4.5 Checking Conditions at a TextIter

Tag conditions at the `TextIter` location can be checked using the following methods:

```
result = iter.begins_tag(tag=None)  # TRUE if tag is toggled on at iter

result = iter.ends_tag(tag=None)  # TRUE if tag is toggled off at iter

result = iter.toggles_tag(tag=None) # TRUE if tag is toggled on or off at iter

result = iter.has_tag(tag)    # TRUE if tag is active at iter
```

These methods return `TRUE` if the given *tag* satisfies the condition at *iter*. If the *tag* is `None` for the first three methods then the result is `TRUE` if any tag satisfies the condition at *iter*.

The following methods indicate whether the text at the `TextIter` location is editable or allows text insertion:

```
result = iter.editable()

result = iter.can_insert(default_editability)
```

The editable() method indicates whether the *iter* is in an editable range of text while the can_insert() method indicates whether text can be inserted at *iter* considering the default editability of the `TextView`, `TextBuffer` and applicable tags. The *default_editability* is usually determined by calling the method:

```
default_editability = textview.get_editable()
```

The equivalence of two TextIters can be determined with the method:

```
are_equal = lhs.equal(rhs)
```

Two TextIters can be compared with the method:

```
result = lhs.compare(rhs)
```

result will be: -1 if *lhs* is less than *rhs*; 0 if *lhs* equals *rhs*; and, 1 if *lhs* is greater than *rhs*.

To determine whether a TextIter is located between two given TextIters use the method:

```
result = iter.in_range(start, end)
```

result is TRUE if *iter* is between *start* and *end*. Note: *start* and *end* must be in ascending order. This can be guaranteed using the method:

```
first.order(second)
```

which will reorder the TextIter offsets so that *first* is before *second*.

13.4.6 Checking Location in Text

The location of a TextIter with respect to the text in a TextBuffer can be determined by the following methods:

```
result = iter.starts_word()

result = iter.ends_word()

result = iter.inside_word()

result = iter.starts_sentence()

result = iter.ends_sentence()

result = iter.inside_sentence()

result = starts_line()

result = iter.ends_line()
```

result returns TRUE if the TextIter is at the given text location. These methods are somewhat self-explanatory. The definition of the text components and their boundaries is determined by the language used at the TextIter. Note that a line is a collection of sentences similar to a paragraph.

The following methods can be used to determine if a TextIter is at the start or end of the TextBuffer:

```
result = iter.is_start()

result = iter.is_end()
```

result is TRUE if the TextIter is at the start or end of the TextBuffer.

Since a TextBuffer may contain multiple characters which are effectively viewed as one cursor position (e.g. carriage return-linefeed combination or letter with an accent mark) it's possible that a TextIter could be in a location which is not a cursor position. The following method indicates whether a TextIter is at a cursor position:

```
result = iter.is_cursor_position()
```

13.4.7 Moving Through Text

TextIters can be moved through a TextBuffer in various text unit strides. The definition of the text units is set by the PangoLanguage in use at the TextIter location. The basic methods are:

```
result = iter.forward_char()      # forward by one character

result = iter.backward_char()     # backward by one character

result = iter.forward_word_end()    # forward to the end of the word

result = iter.backward_word_start()   # backward to the start of the word

result = iter.forward_sentence_end()    # forward to the end of the sentence

result = iter.backward_sentence_start() # backward to the start of the sentence

result = iter.forward_line()      # forward to the start of the next line

result = iter.backward_line()     # backward to the start of the previous line

result = iter.forward_to_line_end()   # forward to the end of the line

result = iter.forward_cursor_position()   # forward by one cursor position

result = iter.backward_cursor_position()  # forward by one cursor position
```

result is TRUE if the TextIter was moved and FALSE if the TextIter is at the start or end of the TextBuffer.

All of the above methods (except forward_to_line_end()) have corresponding methods that take a count (that can be positive or negative) to move the TextIter in multiple text unit strides:

```
result = iter.forward_chars(count)

result = iter.backward_chars(count)

result = iter.forward_word_ends(count)

result = iter.backward_word_starts(count)

result = iter.forward_sentence_ends(count)

result = iter.backward_sentence_starts(count)

result = iter.forward_lines(count)

result = iter.backward_lines(count)

result = iter.forward_cursor_positions(count)

result = iter.backward_cursor_positions(count)
```

13.4.8 Moving to a Specific Location

A TextIter can be moved to a specific location in the TextBuffer using the following methods:

```
iter.set_offset(char_offset)      # move to given character offset

iter.set_line(line_number)      # move to start of given line

iter.set_line_offset(char_on_line)    # move to given character offset in ←
    current line

iter.forward_to_end()       # move to end of the buffer
```

In addition, a TextIter can be moved to a location where a tag is toggled on or off by using the methods:

```
result = iter.forward_to_tag_toggle(tag)

result = iter.backward_to_tag_taoggle(tag)
```

result is TRUE if the TextIter was moved to a new location where *tag* is toggled. If *tag* is None then the TextIter will be moved to the next location where any tag is toggled.

13.4.9 Searching in Text

A search for a string in a TextBuffer is done using the methods:

```
match_start, match_end = iter.forward_search(str, flags, limit=None)

match_start, match_end = iter.backward_search(str, flags, limit=None)
```

The *return* value is a tuple containing TextIters that indicate the location of the first character of the match and the first character after the match. *str* is the character string to be located. *flags* modifies the conditions of the search; *flag* values can be:

```
gtk.TEXT_SEARCH_VISIBLE_ONLY    # invisible characters are ignored

gtk.TEXT_SEARCH_TEXT_ONLY   # pixbufs and child anchors are ignored
```

limit is an optional TextIter that bounds the search range.

13.5 Text Marks

A TextMark indicates a location in a TextBuffer between two characters that is preserved across buffer modifications. TextMarks are created, moved and deleted using the TextBuffer methods as described in the TextBuffer section.

A TextBuffer has two built-in TextMarks named: *insert* and *selection_bound* which refer to the insertion point and the boundary of the selection (these may refer to the same location).

The name of a TextMark can be retrieved using the method:

```
name = textmark.get_name()
```

By default marks other than insert are not visible (displayed as a vertical bar). The visibility of a mark can be set and retrieved using the methods:

```
setting = textmark.get_visible()

textmark.set_visible(setting)
```

where *setting* is TRUE if the mark is visible.

The TextBuffer that contains a TextMark can be obtained using the method:

```
buffer = textmark.get_buffer()
```

You can determine whether a TextMark has been deleted using the method:

```
setting = textmark.get_deleted()
```

The left gravity of a TextMark can be retrieved using the method:

```
setting = textmark.get_left_gravity()
```

The left gravity of a TextMark indicates where the mark will end up after an insertion. If left gravity is TRUE the mark will be to the left of the insertion; if FALSE, to the right of the insertion.

13.6 Text Tags and Tag Tables

TextTags specify attributes that can be applied to a range of text in a TextBuffer. Each `TextBuffer` has a `TextTagTable` that contains the `TextTag`s that can be applied within the `TextBuffer`. `TextTagTable`s can be used with more than one `TextBuffer` to provide consistent text styles.

13.6.1 Text Tags

`TextTag`s can be named or anonymous. A `TextTag` is created using the function:

```
tag = gtk.TextTag(name=None)
```

If *`name`* is not specified or is `None` the *`tag`* will be anonymous. `TextTag`s can also be created using the `TextBuffer` convenience method create_tag() which also allows you specify the *`tag`* attributes and adds the *`tag`* to the buffer's tag table (see Section 13.3).

The attributes that can be contained in a `TextTag` are:

name	Read / Write	Name of the text tag. `None` if anonymous.
background	Write	Background color as a string
foreground	Write	Foreground color as a string
background-gdk	Read / Write	Background color as a GdkColor
foreground-gdk	Read / Write	Foreground color as a GdkColor
background-stipple	Read / Write	Bitmap to use as a mask when drawing the text background
foreground-stipple	Read / Write	Bitmap to use as a mask when drawing the text foreground
font	Read / Write	Font description as a string, e.g. "Sans Italic 12"
font-desc	Read / Write	Font description as a PangoFontDescription
family	Read / Write	Name of the font family, e.g. Sans, Helvetica, Times, Monospace
style	Read / Write	Font style as a PangoStyle, e.g. pango.STYLE_ITALIC.
variant	Read / Write	Font variant as a PangoVariant, e.g. pango.VARIANT_SMALL_CAPS.
weight	Read / Write	Font weight as an integer, see predefined values in PangoWeight; for example, pango.WEIGHT_BOLD.
stretch	Read / Write	Font stretch as a PangoStretch, e.g. pango.STRETCH_CONDENSED.
size	Read / Write	Font size in Pango units.
size-points	Read / Write	Font size in points
scale	Read / Write	Font size as a scale factor relative to the default font size. This properly adapts to theme changes etc. so is recommended. Pango predefines some scales such as pango.SCALE_X_LARGE.
pixels-above-lines	Read / Write	Pixels of blank space above paragraphs
pixels-below-lines	Read / Write	Pixels of blank space below paragraphs
pixels-inside-wrap	Read / Write	Pixels of blank space between wrapped lines in a paragraph
editable	Read / Write	Whether the text can be modified by the user
wrap-mode	Read / Write	Whether to wrap lines never, at word boundaries, or at character boundaries
justification	Read / Write	Left, right, or center justification
direction	Read / Write	Text direction, e.g. right-to-left or left-to-right
left-margin	Read / Write	Width of the left margin in pixels
indent	Read / Write	Amount to indent the paragraph, in pixels
strikethrough	Read / Write	Whether to strike through the text
right-margin	Read / Write	Width of the right margin in pixels
underline	Read / Write	Style of underline for this text
rise	Read / Write	Offset of text above the baseline (below the baseline if rise is negative) in pixels
background-full-height	Read / Write	Whether the background color fills the entire line height or only the height of the tagged characters
language	Read / Write	The language this text is in, as an ISO code. Pango can use this as a hint when rendering the text. If you don't understand this parameter, you probably don't need it.
tabs	Read / Write	Custom tabs for this text

invisible	Read / Write	Whether this text is hidden. Not implemented in GTK+ 2.0

The attributes can be set by using the method:

```
tag.set_property(name, value)
```

Where *name* is a string containing the name of the property and *value* is what the property should be set to.

Likewise the attribute value can be retrieved with the method:

```
value = tag.get_property(name)
```

Since the tag does not have a value set for every attribute there are a set of boolean properties that indicate whether the attribute has been set in the tag:

background-set	Read / Write
foreground-set	Read / Write
background-stipple-set	Read / Write
foreground-stipple-set	Read / Write
family-set	Read / Write
style-set	Read / Write
variant-set	Read / Write
weight-set	Read / Write
stretch-set	Read / Write
size-set	Read / Write
scale-set	Read / Write
pixels-above-lines-set	Read / Write
pixels-below-lines-set	Read / Write
pixels-inside-wrap-set	Read / Write
editable-set	Read / Write
wrap-mode-set	Read / Write
justification-set	Read / Write
direction-set	Read / Write
left-margin-set	Read / Write
indent-set	Read / Write
strikethrough-set	Read / Write
right-margin-set	Read / Write
underline-set	Read / Write
rise-set	Read / Write
background-full-height-set	Read / Write
language-set	Read / Write
tabs-set	Read / Write
invisible-set	Read / Write

Therefore to obtain the attribute from a tag, you have to first check whether the attribute has been set in the tag. For example to get a valid justification attribute you may have to do something like:

```
if tag.get_property("justification-set"):
justification = tag.get_property("justification")
```

The priority of a tag is by default the order in which they are added to the `TextTagTable`. The higher priority tag takes precedence if multiple tags try to set the same attribute for a range of text. The priority can be obtained and set with the methods:

```
priority = tag.get_priority()

tag.set_priority(priority)
```

The priority of a tag must be between 0 and one less than the `TextTagTable` size.

13.6.2 Text Tag Tables

A `TextTagTable` will be created by default when a `TextBuffer` is created. A `TextTagTable` can also be created with the function:

```
table = TextTagTable()
```

A `TextTag` can be added to a `TextTagTable` using the method:

```
table.add(tag)
```

The *tag* must not be in the *table* and must not have the same name as another tag in the table.

You can find a `TextTag` in a `TextTagTable` using the method:

```
tag = table.lookup(name)
```

The method returns the *tag* in the table with the given *name* or `None` if no tag has that *name*.

A `TextTag` can be removed from a `TextTagTable` with the method:

```
table.remove(tag)
```

The size of the `TextTagTable` can be obtained with the method:

```
size = table.get_size()
```

13.7 A TextView Example

The testtext.py example program (derived from the `testtext.c` program included in the GTK+ 2.0.x distribution) demonstrates the use of the `TextView` widget and its associated objects: `TextBuffers`, `TextIters`, `TextMarks`, `TextTags`, `TextTagTables`. Figure 13.2 illustrates its operation:

Figure 13.2 TextView Example

The testtext.py program defines a number of classes in addition to the application class `TestText`:

- `Buffer` class, lines 99-496, is subclassed from the `gtk.TextBuffer` type. It provides the editing buffer capabilities used by the `View` objects.
- `View` class, lines 498-1126, is subclassed from the `gtk.Window` type and wraps a `gtk.TextView` object that uses a `Buffer` object instead of a `gtk.TextBuffer` object. It provides a window and the visual display of the contents of a `Buffer` object as well as a menubar.
- `FileSel` class, lines 73-97, is subclassed from the `gtk.FileSelection` type to provide selection of filenames for the `Buffer` contents.
- `Stack` class to provide simple stack objects.

The color cycle display is implemented by using text tags applied to a section of text in a buffer. Lines 109-115 (in the __init__() method) create these tags and lines 763-784 (do_apply_colors() method) apply the color tags to a section of text two characters at a time. Lines 202-239 provide the methods (color_cycle_timeout(), set_colors() and cycle_colors()) that produce the color cycle display when enabled. Color cycling is enabled by setting (line 220) the *foreground_gdk* property of the individual

`color_tags` (which also sets the `foreground_set` property). Color cycling is disabled by setting the `foreground_set` property to `FALSE` (line 222). The colors are periodically changed by shifting the `start_hue` (line 237)

A new `Buffer` is filled with example content when the Test → Example menu item is selected (the fill_example_buffer() method in lines 302-372). The example buffer contains text of various colors, styles and languages and pixbufs. The init_tags() method (lines 260-300) sets up a variety of `TextTag`s for use with the example text. The event signal of these tags is connected to the tag_event_handler() method (lines 241-256) to illustrate button and motion event capture.

The `TextView` wrap mode is set to WRAP_WORD (line 580) and the `TextView` border windows are displayed by setting their sizes in lines 587-588 and line 596-597. The left and right border windows are used to display line numbers and the top and bottom border windows display the tab locations when custom tabs are set. The border windows are updated when an "expose-event" signal is received by the `TextView` (lines 590 and 599). The line_numbers_expose() method (lines 1079-1116) determines whether the left or right border window has an expose event and if so calculates the size of the expose area. Then the location of the line start and the line number for each line in the exposed area is calculated in the get_lines() method (lines 1057-1077). The line numbers are then drawn in the border window at the location (transformed by line 1109).

The custom tab locations are displayed in the top and bottom border windows in a similar fashion (lines 1013-1055). They are displayed only when the cursor is moved inside a range of text that has the custom tab attribute set. This is detected by handling the "mark-set" signal in the cursor_set_handler() method (lines 999-1011) and invalidating the top and bottom border windows if the mark set is the `insert` mark.

Movable widgets are added to a `View` with the do_add_children() method (lines 892-899) which calls the add_movable_children() method (lines 874-890). The children are `gtk.Label`s that can be dragged around inside the various windows that are part of a `TextView` widget.

Likewise, widgets are added to the `TextView` windows of a `View` and the `Buffer` by using the do_add_focus_children() method (lines 901-949).

Chapter 14

Tree View Widget

The `TreeView` widget displays lists and trees displaying multiple columns. It replaces the previous set of List, `CList`, `Tree` and `CTree` widgets with a much more powerful and flexible set of objects that use the Model-View-Controller (MVC) principle to provide the following features:

- two pre-defined models: one for lists and one for trees
- multiple views of the same model are automatically updated when the model changes
- selective display of the model data
- use of model data to customize the TreeView display on a row-by-row basis
- pre-defined data rendering objects for displaying text, images and boolean data
- stackable models for providing sorted and filtered views of the underlying model data
- reorderable and resizeable columns
- automatic sort by clicking column headers
- drag and drop support
- support for custom models entirely written in Python
- support for custom cell renderers entirely written in Python

Of course, all this capability comes at the price of a significantly more complex set of objects and interfaces that appear overwhelming at first. In the rest of this chapter we'll explore the TreeView objects and interfaces to reach an understanding of common usage. The more esoteric aspects, you'll have to explore on your own.

We'll start with a quick overview tour of the objects and interfaces and then dive into the `TreeModel` interface and the predefined `ListStore` and `TreeStore` classes.

14.1 Overview

A `TreeView` widget is the user interface object that displays the data stored in an object that implements the `TreeModel` interface. Two base tree model classes are provided in PyGTK 2.0:

- the `TreeStore` that provides hierarchical data storage organized as tree rows with columnar data. Each tree row can have zero or more child rows. All rows must have the same number of columns.
- the `ListStore` that provides tabular data storage organized in rows and columns similar to a table in a relational database. The `ListStore` is really a simplified version of a `TreeStore` where the rows have no children. It has been created to provide a simpler (and presumably more efficient) interface to this common data model. And,

The two additional tree models stack on top of (or interpose on) the base models:

- the `TreeModelSort` that provides a model where the data of the underlying tree model is maintained in a sorted order. And,
- the `TreeModelFilter` that provides a model containing a subset of the data in the underlying model. Note this model is available only in PyGTK 2.4 and above.

A `TreeView` displays all of the rows of a `TreeModel` but may display only some of the columns. Also the columns may be presented in a different order than the `TreeModel` stores them.

The `TreeView` uses `TreeViewColumn` objects to organize the display of the columnar data. Each `TreeViewColumn` displays one column with an optional header that may contain the data from several `TreeModel` columns. The individual `TreeViewColumn`s are packed (similar to `HBox` containers) with `CellRenderer` objects to render the display of the associated data from a `TreeModel` row and column location. There are three predefined `CellRenderer` classes:

- the `CellRendererPixbuf` that renders a pixbuf image into the cells of a `TreeViewColumn`.
- the `CellRendererText` that renders a string into the cells of a `TreeViewColumn`. It will convert the column data to a string format if needed i.e. if displaying a model column containing float data, the `CellRendererText` will convert it to a string before rendering it.
- the `CellRendererToggle` that renders a boolean value as a toggle button into the cells of a `TreeViewColumn`.

A `TreeViewColumn` can contain several `CellRenderer` objects to provide a column that, for example, may have an image and text packed together.

Finally, the `TreeIter`, `TreeRowReference` and `TreeSelection` objects provide a transient pointer to a row in a `TreeModel`, a persistent pointer to a row in a `TreeModel` and an object managing the selections in a `TreeView`.

A `TreeView` display is composed using the following general operations not necessarily in this order:

- A tree model object is created usually a `ListStore` or `TreeStore` with one or more columns of a specified data type.
- The tree model may be populated with one or more rows of data.
- A `TreeView` widget is created and associated with the tree model.
- One or more `TreeViewColumn`s are created and inserted in the `TreeView`. Each of these will present a single display column.
- For each `TreeViewColumn` one or more `CellRenderer`s are created and added to the `TreeViewColumn`.
- The attributes of each `CellRenderer` are set to indicate from which column of the tree model to retrieve the attribute data. for example the text to be rendered. This allows the `CellRenderer` to render each column in a row differently.
- The `TreeView` is inserted and displayed in a `Window` or `ScrolledWindow`.
- The data in the tree model is manipulated programmatically in response to user actions. The `TreeView` will automatically track the changes.

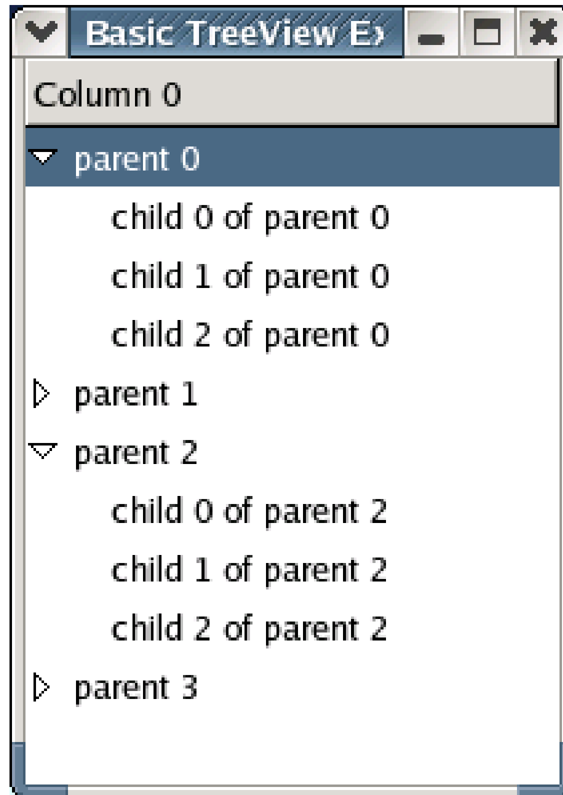
The example program basictreeview.py illustrates the creation and display of a simple `TreeView`:

```
1   #!/usr/bin/env python
2
3   # example basictreeview.py
4
5   import pygtk
6   pygtk.require('2.0')
7   import gtk
8
9   class BasicTreeViewExample:
10
11      # close the window and quit
```

```
12      def delete_event(self, widget, event, data=None):
13          gtk.main_quit()
14          return False
15
16      def __init__(self):
17          # Create a new window
18          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
19
20          self.window.set_title("Basic TreeView Example")
21
22          self.window.set_size_request(200, 200)
23
24          self.window.connect("delete_event", self.delete_event)
25
26          # create a TreeStore with one string column to use as the model
27          self.treestore = gtk.TreeStore(str)
28
29          # we'll add some data now - 4 rows with 3 child rows each
30          for parent in range(4):
31              piter = self.treestore.append(None, ['parent %i' % parent])
32              for child in range(3):
33                  self.treestore.append(piter, ['child %i of parent %i' %
34                                                (child, parent)])
35
36          # create the TreeView using treestore
37          self.treeview = gtk.TreeView(self.treestore)
38
39          # create the TreeViewColumn to display the data
40          self.tvcolumn = gtk.TreeViewColumn('Column 0')
41
42          # add tvcolumn to treeview
43          self.treeview.append_column(self.tvcolumn)
44
45          # create a CellRendererText to render the data
46          self.cell = gtk.CellRendererText()
47
48          # add the cell to the tvcolumn and allow it to expand
49          self.tvcolumn.pack_start(self.cell, True)
50
51          # set the cell "text" attribute to column 0 - retrieve text
52          # from that column in treestore
53          self.tvcolumn.add_attribute(self.cell, 'text', 0)
54
55          # make it searchable
56          self.treeview.set_search_column(0)
57
58          # Allow sorting on the column
59          self.tvcolumn.set_sort_column_id(0)
60
61          # Allow drag and drop reordering of rows
62          self.treeview.set_reorderable(True)
63
64          self.window.add(self.treeview)
65
66          self.window.show_all()
67
68  def main():
69      gtk.main()
70
71  if __name__ == "__main__":
72      tvexample = BasicTreeViewExample()
73      main()
```

In real programs the `TreeStore` would likely be populated with data after the `TreeView` is displayed due to some user action. We'll look at the details of the `TreeView` interfaces in more detail in the sections to come. Figure 14.1 shows the window created by the basictreeview.py program after a couple of parent rows have been expanded.

Figure 14.1 Basic TreeView Example Program



Next let's examine the `TreeModel` interface and the models that implement it.

14.2 The TreeModel Interface and Data Stores

14.2.1 Introduction

The `TreeModel` interface is implemented by all the `TreeModel` subclasses and provides methods to:

- retrieve the characteristics of the data store such as the number of columns and the type of data in a column.
- retrieve a `TreeIter` (a transient reference) that points at a row in the model
- retrieve information about a node (or row) such as the number of its child nodes, a list of its child nodes, the contents of its columns and a pointer to its parent node
- provide notification of `TreeModel` data changes

14.2.2 Creating TreeStore and ListStore Objects

The base data store classes: `ListStore` and `TreeStore` provide the means to define and manage the rows and columns of data in the tree model. The constructors of both these objects require the column types to be specified as any of:

- Python types such as the built-in types: int, str, long, float and object
- PyGTK types such as `Button`, `VBox`, `gdk.Rectangle`, `gdk.Pixbuf`

- GObject types (GTK+ GTypes) specified either as GObject Type constants or as strings. Most GTypes are mapped to a Python type:
  - gobject.TYPE_CHAR or 'gchar'
  - gobject.TYPE_UCHAR or 'guchar'
  - gobject.TYPE_BOOLEAN or 'gboolean'
  - gobject.TYPE_INT or 'gint'
  - gobject.TYPE_UINT or 'guint'
  - gobject.TYPE_LONG or 'glong
  - gobject.TYPE_ULONG or 'gulong
  - gobject.TYPE_INT64 or 'gint64'
  - gobject.TYPE_UINT64 or 'guint64'
  - gobject.TYPE_FLOAT or 'gfloat'
  - gobject.TYPE_DOUBLE or 'gdouble'
  - gobject.TYPE_STRING or 'gchararray'
  - gobject.TYPE_OBJECT or 'GObject

For example to create a `ListStore` or `TreeStore` with rows containing a `gdk.Pixbuf`, an integer, a string and boolean you could do something like:

```
liststore = ListStore(gtk.gdk.Pixbuf, int, str, 'gboolean')

treestore = TreeStore(gtk.gdk.Pixbuf, int, str, 'gboolean')
```

Once a `ListStore` or `TreeStore` is created and its columns defined, they cannot be changed or modified. It's also important to realize that there is no preset relation between the columns in a `TreeView` and the columns of its `TreeModel`. That is, the fifth column of data in a `TreeModel` may be displayed in the first column of one `TreeView` and in the third column in another. So you don't have to worry about how the data will be displayed when creating the data store.

If these two data stores do not fit your application it is possible to define your own custom data store in Python as long as it implements the TreeModel interface. I'll talk more about this later in Section 14.11.

14.2.3 Referring to TreeModel Rows

Before we can talk about managing the data rows in a `TreeStore` or `ListStore` we need a way of specifying which row we want to deal with. PyGTK has three ways of referring to `TreeModel` rows: a tree path, a `TreeIter` and a `TreeRowReference`.

14.2.3.1 Tree Paths

A tree path is a int, string or tuple representation of the location of a row in the store. An int value specifies the top level row in the model starting from 0. For example, a tree path value of 4 would specify the fifth row in the store. By comparison, a string representation of the same row would be "4" and the tuple representation would be (4,). This is sufficient for specifying any row in a `ListStore` but for a TreeStore we have to be able to represent the child rows. For these cases we have to use either the string or tuple representations.

Since a `TreeStore` can have an arbitrarily deep hierarchy the string representation specifies the path from the top level to the designated row using ints separated by the ":" character. Similarly, the tuple representation specifies the tree path starting from the top level to the row as a sequence of ints. For example, valid tree path string representations are: "0:2" (specifies the row that is the third child of the first row) and "4:0:1" (specifies the row that is the second child of the first child of the fifth row). By comparison the same tree paths are represented by the tuples (0, 2) and (4, 0, 1) respectively.

A tree path provides the only way to map from a `TreeView` row to a `TreeModel` row because the tree path of a `TreeView` row is the same as the tree path of the corresponding `TreeModel` row. There are also some problems with tree paths:

- a tree path can specify a row that doesn't exist in the ListStore or TreeStore.

- a tree path can point to a different data row after inserting or deleting a row in the `ListStore` or `TreeStore`.

PyGTK uses the tuple representation when returning tree paths but will accept any of the three forms for a tree path representation. You should use the tuple representation for a tree path for consistency.

A tree path can be retrieved from a `TreeIter` using the get_path() method:

```
path = store.get_path(iter)
```

where *iter* is a `TreeIter` pointing at a row in store and *path* is the row's tree path as a tuple.

14.2.3.2 TreeIters

A `TreeIter` is an object that provides a transient reference to a `ListStore` or `TreeStore` row. If the contents of the store change (usually because a row is added or deleted) the `TreeIter`s can become invalid. A `TreeModel` that supports persistent TreeIters should set the `gtk.TREE_MODEL_ITERS_PERSIST` flag. An application can check for this flag using the get_flags() method.

A `TreeIter` is created by one of the `TreeModel` methods that are applicable to both `TreeStore` and `ListStore` objects:

```
treeiter = store.get_iter(path)
```

where *treeiter* points at the row at the tree path *path*. The ValueError exception is raised if the tree path is invalid.

```
treeiter = store.get_iter_first()
```

where *treeiter* is a TreeIter pointing at the row at tree path (0,). *treeiter* will be `None` if the store is empty.

```
treeiter = store.iter_next(iter)
```

where *treeiter* is a `TreeIter` that points at the next row at the same level as the `TreeIter` specified by *iter*. *treeiter* will be `None` if there is no next row (*iter* is also invalidated).

The following methods are useful only for retrieving a `TreeIter` from a `TreeStore`:

```
treeiter = treestore.iter_children(parent)
```

where *treeiter* is a `TreeIter` pointing at the first child row of the row specified by the `TreeIter` *parent*. *treeiter* will be `None` if there is no child.

```
treeiter = treestore.iter_nth_child(parent, n)
```

where *treeiter* is a `TreeIter` pointing at the child row (with the index *n*) of the row specified by the `TreeIter` *parent*. *parent* may be `None` to retrieve a top level row. *treeiter* will be `None` if there is no child.

```
treeiter = treestore.iter_parent(child)
```

where *treeiter* is a `TreeIter` pointing at the parent row of the row specified by the `TreeIter` *child*. treeiter will be `None` if there is no child.

A tree path can be retrieved from a `TreeIter` using the get_path() method:

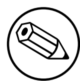
```
path = store.get_path(iter)
```

where *iter* is a `Treeiter` pointing at a row in store and *path* is the row's tree path as a tuple.

14.2.3.3 TreeRowReferences

A `TreeRowReference` is a persistent reference to a row of data in a store. While the tree path (i.e. the location) of the row might change as rows are added to or deleted from the store, the `TreeRowReference` will point at the same data row as long as it exists.

NOTE



`TreeRowReference`s are only available in PyGTK 2.4 and above.

You can create a `TreeRowReference` using its constructor:

```
treerowref = TreeRowReference(model, path)
```

where *model* is the `TreeModel` containing the row and *path* is the tree path of the row to track. If *path* isn't a valid tree path for *model*, `None` is returned.

14.2.4 Adding Rows

14.2.4.1 Adding Rows to a ListStore

Once you have a `ListStore` you'll need to add data rows using one of the following methods:

```
iter = append(row=None)
iter = prepend(row=None)
iter = insert(position, row=None)
iter = insert_before(sibling, row=None)
iter = insert_after(sibling, row=None)
```

Each of these methods inserts a row at an implied or specified position in the `ListStore`. The append() and prepend() methods use implied positions: after the last row and before the first row, respectively. The insert() method takes an integer (the parameter *position*) that specifies the location where the row will be inserted. The other two methods take a `TreeIter` (*sibling*) that references a row in the `ListStore` to insert the row before or after.

The *row* parameter specifies the data that should be inserted in the row after it is created. If *row* is `None` or not specified, an empty row will be created. If *row* is specified it must be a tuple or list containing as many items as the number of columns in the `ListStore`. The items must also match the data type of their respective `ListStore` columns.

All methods return a `TreeIter` that points at the newly inserted row. The following code fragment illustrates the creation of a `ListStore` and the addition of data rows to it:

```
...
liststore = gtk.ListStore(int, str, gtk.gdk.Color)
liststore.append([0,'red',colormap.alloc_color('red')])
liststore.append([1,'green',colormap.alloc_color('green')])
iter = liststore.insert(1, (2,'blue',colormap.alloc_color('blue')) )
iter = liststore.insert_after(iter, [3,'yellow',colormap.alloc_color('blue')])
...
```

14.2.4.2 Adding Rows to a TreeStore

Adding a row to a `TreeStore` is similar to adding a row to a `ListStore` except that you also have to specify a parent row (using a `TreeIter`) to add the new row to. The `TreeStore` methods are:

```
iter = append(parent, row=None)
iter = prepend(parent, row=None)
iter = insert(parent, position, row=None)
iter = insert_before(parent, sibling, row=None)
iter = insert_after(parent, sibling, row=None)
```

If *parent* is `None`, the row will be added to the top level rows.

Each of these methods inserts a row at an implied or specified position in the `TreeStore`. The append() and prepend() methods use implied positions: after the last child row and before the first child row, respectively. The insert() method takes an integer (the parameter *position*) that specifies the location where the child row will be inserted. The other two methods take a `TreeIter` (*sibling*) that references a child row in the `TreeStore` to insert the row before or after.

The *row* parameter specifies the data that should be inserted in the row after it is created. If *row* is `None` or not specified, an empty row will be created. If *row* is specified it must be a tuple or list containing as many items as the number of columns in the `TreeStore`. The items must also match the data type of their respective `TreeStore` columns.

All methods return a `TreeIter` that points at the newly inserted row. The following code fragment illustrates the creation of a `TreeStore` and the addition of data rows to it:

```
...
folderpb = gtk.gdk.pixbuf_from_file('folder.xpm')
filepb = gtk.gdk.pixbuf_from_file('file.xpm')
treestore = gtk.TreeStore(int, str, gtk.gdk.Pixbuf)
iter0 = treestore.append(None, [1,'(0,)',folderpb] )
treestore.insert(iter0, 0, [11,'(0,0)',filepb])
treestore.append(iter0, [12,'(0,1)',filepb])
iter1 = treestore.insert_after(None, iter0, [2,'(1,)',folderpb])
treestore.insert(iter1, 0, [22,'(1,1)',filepb])
treestore.prepend(iter1, [21,'(1,0)',filepb])
...
```

14.2.4.3 Large Data Stores

When a `ListStore` or `TreeStore` contains a large number of data rows, adding new rows can become very slow. There are a few things that you can do to mitigate this problem:

- If adding a large number of rows disconnect the `TreeModel` from its `TreeView` (using the set_model() method with the *model* parameter set to `None`) to avoid `TreeView` updates for each row entered.
- Likewise, disable sorting (using the set_default_sort_func() method with the *sort_func* set to `None`) while adding a large number of rows.
- Limit the number of `TreeRowReference`s in use since they update their path with each addition or removal.
- Set the `TreeView` "fixed-height-mode" property to `TRUE` making all rows have the same height and avoiding the individual calculation of the height of each row. Only available in PyGTK 2.4 and above.

14.2.5 Removing Rows

14.2.5.1 Removing Rows From a ListStore

You can remove a data row from a `ListStore` by using the remove() method:

```
treeiter = liststore.remove(iter)
```

where *iter* is a `TreeIter` pointing at the row to remove. The returned `TreeIter` (*treeiter*) points at the next row or is invalid if *iter* was pointing at the last row.

The clear() method removes all rows from the `ListStore`:

```
liststore.clear()
```

14.2.5.2 Removing Rows From a TreeStore

The methods for removing data rows from a `TreeStore` are similar to the `ListStore` methods:

```
result = treestore.remove(iter)
treestore.clear()
```

where *result* is `TRUE` if the row was removed and *iter* points at the next valid row. Otherwise, *result* is `FALSE` and *iter* is invalidated.

14.2.6 Managing Row Data

14.2.6.1 Setting and Retrieving Data Values

The methods for accessing the data values in a `ListStore` and `TreeStore` have the same format. All store data manipulations use a `TreeIter` to specify the row that you are working with. Once you have a `TreeIter` it can be used to retrieve the values of a row column using the get_value() method:

```
value = store.get_value(iter, column)
```

where *iter* is a `TreeIter` pointing at a row, *column* is a column number in *store*, and, *value* is the value stored at the row-column location.

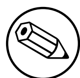
If you want to retrieve the values from multiple columns in one call use the get() method:

```
values = store.get(iter, column, ...)
```

where *iter* is a `TreeIter` pointing at a row, *column* is a column number in *store*, and, `...` represents zero or more additional column numbers and *values* is a tuple containing the retrieved data values. For example to retrieve the values in columns 0 and 2:

```
val0, val2 = store.get(iter, 0, 2)
```

NOTE



The get() method is only available in PyGTK 2.4 and above.

Setting a single column value is effected using the set_value() method:

```
store.set_value(iter, column, value)
```

where *iter* (a `TreeIter`) and *column* (an int) specify the row-column location in *store* and *column* is the column number where *value* is to be set. *value* must be the same data type as the *store* column.

If you wish to set the value of more than one column in a row at a time, use the set() method:

```
store.set(iter, ...)
```

where *iter* specifies the store row and `...` is one or more column number - value pairs indicating the column and and value to set. For example, the following call:

```
store.set(iter, 0, 'Foo', 5, 'Bar', 1, 123)
```

sets the first column to 'Foo', the sixth column to 'Bar' and the second column to 123 in the *store* row specified by *iter*.

14.2.6.2 Rearranging ListStore Rows

Individual `ListStore` rows can be moved using one of the following methods that are available in PyGTK 2.2 and above:

```
liststore.swap(a, b)
liststore.move_after(iter, position)
liststore.move_before(iter, position)
```

swap() swaps the locations of the rows referenced by the `TreeIters` *a* and *b*. move_after() and move_before() move the row referenced by the `TreeIter` *iter* after or before the row referenced by the `TreeIter` *position*. If *position* is `None`, move_after() will place the row at the beginning of the store while move_before(), at the end of the store.

If you want to completely rearrange the `ListStore` data rows, use the following method:

```
liststore.reorder(new_order)
```

where *new_order* is a list of integers that specify the new row order. The child nodes will be rearranged so that the *liststore* node that is at position index *new_order*`[i]` will be located at position index `i`.

For example, if *liststore* contained four rows:

```
'one'
'two'
'three'
'four'
```

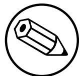
The method call:

```
liststore.reorder([2, 1, 3, 0])
```

would produce the resulting order:

```
'three'
'two'
'four'
'one'
```

NOTE



These methods will only rearrange unsorted `ListStore`s.

If you want to rearrange rows in PyGTK 2.0 you have to remove and insert rows using the methods described in Section 14.2.4 and Section 14.2.5.

14.2.6.3 Rearranging TreeStore Rows

The methods used to rearrange `TreeStore` rows are similar to the `ListStore` methods except they only affect the child rows of an implied parent row - it is not possible to, say, swap rows with different parent rows.:

```
treestore.swap(a, b)
treestore.move_after(iter, position)
treestore.move_before(iter, position)
```

swap() swaps the locations of the child rows referenced by the TreeIters *a* and *b*. *a* and *b* must both have the same parent row. move_after() and move_before() move the row referenced by the `TreeIter` *iter* after or before the row referenced by the `TreeIter` *position*. *iter* and *position* must both have the same parent row. If *position* is `None`, move_after() will place the row at the beginning of the store while move_before(), at the end of the store.

The reorder() method requires an additional parameter specifying the parent row whose child rows will be reordered:

```
treestore.reorder(parent, new_order)
```

where *new_order* is a list of integers that specify the new child row order of the parent row specified by the `TreeIter` *parent* as:

```
new_order[newpos] = oldpos
```

For example, if *treestore* contained four rows:

```
'parent'
    'one'
    'two'
    'three'
    'four'
```

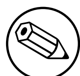
The method call:

```
treestore.reorder(parent, [2, 1, 3, 0])
```

would produce the resulting order:

```
'parent'
    'three'
    'two'
    'four'
    'one'
```

NOTE



These methods will only rearrange unsorted `TreeStore`s.

14.2.6.4 Managing Multiple Rows

One of the trickier aspects of dealing with `ListStore`s and `TreeStore`s is the operation on multiple rows, e.g. moving multiple rows, say, from one parent row to another or removing rows based on certain criteria. The difficulty arises from the need to use a `TreeIter` that may become invalid as the result of the operation. For `ListStore`s and `TreeStore`s the TreeIters are persistent as can be checked by using the get_flags() method and testing for the `gtk.TREE_MODEL_ITERS_PERSIST` flag. However the stackable `TreeModelFilter` and `TreeModelSort` classes do not have persistent `TreeIter`s.

Assuming that `TreeIter`s don't persist how do we move all the child rows from one parent row to another? We have to:

- iterate over the parent's children
- retrieve each row's data
- remove each child row
- insert a new row with the old row data in the new parent's list

We can't rely on the remove() method to return a valid `TreeIter` so we'll just ask for the first child iter until it returns `None`. A possible function to move child rows is:

```
def move_child_rows(treestore, from_parent, to_parent):
  n_columns = treestore.get_n_columns()
  iter = treestore.iter_children(from_parent)
  while iter:
    values = treestore.get(iter, *range(n_columns))
    treestore.remove(iter)
    treestore.append(to_parent, values)
    iter = treestore.iter_children(from_parent)
  return
```

The above function covers the simple case of moving all child rows of a single parent row but what if you want to remove all rows in the `TreeStore` based on some match criteria, say the first column value? Here you might think that you could use the foreach() method to iterate over all the rows and remove the matching ones:

```
store.foreach(func, user_data)
```

where *func* is a function that is invoked for each store row and has the signature:

```
def func(model, path, iter, user_data):
```

where *model* is the TreeModel data store, *path* is the tree path of a row in *model*, *iter* is a TreeIter pointing at *path* and *user_data* is the passed in data. if *func* returns TRUE the foreach() method will cease iterating and return.

The problem with that is that changing the contents of the store while the foreach() method is iterating over it may have unpredictable results. Using the foreach() method to create and save TreeRowReferences to the rows to be removed and then removing them after the foreach() method completes would be a good strategy except that it doesn't work for PyGTK 2.0 and 2.2 where TreeRowReferences are not available.

A reliable strategy that covers all the PyGTK variants is to use the foreach() method to gather the tree paths of rows to be removed and then remove them in reverse order to preserve the validity of the tree paths. An example code fragment utilizing this strategy is:

```
...
# match if the value in the first column is >= the passed in value
# data is a tuple containing the match value and a list to save paths
def match_value_cb(model, path, iter, data):
  if model.get_value(iter, 0) >= data[0]:
    data[1].append(path)
  return False     # keep the foreach going

pathlist = []
treestore.foreach(match_value_cb, (10, pathlist))

# foreach works in a depth first fashion
pathlist.reverse()
for path in pathlist:
  treestore.remove(treestore.get_iter(path))
...
```

If you want to search a TreeStore for the first row that matches some criteria, you probably want to do the iteration yourself using something like:

```
 treestore = TreeStore(str)
 ...
 def match_func(model, iter, data):
     column, key = data # data is a tuple containing column number, key
     value = model.get_value(iter, column)
     return value == key
 def search(model, iter, func, data):
     while iter:
         if func(model, iter, data):
             return iter
         result = search(model, model.iter_children(iter), func, data)
         if result: return result
         iter = model.iter_next(iter)
     return None
 ...
 match_iter = search(treestore, treestore.iter_children(None),
                     match_func, (0, 'foo'))
```

The search() function iterates recursively over the row (specified by *iter*) and its siblings and their child rows in a depth first fashion looking for a row that has a column matching the given key string. The search terminates when a row is found.

14.2.7 Python Protocol Support

The classes that implement the TreeModel interface (TreeStore and ListStore and in PyGTK 2.4, also the TreeModelSort and TreeModelFilter) support the Python mapping and iterator protocols. The iterator protocol allows you to use the Python iter() function on a TreeModel to create an iterator to be used to iterate over the top level rows in the TreeModel. A more useful capability is to iterate using the for statement or a list comprehension. For example:

```
...
liststore = gtk.ListStore(str, str)
```

```
...
# add some rows to liststore
...
# for looping
for row in liststore:
    # do individual row processing
...
# list comprehension returning a list of values in the first column
values = [ r[0] for r in liststore ]
...
```

Other parts of the mapping protocols that are supported are using `del` to delete a row in the model and extracting a PyGTK `TreeModelRow` from the model using a key value that is a tree path or `TreeIter`. For example, the following statements all return the first row in a `TreeModel` and the final statement deletes the first child row of the first row:

```
row = model[0]
row = model['0']
row = model["0"]
row = model[(0,)]
i = model.get_iter(0)
row = model[i]
del model[(0,0)]
```

In addition, you can set the values in an existing row similar to the following:

```
...
liststore = gtk.ListStore(str, int, object)
...
liststore[0] = ['Button', 23, gtk.Button('Label')]
```

A PyGTK `TreeModelRow` object supports the Python sequence and iterator protocols. You can get an iterator to iterate over the column values in the row or use the for statement or list comprehension as well. A `TreeModelRow` uses the column number as the index to extract a value. For example:

```
...
liststore = gtk.ListStore(str, int)
liststore.append(['Random string', 514])
...
row = liststore[0]
value1 = row[1]
value0 = liststore['0'][0]
for value in row:
    print value
val0, val1 = row
...
```

Using the example from the previous section to iterate over a `TreeStore` to locate a row containing a particular value, the code becomes:

```
treestore = TreeStore(str)
...
def match_func(row, data):
    column, key = data # data is a tuple containing column number, key
    return row[column] == key
...
def search(rows, func, data):
    if not rows: return None
    for row in rows:
        if func(row, data):
            return row
        result = search(row.iterchildren(), func, data)
        if result: return result
    return None
...
match_row = search(treestore, match_func, (0, 'foo'))
```

You can also set a value in an existing column using:

```
treestore[(1,0,1)][1] = 'abc'
```

The `TreeModelRow` also supports the `del` statement and conversion to lists and tuples using the Python `list`() and `tuple`() functions. As illustrated in the above example the `TreeModelRow` has the iterchildren() method that returns an iterator for iterating over the child rows of the `TreeModelRow`.

14.2.8 TreeModel Signals

Your application can track changes in a `TreeModel` by connecting to the signals that are emitted by the `TreeModel`: "row-changed", "row-deleted", "row-inserted", "row-has-child-toggled" and "rows-reordered". These signals are used by a `TreeView` to track changes in its `TreeModel`.

If you connect to these signals in your application, you may see clusters of signals when some methods are called. For example the call to add the first child row to a parent row:

```
treestore.append(parent, ['qwe', 'asd', 123])
```

will cause the following signal emissions:

- "row-inserted" where the inserted row will be empty.
- "row-has-child-toggled" since *parent* didn't previously have any child rows.
- "row-changed" for the inserted row when setting the value 'qwe' in the first column.
- "row-changed" for the inserted row when setting the value 'asd in the second column.
- "row-changed" for the inserted row when setting the value 123 in the third column.

Note that you can't retrieve the row order in the "rows-reordered" callback since the new row order is passed as an opaque pointer to an array of integers.

See the PyGTK Reference Manual for more information on the `TreeModel` signals.

14.2.9 Sorting TreeModel Rows

14.2.9.1 The TreeSortable Interface

The `ListStore` and `TreeStore` objects implement the `TreeSortable` interface that provides methods for controlling the sorting of `TreeModel` rows. The key element of the interface is a "sort column ID" which is an arbitrary integer value referring to a sort comparison function and associated user data. A sort column ID must be greater than or equal to zero. A sort column ID is created by using the method:

```
treesortable.set_sort_func(sort_column_id, sort_func, user_data=None)
```

where *sort_column_id* is a programmer assigned integer value, *sort_func* is a function or method used to compare rows and *user_data* is context data. *sort_func* has the signature:

```
def sort_func_function(model, iter1, iter2, data)
def sort_func_method(self, model, iter1, iter2, data)
```

where *model* is the `TreeModel` containing the rows pointed to by the `TreeIters` *iter1* and *iter2* and *data* is *user_data*. *sort_func* should return: -1 if the *iter1* row should precede the *iter2* row; 0, if the rows are equal; and, 1 if the *iter2* row should precede the *iter1* row. The sort comparison function should always assume that the sort order is `gtk.SORT_ASCENDING` as the sort order will be taken into account by the `TreeSortable` implementations.

The same sort comparison function can be used for multiple sort column IDs by varying the user_data to provide context information. For example, the *user_data* specified in the set_sort_func() method could be the index of the column to extract the sort data from.

Once a sort column ID is created a store can use it for sorting by calling the method:

```
treesortable.set_sort_column_id(sort_column_id, order)
```

where *order* is the sort order either `gtk.SORT_ASCENDING` or `gtk.SORT_DESCENDING`.

The sort column ID of -1 means that the store should use the default sort function that is set using the method:

```
treesortable.set_default_sort_func(sort_func, user_data=None)
```

You can check if a store has a default sort function using the method:

```
result = treesortable.has_default_sort_func()
```

which returns TRUE if a default sort function has been set.

Once a sort column ID has been set on a TreeModel implementing the TreeSortable interface it cannot be returned to the original unsorted state. You can change the sort function or use a default sort function but you cannot set the TreeModel to have no sort function.

14.2.9.2 Sorting in ListStores and TreeStores

When a ListStore or TreeStore object is created it automatically sets up sort column IDs corresponding to the columns in the store using the column index number. For example, a ListStore with three columns would have three sort column IDs (0, 1, 2) setup automatically. These sort column IDs are associated with an internal sort comparison function that handles the fundamental types:

- 'gboolean'
- str
- int
- long
- float

Initially a ListStore or TreeStore is set with a sort column ID of -2 that indicates that no sort function is being used and that the store is unsorted. Once you set a sort column ID on a ListStore or TreeStore you cannot set it back to -2.

If you want to maintain the default sort column IDs you can set up a sort column ID well out of the range of the number of columns such as 1000 and up. Then you can switch between the default sort function and your application sort functions as needed.

14.3 TreeViews

A TreeView is basically a container for the TreeViewColumn and CellRenderer objects that do the actual display of the data store data. It also provides an interface to the displayed data rows and to the characteristics that control the data display.

14.3.1 Creating a TreeView

A TreeView is created using its constructor:

```
treeview = gtk.TreeView(model=None)
```

where *model* is an object implementing the TreeModel interface (usually a ListStore or TreeStore). If *model* is None or not specified the TreeView will not be associated with a data store.

14.3.2 Getting and Setting the TreeView Model

The tree model providing the data store for a TreeView can be retrieved using the get_model() method:

```
model = treeview.get_model()
```

A TreeModel may be simultaneously associated with more than one TreeView which automatically changes its display when the TreeModel data changes. While a TreeView always displays all of the rows of its tree model, it may display only some of the tree model columns. This means that two TreeViews associated with the same TreeModel may provide completely different views of the same data.

It's also important to realize that there is no preset relation between the columns in a TreeView and the columns of its TreeModel. That is, the fifth column of data in a TreeModel may be displayed in the first column of one TreeView and in the third column in another.

A TreeView can change its tree model using the set_model() method:

```
treeview.set_model(model=None)
```

where *model* is an object implementing the TreeModel interface (e.g. ListStore and TreeStore). If *model* is None, the current model is discarded.

14.3.3 Setting TreeView Properties

The TreeView has a number of properties that can be managed using its methods:

"enable-search"	Read-Write	If TRUE, the user can search through columns interactively. Default is TRUE
"expander-column"	Read-Write	The column for the expander. Default is 0
"fixed-height-mode"	Read-Write	If TRUE, assume all rows have the same height thereby speeding up display. Available in GTK+ 2.4 and above. Default is FALSE
"hadjustment"	Read-Write	The horizontal Adjustment for the widget. New one created by default.
"headers-clickable"	Write	If TRUE, the column headers respond to click events. Default is FALSE
"headers-visible"	Read-Write	If TRUE, show the column header buttons. Default is TRUE
"model"	Read-Write	The model for the tree view. Default is None
"reorderable"	Read-Write	If TRUE, the view is reorderable. Default is FALSE
"rules-hint"	Read-Write	If TRUE, hint to the theme engine to draw rows in alternating colors. Default is FALSE
"search-column"	Read-Write	The model column to search when searching through code. Default is -1.
"vadjustment"	Read-Write	The vertical Adjustment for the widget. New one created by default.

The corresponding methods are:

```
enable_search = treeview.get_enable_search()
treeview.set_enable_search(enable_search)

column = treeview.get_expander_column()
treeview.set_expander_column(column)

hadjustment = treeview.get_hadjustment()
treeview.set_hadjustment(adjustment)

treeview.set_headers_clickable(active)

headers_visible = treeview.get_headers_visible()
treeview.set_headers_visible(headers_visible)

reorderable = treeview.get_reorderable()
treeview.set_reorderable(reorderable)

riles_hint = treeview.get_rules_hint()
treeview.set_rules_hint(setting)

column = treeview.get_search_column()
treeview.set_search_column(column)

vadjustment = treeview.get_vadjustment()
treeview.set_vadjustment(adjustment)
```

Most of these are obvious from the description. However, the "enable-search" property requires the "search-column" property to be set to the number of a valid column in the tree model. Then when the

user presses Control+f a search dialog is popped up that the user can type in. The first matching row will be automatically selected as the user types.

Likewise, the "headers-clickable" property really just sets the "clickable" property of the underlying TreeViewColumns. A TreeViewColumn will not be sortable unless the tree model implements the TreeSortable interface and the TreeViewColumn set_sort_column_id() method has been called with a valid column number.

The "reorderable" property enables the user to reorder the TreeView model by dragging and dropping the TreeView rows displayed.

The "rules-hint" property should only be set if you have lots of columns and think that alternating colors may help the user.

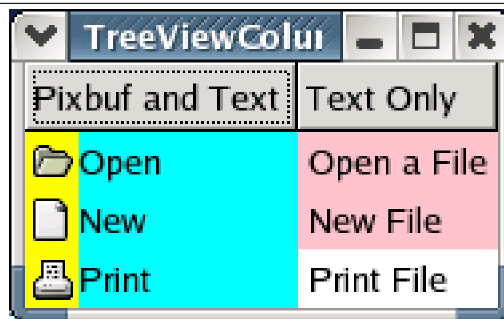
14.4 CellRenderers

14.4.1 Overview

TreeViewColumns and CellRenderers work together to display a column of data in a TreeView. The TreeViewColumn provides the column title and a vertical space for the CellRenderers to render a portion of the data from the TreeView data store. A CellRenderer handles the rendering of each row and column data within the confines of the TreeViewColumn. A TreeViewColumn can contain more than one CellRenderer to provide a row display similar to an HBox. A common use of multiple CellRenderers is to combine a CellRendererPixbuf and a CellRendererText in one column.

An example illustrating the layout of two TreeViewColumns: one with two CellRenderers and one with one CellRenderer is shown in Figure 14.2:

Figure 14.2 TreeViewColumns with CellRenderers



The application of each CellRenderer is indicated with a different background color: yellow for the CellRendererPixbuf, cyan for one CellRendererText, and pink for the other CellRendererText. Note that the CellRendererPixbuf and the first CellRendererText are in the same column headed by the "Pixbuf and Text" header. The background color of the CellRendererText rendering "Print File" is the default color to show the application area in a single row.

Figure 14.2 was created by the treeviewcolumn.py program.

14.4.2 CellRenderer Types

The type of CellRenderer needed is determined by the type of tree model data display required; PyGTK has three pre-defined CellRenderers:

CellRendererPixbuf renders pixbuf images either created by the program or one of the stock items.

CellRendererText renders text strings, and numbers that can be converted to a string (including ints, floats, booleans).

CellRendererToggle renders a boolean value as a toggle button or a radio button

14.4.3 CellRenderer Properties

The properties of a CellRenderer determine how the data will be rendered:

"mode"	Read-Write	The editable mode of the CellRenderer. One of: gtk.CELL_RENDERER_MODE_INERT, gtk.CELL_RENDERER_MODE_ACTIVATABLE or gtk.CELL_RENDERER_MODE_EDITABLE
"visible"	Read-Write	If TRUE the cell is displayed
"xalign"	Read-Write	The fraction of *free* space to the left of the cell in the range 0.0 to 1.0.
"yalign"	Read-Write	The fraction of *free* space above the cell in the range 0.0 to 1.0.
"xpad"	Read-Write	The amount of padding to the left and right of the cell.
"ypad"	Read-Write	The amount of padding above and below cell.
"width"	Read-Write	The fixed width of the cell.
"height"	Read-Write	The fixed height of the cell.
"is-expander"	Read-Write	If TRUE the row has children
"is-expanded"	Read-Write	If TRUE the row has children and it is expanded to show the children.
"cell-background"	Write	The background color of the cell as a string.
"cell-background-gdk"	Read-Write	The background color of the cell as a gtk.gdk.Color.
"cell-background-set"	Read-Write	If TRUE the cell background color is set by this cellrenderer

The above properties are available for all CellRenderer subclasses. The individual CellRenderer types also have their own properties.

The CellRendererPixbuf has these properties:

"pixbuf"	Read-Write	The pixbuf to render - overridden by "stock-id"
"pixbuf-expander-open"	Read-Write	Pixbuf for open expander.
"pixbuf-expander-closed"	Read-Write	Pixbuf for closed expander.
"stock-id"	Read-Write	The stock ID of the stock icon to render
"stock-size"	Read-Write	The size of the rendered icon
"stock-detail"	Read-Write	Render detail to pass to the theme engine

The CellRendererText has a large number of properties mostly dealing with style specification:

"text"	Read-Write	Text to render
"markup"	Read-Write	Marked up text to render.
"attributes"	Read-Write	A list of style attributes to apply to the text of the renderer.
"background"	Write	Background color as a string
"foreground"	Write	Foreground color as a string
"background-gdk"	Read-Write	Background color as a gtk.gdk.Color
"foreground-gdk"	Read-Write	Foreground color as a gtk.gdk.Color
"font"	Read-Write	Font description as a string
"font-desc"	Read-Write	Font description as a pango.FontDescription
"family"	Read-Write	Name of the font family, e.g. Sans, Helvetica, Times, Monospace
"style"	Read-Write	Font style
"variant"	Read-Write	Font variant
"weight"	Read-Write	Font weight
"stretch"	Read-Write	Font stretch
"size"	Read-Write	Font size
"size-points"	Read-Write	Font size in points
"scale"	Read-Write	Font scaling factor
"editable"	Read-Write	If TRUE the text can be modified by the user
"strikethrough"	Read-Write	If TRUE strike through the text
"underline"	Read-Write	Style of underline for this text
"rise"	Read-Write	Offset of text above the baseline (below the baseline if rise is negative)
"language"	Read-Write	The language this text is in, as an ISO code. Pango can use this as a hint when rendering the text. If you don't understand this parameter, you probably don't need it. GTK+ 2.4 and above.

"single-paragraph-mode"	Read-Write	If TRUE, keep all text in a single paragraph. GTK+ 2.4 and above.

"background-set"	Read-Write	If TRUE apply the background color
"foreground-set"	Read-Write	If TRUE apply the foreground color
"family-set"	Read-Write	If TRUE apply the font family
"style-set"	Read-Write	If TRUE apply the font style
"variant-set"	Read-Write	If TRUE apply the font variant
"weight-set"	Read-Write	If TRUE apply the font weight
"stretch-set"	Read-Write	If TRUE apply the font stretch
"size-set"	Read-Write	If TRUE apply the font size
"scale-set"	Read-Write	If TRUE scale the font
"editable-set"	Read-Write	If TRUE apply the text editability
"strikethrough-set"	Read-Write	If TRUE apply the strikethrough
"underline-set"	Read-Write	If TRUE apply the text underlining
"rise-set"	Read-Write	If TRUE apply the rise
"language-set"	Read-Write	If TRUE apply the language used to render the text. GTK+ 2.4 and above.

Almost every `CellRendererText` property has an associated boolean property (with the "-set" suffix) that indicates if the property is to be applied. This allows you to set a property globally and selectively enable and disable its application.

The `CellRendererToggle` has the following properties:

"activatable"	Read-Write	If TRUE, the toggle button can be activated
"active"	Read-Write	If TRUE, the button is active.
"radio"	Read-Write	If TRUE, draw the toggle button as a radio button
"inconsistent"	Read-Write	If TRUE, the button is in an inconsistent state. GTK+ 2.2 and above.

The properties can be set for all rows by using the gobject.set_property() method. See the treeviewcolumn.py program for an example using this method.

14.4.4 CellRenderer Attributes

An attribute associates a tree model column with a `CellRenderer` property; the `CellRenderer` sets the property from the row's column value before rendering the cell. This allows you to customize the cell display using tree model data. An attribute can be added to the current set by using:

```
treeviewcolumn.add_attribute(cell_renderer, attribute, column)
```

where the property specified by *attribute* is set for the *cell_renderer* from *column*. For example:

```
treeviewcolumn.add_attribute(cell, "cell-background", 1)
```

sets the `CellRenderer` background to the color specified by the string in the second column of the data store.

To clear all attributes and set several new attributes at once use:

```
treeviewcolumn.set_attributes(cell_renderer, ...)
```

where the attributes of *cell_renderer* are set by key-value pairs: property=column. For example, for a `CellRendererText`:

```
treeviewcolumn.set_attributes(cell, text=0, cell_background=1, xpad=3)
```

sets, for each row, the text from the first column, the background color from the second column and the horizontal padding from the fourth column. See the treeviewcolumn.py program for an example using these methods.

The attributes of a `CellRenderer` can be cleared using:

```
treeviewcolumn.clear_attributes(cell_renderer)
```

14.4.5 Cell Data Function

If setting attributes is not sufficient for your needs you can set a function to be called for each row to set the properties for that `CellRenderer` using:

```
treeviewcolumn.set_cell_data_func(cell_renderer, func, data=None)
```

where *func* has the signature:

```
def func(column, cell_renderer, tree_model, iter, user_data)
```

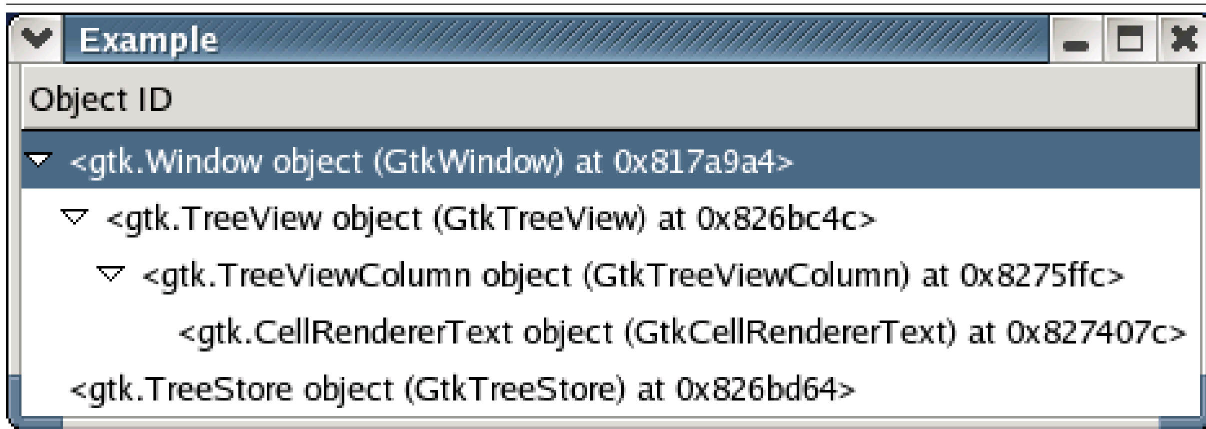
where *column* is the `TreeViewColumn` containing *cell_renderer*, *tree_model* is the data store and *iter* is a `TreeIter` pointing at a row in *tree_model*. *user_data* is the value of *data* that was passed to set_cell_data_func().

In *func* you set whatever properties you want on *cell_renderer*. For example the following code fragment sets the text property to display PyGTK objects as an ID string.

```
...
def obj_id_str(treeviewcolumn, cell, model, iter):
    pyobj = model.get_value(iter, 0)
    cell.set_property('text', str(pyobj))
    return
...
treestore = gtk.TreeStore(object)
win = gtk.Window()
treeview = gtk.TreeView(treestore)
win.add(treeview)
cell = CellRendererText()
tvcolumn = gtk TreeViewColumn('Object ID', cell)
tvcolumn.set_cell_data_func(cell, obj_id_str)
treeview.append_column(tvcolumn)
iter = treestore.append(None, [win])
iter = treestore.append(iter, [treeview])
iter = treestore.append(iter, [tvcolumn])
iter = treestore.append(iter, [cell])
iter = treestore.append(None, [treestore])
...
```

The resulting display should be something like Figure 14.3:

Figure 14.3 CellRenderer Data Function



Another use of a cell data function is to control the formatting of a numerical text display e.g. a float value. A `CellRendererText` will display and automatically convert a float to a string but with a default format "%f".

With cell data functions you can even generate the cell data for the columns from external data. For example the filelisting.py program uses a `ListStore` with just one column that holds a list of file names. The `TreeView` displays columns that include a pixbuf, the file name and the file's size, mode and time of last change. The data is generated by the following cell data functions:

```
    def file_pixbuf(self, column, cell, model, iter):
        filename = os.path.join(self.dirname, model.get_value(iter, 0))
        filestat = statcache.stat(filename)
        if stat.S_ISDIR(filestat.st_mode):
            pb = folderpb
        else:
            pb = filepb
        cell.set_property('pixbuf', pb)
        return

    def file_name(self, column, cell, model, iter):
        cell.set_property('text', model.get_value(iter, 0))
        return

    def file_size(self, column, cell, model, iter):
        filename = os.path.join(self.dirname, model.get_value(iter, 0))
        filestat = statcache.stat(filename)
        cell.set_property('text', filestat.st_size)
        return

    def file_mode(self, column, cell, model, iter):
        filename = os.path.join(self.dirname, model.get_value(iter, 0))
        filestat = statcache.stat(filename)
        cell.set_property('text', oct(stat.S_IMODE(filestat.st_mode)))
        return


    def file_last_changed(self, column, cell, model, iter):
        filename = os.path.join(self.dirname, model.get_value(iter, 0))
        filestat = statcache.stat(filename)
        cell.set_property('text', time.ctime(filestat.st_mtime))
        return
```

These cell data functions retrieve the file information using the name, extract the needed data and set the cell 'text' or 'pixbuf' property with the data. Figure 14.4 shows the example program in action:

Figure 14.4 File Listing Example Using Cell Data Functions

/usr/include

Name	Size	Mode	Last Changed
..	4096	0755	Sat May 1 13:09:07 2004
DPS	4096	0755	Sat May 1 13:45:32 2004
FlexLexer.h	5826	0644	Fri Jan 24 16:05:45 2003
GL	4096	0755	Sat May 1 13:45:41 2004
Imlib.h	5964	0644	Fri Jan 24 14:44:27 2003
Imlib_private.h	4790	0644	Fri Jan 24 14:44:27 2003
Imlib_types.h	5132	0644	Fri Jan 24 14:44:27 2003
Mrm	4096	0755	Sat May 1 13:45:12 2004
SDL	4096	0755	Sat May 1 13:46:03 2004
X11	4096	0755	Sat May 1 13:45:34 2004
Xm	8192	0755	Sat May 1 13:45:13 2004
_G_config.h	2647	0644	Thu Mar 13 15:00:22 2003
a.out.h	83	0644	Thu Mar 13 15:01:10 2003

14.4.6 CellRendererText Markup

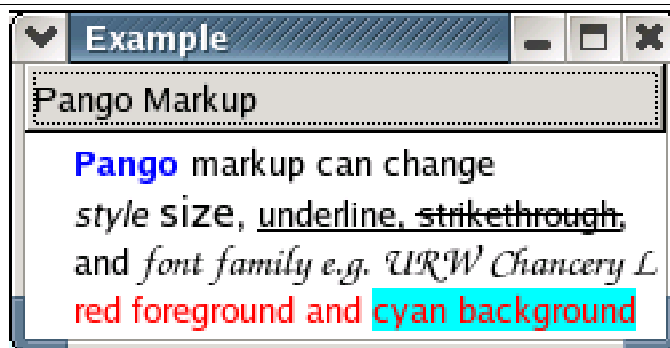
A `CellRendererText` can use Pango markup (by setting the "markup" property) instead of a plain text string to encode various text attributes and provide a rich text display with multiple font style changes. See the Pango Markup reference in the PyGTK Reference Manual for details on the Pango markup language.

The following code fragment illustrates the use of the "markup" property:

```
  ...
  liststore = gtk.ListStore(str)
  cell = gtk.CellRendererText()
  tvcolumn = gtk.TreeViewColumn('Pango Markup', cell, markup=0)
  ...
  liststore.append(['<span foreground="blue"><b>Pango</b></span> markup can'
' change\n<i>style</i> <big>size</big>, <u>underline,'
 <s>strikethrough</s></u>,\n'
'and <span font_family="URW Chancery L"><big>font family '
'e.g. URW Chancery L</big></span>\n<span foreground="red">red'
' foreground and <span background="cyan">cyan background</span></span>'])
  ...
```

produces a display similar to Figure 14.5:

Figure 14.5 CellRendererText Markup



If you create pango markup on the fly you have to be careful to replace the characters that are special to the markup language: "<", ">", "&". The Python library function `cgi.escape()` can do these basic conversions.

14.4.7 Editable Text Cells

`CellRendererText` cells can be made editable to allow a user to edit the contents of the cell that is selected by clicking it or pressing one of the **Return**, **Enter**, **Space** or Shift+Space keys. A `CellRendererText` is made editable for all rows by setting its "editable" property to `TRUE` as follows:

```
cellrenderertext.set_property('editable', True)
```

Individual cells can be set editable by adding an attribute to the `TreeViewColumn` using the `CellRendererText` similar to:

```
treeviewcolumn.add_attribute(cellrenderertext, "editable", 2)
```

which sets the "editable" property to the value contained in the third column of the data store.

Once the cell editing completes, your application should handle the "edited" signal to retrieve the new text and set the associated data store value. Otherwise the cell value reverts to its original value. The signature of the "edited" signal handler is:

```
def edited_cb(cell, path, new_text, user_data)
```

where *`cell`* is the `CellRendererText`, *`path`* is the tree path (as a string) to the row containing the edited cell, *`new_text`* is the edited text and *`user_data`* is context data. Since the `TreeModel` is needed to use *`path`* to set *`new_text`* in the data store you probably want to pass the `TreeModel` as *`user_data`* in the connect() method:

```
cellrenderertext.connect('edited', edited_cb, model)
```

If you have two or more editable cells in a row, you could pass the `TreeModel` column number as part of *`user_data`* as well as the `TreeModel`:

```
cellrenderertext.connect('edited', edited_cb, (model, col_num))
```

Then you can set the new text in the "edited" handler similar to this example using a `ListStore`:

```
def edited_cb(cell, path, new_text, user_data):
    liststore, column = user_data
    liststore[path][column] = new_text
    return
```

14.4.8 Activatable Toggle Cells

`CellRendererToggle` buttons can be made activatable by setting the "activatable" property to `TRUE`. Similar to editable `CellRendererText` cells the "activatable" property can be set for the entire `CellRendererToggle` set of cells using the set_property() method or for individual cells by adding an attribute to the `TreeViewColumn` containing the `CellRendererToggle`.

```
cellrenderertoggle.set_property('activatable', True)

treeviewcolumn.add_attribute(cellrenderertoggle, "activatable", 1)
```

The setting of the individual toggle buttons can be derived from the values in a `TreeModel` column by adding an attribute, for example:

```
treeviewcolumn.add_attribute(cellrenderertoggle, "active", 2)
```

You should connect to the "toggled" signal to get notification of user clicks on the toggle buttons so that your application can change the value in the data store. For example:

```
cellrenderertoggle.connect("toggled", toggled_cb, (model, column))
```

The callback has the signature:

```
def toggled_cb(cellrenderertoggle, path, user_data)
```

where *path* is the tree path, as a string, pointing to the row containing the toggle that was clicked. You should pass the `TreeModel` and possibly the column index as part of *user_data* to provide the necessary context for setting the data store values. For example, your application can toggle the data store value as follows:

```
def toggled_cb(cell, path, user_data):
    model, column = user_data
    model[path][column] = not model[path][column]
    return
```

If your application wants to display the toggle buttons as radio buttons and have only one be set, it will have to scan the data store to deactivate the active radio button and then set the toggled button. For example:

```
def toggled_cb(cell, path, user_data):
    model, column = user_data
    for row in model:
        row[column] = False
    model[path][column] = True
    return
```

takes the lazy approach of setting all data store values to `FALSE` before setting the value to `TRUE` for the row specified by *path*.

14.4.9 Editable and Activatable Cell Example Program

The cellrenderer.py program illustrates the application of editable `CellRendererText` and activatable `CellRendererToggle` cells in a `TreeStore`.

```
 1  #!/usr/bin/env python
 2  # vim: ts=4:sw=4:tw=78:nowrap
 3  """ Demonstration using editable and activatable CellRenderers """
 4  import pygtk
 5  pygtk.require("2.0")
 6  import gtk, gobject
 7
 8  tasks =  {
 9      "Buy groceries": "Go to Asda after work",
10      "Do some programming": "Remember to update your software",
11      "Power up systems": "Turn on the client but leave the server",
12      "Watch some tv": "Remember to catch ER"
13      }
14
15  class GUI_Controller:
16      """ The GUI class is the controller for our application """
17      def __init__(self):
18          # setup the main window
```

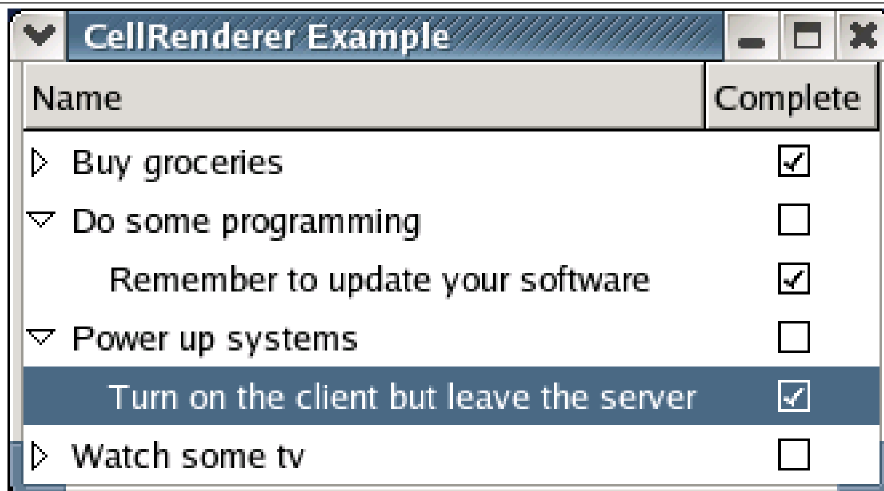
```
19          self.root = gtk.Window(type=gtk.WINDOW_TOPLEVEL)
20          self.root.set_title("CellRenderer Example")
21          self.root.connect("destroy", self.destroy_cb)
22          # Get the model and attach it to the view
23          self.mdl = Store.get_model()
24          self.view = Display.make_view( self.mdl )
25          # Add our view into the main window
26          self.root.add(self.view)
27          self.root.show_all()
28          return
29      def destroy_cb(self, *kw):
30          """ Destroy callback to shutdown the app """
31          gtk.main_quit()
32          return
33      def run(self):
34          """ run is called to set off the GTK mainloop """
35          gtk.main()
36          return
37
38  class InfoModel:
39      """ The model class holds the information we want to display """
40      def __init__(self):
41          """ Sets up and populates our gtk.TreeStore """
42          self.tree_store = gtk.TreeStore( gobject.TYPE_STRING,
43                                           gobject.TYPE_BOOLEAN )
44          # places the global people data into the list
45          # we form a simple tree.
46          for item in tasks.keys():
47              parent = self.tree_store.append( None, (item, None) )
48              self.tree_store.append( parent, (tasks[item],None) )
49          return
50      def get_model(self):
51          """ Returns the model """
52          if self.tree_store:
53              return self.tree_store
54          else:
55              return None
56
57  class DisplayModel:
58      """ Displays the Info_Model model in a view """
59      def make_view( self, model ):
60          """ Form a view for the Tree Model """
61          self.view = gtk.TreeView( model )
62          # setup the text cell renderer and allows these
63          # cells to be edited.
64          self.renderer = gtk.CellRendererText()
65          self.renderer.set_property( 'editable', True )
66          self.renderer.connect( 'edited', self.col0_edited_cb, model )
67
68          # The toggle cellrenderer is setup and we allow it to be
69          # changed (toggled) by the user.
70          self.renderer1 = gtk.CellRendererToggle()
71          self.renderer1.set_property('activatable', True)
72          self.renderer1.connect( 'toggled', self.col1_toggled_cb, model ) ↩
      73
74          # Connect column0 of the display with column 0 in our list model ↩
      75          # The renderer will then display whatever is in column 0 of
76          # our model .
77          self.column0 = gtk.TreeViewColumn("Name", self.renderer, text=0) ↩
      78
79          # The columns active state is attached to the second column
80          # in the model.  So when the model says True then the button
81          # will show as active e.g on.
82          self.column1 = gtk.TreeViewColumn("Complete", self.renderer1 )
```

```
 83            self.column1.add_attribute( self.renderer1, "active", 1)
 84            self.view.append_column( self.column0 )
 85            self.view.append_column( self.column1 )
 86            return self.view
 87        def col0_edited_cb( self, cell, path, new_text, model ):
 88            """
 89            Called when a text cell is edited.  It puts the new text
 90            in the model so that it is displayed properly.
 91            """
 92            print "Change '%s' to '%s'" % (model[path][0], new_text)
 93            model[path][0] = new_text
 94            return
 95        def col1_toggled_cb( self, cell, path, model ):
 96            """
 97            Sets the toggled state on the toggle button to true or false.
 98            """
 99            model[path][1] = not model[path][1]
100            print "Toggle '%s' to: %s" % (model[path][0], model[path][1],)
101            return
102
103    if __name__ == '__main__':
104        Store = InfoModel()
105        Display = DisplayModel()
106        myGUI = GUI_Controller()
107        myGUI.run()
```

The program provides editable cells in the first column and activatable cells in the second column. Lines 64-66 create an editable `CellRendererText` and connect the "edited" signal to the col0_edited_cb() callback (lines 87-94) that changes the appropriate row column value in the `TreeStore`. Likewise lines 70-72 create an activatable `CellRendererToggle` and connect the "toggled" signal to the col1_toggled_cb() callback (lines 95-101) to change the appropriate row value. When an editable or activatable cell is changed, a message is printed to indicate what the change was.

Figure 14.6 illustrates the cellrenderer.py program in operation.

Figure 14.6 Editable and Activatable Cells



14.5 TreeViewColumns

14.5.1 Creating TreeViewColumns

A `TreeViewColumn` is created using the constructor:

```
treeviewcolumn = gtk.TreeViewColumn(title=None, cell_renderer=None, ...)
```

where *title* is the string to be used as the column header label, and *cell_renderer* is the first CellRenderer to pack in the column. Additional arguments that are passed to the constructor are keyword values (in the format attribute=column) that set attributes on *cell_renderer*. For example:

```
treeviewcolumn = gtk.TreeViewColumn('States', cell, text=0, foreground=1)
```

creates a TreeViewColumn with the CellRendererText cell retrieving its text from the first column of the tree model and the text color from the second column.

14.5.2 Managing CellRenderers

A CellRenderer can be added to a TreeViewColumn using one of the methods:

```
treeviewcolumn.pack_start(cell, expand)
treeviewcolumn.pack_end(cell, expand)
```

pack_start() and pack_end() add *cell* to the start or end, respectively, of the TreeViewColumn. If *expand* is TRUE, *cell* will share in any available extra space allocated by the TreeViewColumn.

The get_cell_renderers() method:

```
cell_list = treeviewcolumn.get_cell_renderers()
```

returns a list of all the CellRenderers in the column.

The clear() method removes all the CellRenderer attributes from the TreeViewColumn:

```
treeviewcolumn.clear()
```

There are a large number of other methods available for a TreeViewColumn - mostly dealing with setting and getting properties. See the PyGTK Reference Manual for more information on the TreeViewColumn properties. The capability of using the built-in sorting facility is set using the method:

```
treeviewcolumn.set_sort_column_id(sort_column_id)
```

sets *sort_column_id* as the tree model sort column ID to use when sorting the TreeView display. This method also sets the "clickable" property of the column that allows the user to click on the column header to activate the sorting. When the user clicks on the column header, the TreeViewColumn sort column ID is set as the TreeModel sort column ID and the TreeModel rows are resorted using the associated sort comparison function. The automatic sorting facility also toggles the sort order of the column and manages the display of the sort indicator. See Section 14.2.9 for more information on sort column IDs and sort comparison functions. Typically when using a ListStore or TreeStore the default sort column ID (i.e. the column index) of the TreeModel column associated with the TreeViewColumn is set as the TreeViewColumn sort column ID.

If you use the TreeViewColumn headers for sorting by using the set_sort_column_id() method, you don't need to use the TreeSortable set_sort_column_id() method.

You can track the sorting operations or use a header click for your own purposes by connecting to the "clicked" signal of the TreeView column. The callback function should be defined as:

```
def callback(treeviewcolumn, user_data, ...)
```

14.6 Manipulating TreeViews

14.6.1 Managing Columns

The TreeViewColumns in a TreeView can be retrieved singly or as a list using the methods:

```
treeviewcolumn = treeview.get_column(n)
columnlist = treeview.get_columns()
```

where *n* is the index (starting from 0) of the column to retrieve. A column can be removed using the method:

```
treeview.remove_column(column)
```

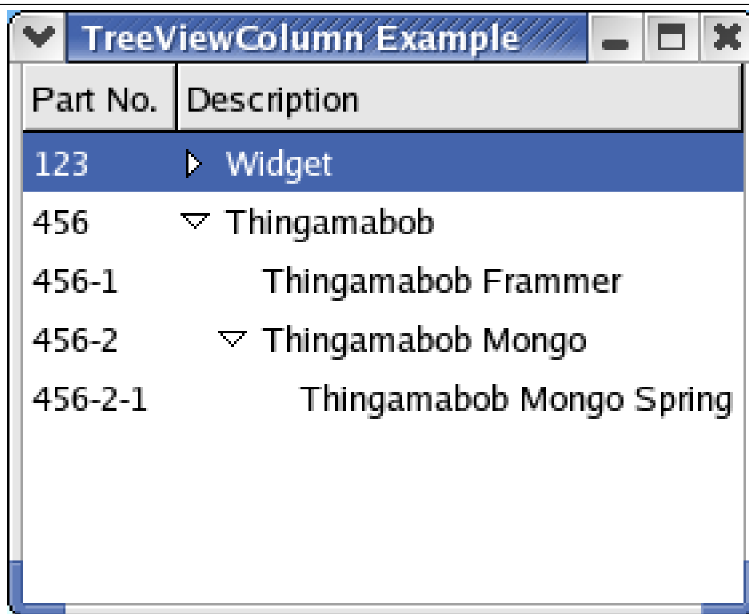
where *column* is a `TreeViewColumn` in *treeview*.

Rows that have child rows are displayed in the `TreeView` with an expander arrow (see Figure 14.3) that the user clicks on to hide or reveal the child row(s). The column that the expander arrow is displayed in can be changed using the method:

```
treeview.set_expander_column(column)
```

where *column* is a `TreeViewColumn` in treeview. This method is useful when you want the first column to not indent. For example, Figure 14.7 illustrates the expander arrow in the second column:

Figure 14.7 Expander Arrow in Second Column



14.6.2 Expanding and Collapsing Child Rows

All the rows displayed in a `TreeView` can be programmatically expanded or collapsed using the following methods:

```
treeview.expand_all()
treeview.collapse_all()
```

These methods are useful if you want to initialize the `TreeView` display to a known state. Individual rows can be expanded or collapsed using:

```
treeview.expand_row(path, open_all)
treeview.collapse_row(path)
```

where *path* is the tree path to a row in treeview, and if *open_all* is `TRUE` all descendant rows of *path* are expanded; otherwise just the immediate children are expanded.

You can determine if a row is expanded using the method:

```
is_expanded = treeview.row_expanded(path)
```

14.7 TreeView Signals

`TreeViews` emit a large number of signals that you can use to track changes in the view of the model. The signals generally fall into the following categories:

- expanding and collapsing rows: "row-collapsed", "row-expanded", "test-collapse-row", "test-expand-row" and "expand-collapse-cursor-row"

- the cursor: "cursor-changed", "expand-collapse-cursor-row", "move-cursor", "select-cursor-parent", "select-cursor-row" and "toggle-cursor-row"
- selection: "select-all", "select-cursor-parent", "select-cursor-row" and "unselect-all".
- miscellaneous: "columns-changed", "row-activated", "set-scroll-adjustments" and "start-interactive-search".

The "test-collapse-row" and "test-expand-row" signals are emitted before a row is collapsed or expanded. The return value from your callback can cancel or allow the operation - `TRUE` to allow and `FALSE` to cancel.

```
def callback(treeview, iter, path, user_data)
```

where *iter* is a `TreeIter` and *path* is a tree path pointing at the row and *user_data* is the data specified in the connect() method.

The "row-activated" signal is emitted when a double click occurs on a row or a non-editable row is selected and one of the keys: **Space**, Shift+Space, **Return** or **Enter** is pressed.

The rest of the signals are emitted after the `TreeView` has changed. The cursor is the row outlined by a box. In most cases moving the cursor also moves the selection. The cursor can be moved independently by Control+Down or Control+Up and various other key combinations.

See the PyGTK Reference Manual for more information on the `TreeView` signals.

14.8 TreeSelections

14.8.1 Getting the TreeSelection

`TreeSelections` are objects that manage selections in a `TreeView`. When a `TreeView` is created a `TreeSelection` is automatically created as well. The `TreeSelection` can be retrieved from the `TreeView` using the method:

```
treeselection = treeview.get_selection()
```

You can retrieve the `TreeView` associated with a `TreeSelection` by calling the method:

```
treeview = treeselection.get_treeview()
```

14.8.2 TreeSelection Modes

The `TreeSelection` supports the following selection modes:

`gtk.SELECTION_NONE` No selection is allowed.

`gtk.SELECTION_SINGLE` A single selection is allowed by clicking.

`gtk.SELECTION_BROWSE` A single selection allowed by browsing with the pointer.

`gtk.SELECTION_MULTIPLE` Multiple items can be selected at once.

You can retrieve the current selection mode by calling the method:

```
mode = treeselection.get_mode()
```

The mode can be set using:

```
treeselection.set_mode(mode)
```

where *mode* is one of the above selection modes.

14.8.3 Retrieving the Selection

The method to use to retrieve the selection depends on the current selection mode. If the selection mode is `gtk.SELECTION_SINGLE` or `gtk.SELECTION_BROWSE`, you should use the following method:

```
(model, iter) = treeselection.get_selected()
```

that returns a 2-tuple containing *model*, the `TreeModel` used by the `TreeView` associated with *treeselection* and *iter*, a `TreeIter` pointing at the selected row. If no row is selected then *iter* is `None`. If the selection mode is `gtk.SELECTION_MULTIPLE` a TypeError exception is raised.

If you have a `TreeView` using the `gtk.SELECTION_MULTIPLE` selection mode then you should use the method:

```
(model, pathlist) = treeselection.get_selected_rows()
```

that returns a 2-tuple containing the tree model and a list of the tree paths of the selected rows. This method is not available in PyGTK 2.0 so you'll have to use a helper function to retrieve the list by using:

```
treeselection.selected_foreach(func, data=None)
```

where *func* is a function that is called on each selected row with *data*. The signature of *func* is:

```
def func(model, path, iter, data)
```

where *model* is the `TreeModel`, *path* is the tree path of the selected row and *iter* is a `TreeIter` pointing at the selected row.

This method can be used to simulate the get_selected_row() method as follows:

```
...
def foreach_cb(model, path, iter, pathlist):
    list.append(path)
...
def my_get_selected_rows(treeselection):
    pathlist = []
    treeselection.selected_foreach(foreach_cb, pathlist)
    model = sel.get_treeview().get_model()
    return (model, pathlist)
...
```

The selected_foreach() method cannot be used to modify the tree model or the selection though you can change the data in the rows.

14.8.4 Using a TreeSelection Function

If you want ultimate control over row selection you can set a function to be called before a row is selected or unselected by using the method:

```
treeselection.set_select_function(func, data)
```

where *func* is a callback function and *data* is user data to be passed to *func* when it is called. *func* has the signature:

```
def func(selection, model, path, is_selected, user_data)
```

where *selection* is the `TreeSelection`, *model* is the `TreeModel` used with the `TreeView` associated with *selection*, *path* is the tree path of the selected row, *is_selected* is `TRUE` if the row is currently selected and *user_data* is *data*. *func* should return `TRUE` if the row's selection status should be toggled.

Setting a select function is useful if:

- you want to control the selection or unselection of a row based on some additional context information. You will need to indicate in some way that the selection change can't be made and perhaps why. For example, you can visually differentiate the row or pop up a `MessageDialog`.
- you need to maintain your own list of selected or unselected rows though this can also be done by connecting to the "changed" signal but with more effort.
- you want to do some additional processing before a row is selected or unselected. For example change the look of the row or modify the row data.

14.8.5 Selecting and Unselecting Rows

You can change the selection programmatically using the following methods:

```
treeselection.select_path(path)
treeselection.unselect_path(path)

treeselection.select_iter(iter)
treeselection.unselect_iter(iter)
```

These methods select or unselect a single row that is specified by either *path*, a tree path or *iter*, a `TreeIter` pointing at the row. The following methods select or unselect several rows at once:

```
treeselection.select_all()
treeselection.unselect_all()

treeselection.select_range(start_path, end_path)
treeselection.unselect_range(start_path, end_path)
```

The select_all() method requires that the selection mode be `gtk.SELECTION_MULTIPLE` as does the select_range() method. The unselect_all() and unselect_range() methods will function with any selection mode. Note that the unselect_all() method is not available in PyGTK 2.0

You can check if a row is selected by using one of the methods:

```
result = treeselection.path_is_selected(path)
result = treeselection.iter_is_selected(iter)
```

that return `TRUE` if the row specified by *path* or *iter* is currently selected. You can retrieve a count of the number of selected rows using the method:

```
count = treeselection.count_selected_rows()
```

This method is not available in PyGTK 2.0 so you'll have to simulate it using the selected_foreach() method similar to the simulation of the get_selected_rows() method in Section 21.2. For example:

```
...
def foreach_cb(model, path, iter, counter):
    counter[0] += 1
...
def my_count_selected_rows(treeselection):
    counter = [0]
    treeselection.selected_foreach(foreach_cb, counter)
    return counter[0]
...
```

14.9 TreeView Drag and Drop

14.9.1 Drag and Drop Reordering

Reordering of the `TreeView` rows (and the underlying tree model rows is enabled by using the set_reorderable() method mentioned above. The *set_reorderable*() method sets the "reorderable" property to the specified value and enables or disables the internal drag and drop of `TreeView` rows. When the "reorderable" property is `TRUE` a user can drag `TreeView` rows and drop them at a new location. This action causes the underlying `TreeModel` rows to be rearranged to match. Drag and drop reordering of rows only works with unsorted stores.

14.9.2 External Drag and Drop

If you want to control the drag and drop or deal with drag and drop from external sources, you'll have to enable and control the drag and drop using the following methods:

```
treeview.enable_model_drag_source(start_button_mask, targets, actions)
treeview.enable_model_drag_dest(targets, actions)
```

These methods enable using rows as a drag source and a drop site respectively. *start_button_mask* is a modifier mask (see the gtk.gtk Constants reference in the PyGTK Reference Manual) that specifies the buttons or keys that must be pressed to start the drag operation. *targets* is a list of 3-tuples that describe the target information that can be given or received. For a drag and drop to succeed at least one of the targets must match in the drag source and drag destination (e.g. the "STRING" target). Each target 3-tuple contains the target name, flags (a combination of `gtk.TARGET_SAME_APP` and `gtk.TARGET_SAME_WIDGET` or neither) and a unique int identifier. *actions* describes what the result of the operation should be:

gtk.gdk.ACTION_DEFAULT, gtk.gdk.ACTION_COPY Copy the data.

gtk.gdk.ACTION_MOVE Move the data, i.e. first copy it, then delete it from the source using the `DELETE` target of the X selection protocol.

gtk.gdk.ACTION_LINK Add a link to the data. Note that this is only useful if source and destination agree on what it means.

gtk.gdk.ACTION_PRIVATE Special action which tells the source that the destination will do something that the source doesn't understand.

gtk.gdk.ACTION_ASK Ask the user what to do with the data.

For example to set up a drag drop destination:

```
treeview.enable_model_drag_dest([('text/plain', 0, 0)],
                gtk.gdk.ACTION_DEFAULT | gtk.gdk.ACTION_MOVE)
```

Then you'll have to handle the `Widget` "drag-data-received" signal to receive that dropped data - perhaps replacing the data in the row it was dropped on. The signature for the callback for the "drag-data-received" signal is:

```
def callback(widget, drag_context, x, y, selection_data, info, timestamp)
```

where *widget* is the `TreeView`, *drag_context* is a `DragContext` containing the context of the selection, *x* and *y* are the position where the drop occurred, *selection_data* is the `SelectionData` containing the data, *info* is the ID integer of the type, *timestamp* is the time when the drop occurred. The row can be identified by calling the method:

```
drop_info = treeview.get_dest_row_at_pos(x, y)
```

where (*x*, *y*) is the position passed to the callback function and *drop_info* is a 2-tuple containing the path of a row and a position constant indicating where the drop is with respect to the row: `gtk.TREE_VIEW_DROP_BEFORE`, `gtk.TREE_VIEW_DROP_AFTER`, `gtk.TREE_VIEW_DROP_INTO_OR_BEFORE` or `gtk.TREE_VIEW_DROP_INTO_OR_AFTER`. The callback function could be something like:

```
treeview.enable_model_drag_dest([('text/plain', 0, 0)],
                gtk.gdk.ACTION_DEFAULT | gtk.gdk.ACTION_MOVE)
treeview.connect("drag-data-received", drag_data_received_cb)
...
...
def drag_data_received_cb(treeview, context, x, y, selection, info, timestamp):
    drop_info = treeview.get_dest_row_at_pos(x, y)
    if drop_info:
        model = treeview.get_model()
        path, position = drop_info
        data = selection.data
        # do something with the data and the model
        ...
    return
...
```

If a row is being used as a drag source it must handle the `Widget` "drag-data-get" signal that populates a selection with the data to be passed back to the drag drop destination with a callback function with the signature:

```
def callback(widget, drag_context, selection_data, info, timestamp)
```

The parameters to *callback* are similar to those of the "drag-data-received" callback function. Since the callback is not passed a tree path or any easy way of retrieving information about the row being dragged, we assume that the row being dragged is selected and the selection mode is `gtk.SELECTION_SINGLE` or `gtk.SELECTION_BROWSE` so we can retrieve the row by getting the `TreeSelection` and retrieving the tree model and `TreeIter` pointing at the row. For example, text from a row could be passed in the drag drop by:

```
...
treestore = gtk.TreeStore(str, str)
...
treeview.enable_model_drag_source(gtk.gdk.BUTTON1_MASK,
                [('text/plain', 0, 0)],
                gtk.gdk.ACTION_DEFAULT | gtk.gdk.ACTION_MOVE)
treeview.connect("drag-data-get", drag_data_get_cb)
...
...
def drag_data_get_cb(treeview, context, selection, info, timestamp):
    treeselection = treeview.get_selection()
    model, iter = treeselection.get_selected()
    text = model.get_value(iter, 1)
    selection.set('text/plain', 8, text)
    return
...
```

The `TreeView` can be disabled as a drag source and drop destination by using the methods:

```
treeview.unset_rows_drag_source()
treeview.unset_rows_drag_dest()
```

14.9.3 TreeView Drag and Drop Example

A simple example program is needed to pull together the pieces of code described above. This example (treeviewdnd.py) is a list that URLs can be dragged from and dropped on. Also the URLs in the list can be reordered by dragging and dropping within the `TreeView`. A couple of buttons are provided to clear the list and to clear a selected item.

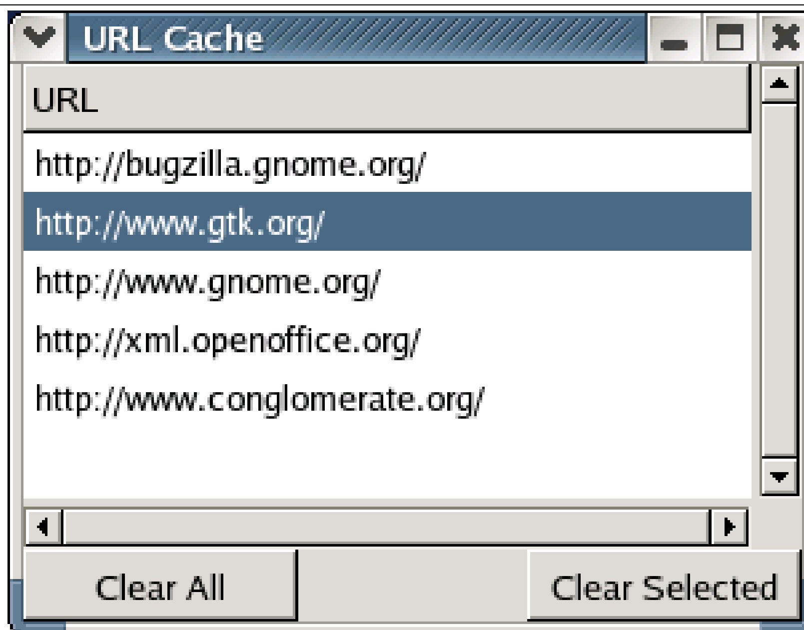
```
 1  #!/usr/bin/env python
 2
 3  # example treeviewdnd.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class TreeViewDnDExample:
10
11      TARGETS = [
12          ('MY_TREE_MODEL_ROW', gtk.TARGET_SAME_WIDGET, 0),
13          ('text/plain', 0, 1),
14          ('TEXT', 0, 2),
15          ('STRING', 0, 3),
16          ]
17      # close the window and quit
18      def delete_event(self, widget, event, data=None):
19          gtk.main_quit()
20          return False
21
22      def clear_selected(self, button):
23          selection = self.treeview.get_selection()
24          model, iter = selection.get_selected()
25          if iter:
26              model.remove(iter)
27          return
28
```

```
29      def __init__(self):
30          # Create a new window
31          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
32
33          self.window.set_title("URL Cache")
34
35          self.window.set_size_request(200, 200)
36
37          self.window.connect("delete_event", self.delete_event)
38
39          self.scrolledwindow = gtk.ScrolledWindow()
40          self.vbox = gtk.VBox()
41          self.hbox = gtk.HButtonBox()
42          self.vbox.pack_start(self.scrolledwindow, True)
43          self.vbox.pack_start(self.hbox, False)
44          self.b0 = gtk.Button('Clear All')
45          self.b1 = gtk.Button('Clear Selected')
46          self.hbox.pack_start(self.b0)
47          self.hbox.pack_start(self.b1)
48
49          # create a liststore with one string column to use as the model
50          self.liststore = gtk.ListStore(str)
51
52          # create the TreeView using liststore
53          self.treeview = gtk.TreeView(self.liststore)
54
55         # create a CellRenderer to render the data
56          self.cell = gtk.CellRendererText()
57
58          # create the TreeViewColumns to display the data
59          self.tvcolumn = gtk.TreeViewColumn('URL', self.cell, text=0)
60
61          # add columns to treeview
62          self.treeview.append_column(self.tvcolumn)
63          self.b0.connect_object('clicked', gtk.ListStore.clear, self. ↩
   liststore)
64          self.b1.connect('clicked', self.clear_selected)
65          # make treeview searchable
66          self.treeview.set_search_column(0)
67
68          # Allow sorting on the column
69          self.tvcolumn.set_sort_column_id(0)
70
71          # Allow enable drag and drop of rows including row move
72          self.treeview.enable_model_drag_source( gtk.gdk.BUTTON1_MASK,
73                                                  self.TARGETS,
74                                                  gtk.gdk.ACTION_DEFAULT|
75                                                  gtk.gdk.ACTION_MOVE)
76          self.treeview.enable_model_drag_dest(self.TARGETS,
77                                               gtk.gdk.ACTION_DEFAULT)
78
79          self.treeview.connect("drag_data_get", self.drag_data_get_data)
80          self.treeview.connect("drag_data_received",
81                                self.drag_data_received_data)
82
83          self.scrolledwindow.add(self.treeview)
84          self.window.add(self.vbox)
85          self.window.show_all()
86
87      def drag_data_get_data(self, treeview, context, selection, target_id,
88                             etime):
89          treeselection = treeview.get_selection()
90          model, iter = treeselection.get_selected()
91          data = model.get_value(iter, 0)
```

```
 92             selection.set(selection.target, 8, data)
 93 
 94         def drag_data_received_data(self, treeview, context, x, y, selection,
 95                                     info, etime):
 96             model = treeview.get_model()
 97             data = selection.data
 98             drop_info = treeview.get_dest_row_at_pos(x, y)
 99             if drop_info:
100                 path, position = drop_info
101                 iter = model.get_iter(path)
102                 if (position == gtk.TREE_VIEW_DROP_BEFORE
103                     or position == gtk.TREE_VIEW_DROP_INTO_OR_BEFORE):
104                     model.insert_before(iter, [data])
105                 else:
106                     model.insert_after(iter, [data])
107             else:
108                 model.append([data])
109             if context.action == gtk.gdk.ACTION_MOVE:
110                 context.finish(True, True, etime)
111             return
112 
113     def main():
114         gtk.main()
115 
116     if __name__ == "__main__":
117         treeviewdndex = TreeViewDnDExample()
118         main()
```

The result of running the example program treeviewdnd.py is illustrated in Figure 14.8:

Figure 14.8 TreeView Drag and Drop Example



The key to allowing both external drag and drop and internal row reordering is the organization of the targets (the `TARGETS` attribute - line 11). An application specific target (`MY_TREE_MODEL_ROW`) is created and used to indicate a drag and drop within the `TreeView` by setting the `gtk.TARGET_SAME_WIDGET` flag. By setting this as the first target the drag destination will attempt to match it first with the drag source targets. Next the source drag actions must include `gtk.gdk.ACTION_MOVE` and `gtk.gdk.ACTION_DEFAULT` (see lines 72-75). When the destination is receiving the data from the source, if the `DragContext` action is `gtk.gdk.ACTION_MOVE` the source is told to delete the data (in this case the row) by calling the `DragContext` method finish() (see lines 109-110). The `TreeView` provides a number of internal functions that we are leveraging to drag, drop and delete the data.

14.10 TreeModelSort and TreeModelFilter

The TreeModelSort and TreeModelFilter objects are tree models that interpose between the base TreeModel (either a TreeStore or a ListStore) and the TreeView to provide a modified model while still retaining the original structure of the base model. These interposing models implement the TreeModel and TreeSortable interfaces but do not provide any methods for inserting or removing rows in the model; you have to insert or remove rows from the underlying store. The TreeModelSort provides a model where the rows are always sorted while the TreeModelFilter provides a model containing a subset of the rows of the base model.

These models can be chained to an arbitrary length if desired; i.e a TreeModelFilter could have a child TreeModelSort that could have a child TreeModelFilter, and so on. As long as there is a TreeStore or ListStore as the anchor of the chain it should just work. In PyGTK 2.0 and 2.2 the TreeModelSort and TreeModelFilter objects do not support the TreeModel Python mapping protocol.

14.10.1 TreeModelSort

The TreeModelSort maintains a sorted model of the child model specified in its constructor. The main use of a TreeModelSort is to provide multiple views of a model that can be sorted differently. If you have multiple views of the same model then any sorting activity is reflected in all the views. By using the TreeModelSort the base store is left in its original unsorted state and the sort models absorb all the sorting activity. To create a TreeModelSort use the constructor:

```
treemodelsort = gtk.TreeModelSort(child_model)
```

where *child_model* is a TreeModel. Most of the methods of a TreeModelSort deal with converting tree paths and TreeIters from the child model to the sorted model and back:

```
sorted_path = treemodelsort.convert_child_path_to_path(child_path)
child_path = treemodelsort.convert_path_to_child_path(sorted_path)
```

These path conversion methods return None if the given path cannot be converted to a path in the sorted model or the child model respectively. The TreeIter conversion methods are:

```
sorted_iter = treemodelsort.convert_child_iter_to_iter(sorted_iter, child_iter)
child_iter = treemodelsort.convert_iter_to_child_iter(child_iter, sorted_iter)
```

The TreeIter conversion methods duplicate the converted argument (its both the return value and the first argument) due to backward compatibility issues; you should set the first arguments to None and just use the return value. For example:

```
sorted_iter = treemodelsort.convert_child_iter_to_iter(None, child_iter)
child_iter = treemodelsort.convert_iter_to_child_iter(None, sorted_iter)
```

Like the path conversion methods, these methods return None if the given TreeIter cannot be converted.

You can retrieve the child TreeModel using the get_model() method.

A simple example program using TreeModelSort objects is treemodelsort.py. Figure 14.9 illustrates the result of running the program and adding six rows:

Figure 14.9 TreeModelSort Example

TreeModelSort Exam|

0-1000	0-1000000	-10000-100(
110	402699	5367
158	803371	7708
400	928810	-1066
478	948689	-8574
982	105496	3904
994	813492	-151

Add a Row

TreeModelSort Exam|

0-1000	0-1000000	-10000-10000
478	948689	-8574
400	928810	-1066
994	813492	-151
982	105496	3904
110	402699	5367
158	803371	7708

Add a Row

TreeModelSort Exam|

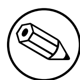
0-1000	0-1000000	-10000-100(
982	105496	3904
110	402699	5367
158	803371	7708
994	813492	-151
400	928810	-1066
478	948689	-8574

Add a Row

Each of the columns in the windows can be clicked to change the sort order independent of the other windows. When the "Add a Row" button is clicked a new row is added to the base `ListStore` and the new row is displayed in each `TreeView` as the selected row.

14.10.2 TreeModelFilter

NOTE



The `TreeModelFilter` is available in PyGTK 2.4 and above.

A `TreeModelFilter` object provides several ways of modifying the view of the base `TreeModel` including:

- displaying a subset of the rows in the child model either based on boolean data in a "visible column", or based on the boolean return value of a "visible function" that takes the child model, a

`TreeIter` pointing at a row in the child model and user data. In both cases if the boolean value is `TRUE` the row will be displayed; otherwise, the row will be hidden.

- using a virtual root node to provide a view of a subtree of the children of a row in the child model. This only makes sense if the underlying store is a `TreeStore`.
- synthesizing the columns and data of a model based on the data in the child model. For example, you can provide a column where the data is calculated from data in several child model columns.

A `TreeModelFilter` object is created using the `TreeModel` method:

```
treemodelfilter = treemodel.filter_new(root=None)
```

where *root* is a tree path in *treemodel* specifying the virtual root for the model or `None` if the root node of *treemodel* is to be used.

By setting a "virtual root" when creating the `TreeModelFilter`, you can limit the model view to the child rows of "root" row in the child model hierarchy. This, of course is only useful when the child model is based on a `TreeStore`. For example, you might want to provide a view of the parts list that makes up a CDROM drive separate from the full parts list of a computer.

The visibility modes are mutually exclusive and can only be set once i.e. once a visibility function or column is set it cannot be changed and the alternative mode cannot be set. The simplest visibility mode extracts a boolean value from a column in the child model to determine if the row should be displayed. The visibility columns is set using:

```
treemodelfilter.set_visible_column(column)
```

where *column* is the number of the column in the child `TreeModel` to extract the boolean values from. For example, the following code fragment uses the values in the third column to set the visibility of the rows:

```
...
treestore = gtk.TreeStore(str, str, "gboolean")
...
modelfilter = treestore.filter_new()
modelfilter.set_visible_column(2)
...
```

Thus any rows in *treestore* that have a value of `TRUE` in the third column will be displayed.

If you have more complicated visibility criteria setting a visibility function should provide sufficient power:

```
treemodelfilter.set_visible_func(func, data=None)
```

where *func* is the function called for each child model row to determine if it should be displayed and *data* is user data passed to *func*. *func* should return `TRUE` if the row should be displayed. The signature of *func* is:

```
def func(model, iter, user_data)
```

where *model* is the child `TreeModel`, *iter* is a `TreeIter` pointing at a row in *model* and *user_data* is the passed in *data*.

If you make a change to the visibility criteria you should call:

```
treemodelfilter.refilter()
```

to force a refiltering of the child model rows.

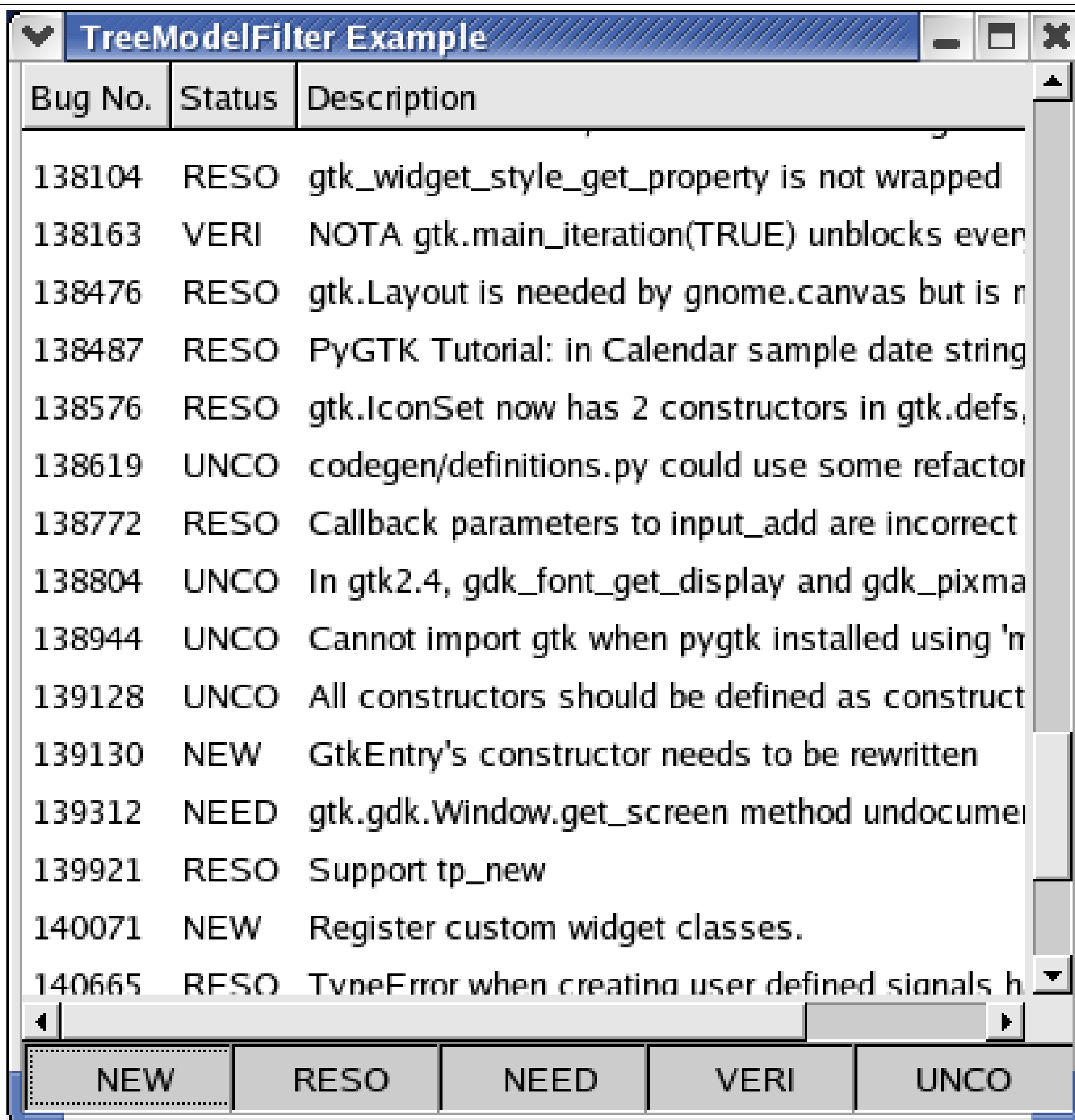
For example, the following code fragment illustrates a `TreeModelFilter` that displays rows based on a comparison between the value in the third column and the contents of the user data:

```
...
def match_type(model, iter, udata):
    value = model.get_value(iter, 2)
    return value in udata
...
show_vals = ['OPEN', 'NEW', 'RESO']
liststore = gtk.ListStore(str, str, str)
```

```
...
modelfilter = liststore.filter_new()
modelfilter.set_visible_func(match_type, show_vals)
...
```

The program treemodelfilter.py illustrates the use of the set_visible_func() method. Figure 14.10 shows the result of running the program.

Figure 14.10 TreeModelFilter Visibility Example



By toggling the buttons at the bottom the contents of the `TreeView` are changed to display only the rows that match one of the active buttons.

A modify function gives you another level of control over the `TreeView` display to the point where you can synthesize one or more (or even all) columns that are represented by the `TreeModelFilter`. You still have to use a base child model that is a `TreeStore` or `ListStore` to determine the number of rows and the hierarchy but the columns can be anything you specify in the method:

```
treemodelfilter.set_modify_func(types, func, data=None)
```

where *types* is a sequence (list or tuple) specifying the column types being represented, *func* is a function called to return the value for a row and column and *data* is an argument to be passed to *func*. The signature of *func* is:

```
def func(model, iter, column, user_data)
```

where *model* is the TreeModelFilter, *iter* is a TreeIter that points to a row in model, *column* is the number of the column that a value is needed for and *user_data* is the parameter *data*. *func* must return a value matching the type for *column*.

A modify function is useful where you want to provide a column of data that needs to be generated using the data in the child model columns. For example if you had a column containing birth dates and wanted to provide a column displaying ages, a modify function could generate the age information using the birth date and the current date. Another example would be to decide what image to display based on some analysis of the data (say, a filename) in a column. This effect can also be achieved using the TreeViewColumn set_cell_data_func() method.

Usually within the modify function, you will have to convert the TreeModelFilter TreeIter to a TreeIter in the child model using:

```
child_iter = treemodelfilter.convert_iter_to_child_iter(filter_iter)
```

Of course, you'll also need to retrieve the child model using:

```
child_model = treemodelfilter.get_model()
```

These give you access to the child model row and its values for generating the value for the specified TreeModelFilter row and column. There's also a method to convert a child TreeIter to a filter model TreeIter and methods to convert filter model paths to and from child tree paths:

```
filter_iter = treemodelfilter.convert_child_iter_to_iter(child_iter)

child_path = treemodelfilter.convert_path_to_child_path(filter_path)
filter_path = treemodelfilter.convert_child_path_to_path(child_path)
```

Of course, you can combine the visibility modes and the modify function to both filter rows and synthesize columns. To get even more control over the view you would have to use a custom TreeModel.

14.11 The Generic TreeModel

When you find that the standard TreeModels are not sufficiently powerful for your application needs, you can use the GenericTreeModel to build your own custom TreeModel in Python. Creating a GenericTreeModel may be useful when there are performance issues with the standard TreeStore and ListStore objects or when you want to directly interface to an external data source (say, a database or filesystem) to save copying the data into and out of a TreeStore or ListStore.

14.11.1 GenericTreeModel Overview

With the GenericTreeModel you build and manage your data model and provide external access though the standard TreeModel interface by defining a set of class methods. PyGTK implements the TreeModel interface and arranges for your TreeModel methods to be called to provide the actual model data.

The implementation details of your model should be kept completely hidden from the external application. This means that the way that your model identifies, stores and retrieves data is unknown to the application. In general the only information that is saved outside your GenericTreeModel are the row references that are wrapped by the external TreeIters. And these references are not visible to the application.

Let's examine in detail the GenericTreeModel interface that you have to provide.

14.11.2 The GenericTreeModel Interface

The GenericTreeModel interface consists of the following methods that must be implemented in your custom tree model: def on_get_flags(*self*)

def on_get_n_columns(*self*)
def on_get_column_type(*self*, *index*)
def on_get_iter(*self*, *path*)
def on_get_path(*self*, *rowref*)
def on_get_value(*self*, *rowref*, *column*)

def on_iter_next(*self*, *rowref*)
def on_iter_children(*self*, *parent*)
def on_iter_has_child(*self*, *rowref*)
def on_iter_n_children(*self*, *rowref*)
def on_iter_nth_child(*self*, *parent*, *n*)
def on_iter_parent(*self*, *child*)
You should note that these methods support all of the `TreeModel` interface including: def get_flags()
def get_n_columns()
def get_column_type(*index*)
def get_iter(*path*)
def get_iter_from_string(*path_string*)
def get_string_from_iter(*iter*)
def get_iter_root()
def get_iter_first()
def get_path(*iter*)
def get_value(*iter*, *column*)
def iter_next(*iter*)
def iter_children(*parent*)
def iter_has_child(*iter*)
def iter_n_children(*iter*)
def iter_nth_child(*parent*, *n*)
def iter_parent(*child*)
def get(*iter*, *column*, *...*)
def foreach(*func*, *user_data*)
To illustrate the use of the `GenericTreeModel` I'll change the filelisting.py example program and show how the interface methods are created. The filelisting-gtm.py program displays the files in a folder with a pixbuf indicating if the file is a folder or not, the file name, the file size, mode and time of last change.

The on_get_flags() method should return a value that is a combination of:

gtk.TREE_MODEL_ITERS_PERSIST `TreeIter`s survive all signals emitted by the tree.

gtk.TREE_MODEL_LIST_ONLY The model is a list only, and never has children

If your model has row references that are valid over row changes (reorder, addition, deletion) then set `gtk.TREE_MODEL_ITERS_PERSIST`. Likewise if your model is a list only then set `gtk.TREE_MODEL_LIST_ONLY`. Otherwise, return 0 if your model doesn't have persistent row references and it's a tree model. For our example, the model is a list with persistent `TreeIter`s.

```
    def on_get_flags(self):
        return gtk.TREE_MODEL_LIST_ONLY|gtk.TREE_MODEL_ITERS_PERSIST
```

The on_get_n_columns() method should return the number of columns that your model exports to the application. Our example maintains a list of column types so we return the length of the list:

```
class FileListModel(gtk.GenericTreeModel):
    ...
    column_types = (gtk.gdk.Pixbuf, str, long, str, str)
    ...
    def on_get_n_columns(self):
        return len(self.column_types)
```

The on_get_column_type() method should return the type of the column with the specified *index*. This method is usually called from a `TreeView` when its model is set. You can either create a list or tuple containing the column data type info or generate it on-the-fly. In our example:

```
    def on_get_column_type(self, n):
        return self.column_types[n]
```

The `GenericTreeModel` interface converts the Python type to a GType so the following code:

```
  flm = FileListModel()
  print flm.on_get_column_type(1), flm.get_column_type(1)
```

would print:

```
<type 'str'> <GType gchararray (64)>
```

The following methods use row references that are kept as private data in a `TreeIter`. The application can't see the row reference in a `TreeIter` so you can use any unique item you want as a row reference. For example in a model containing rows as tuples you could use the tuple id as the row reference. Another example would be to use a filename as the row reference in a model representing files in a directory. In both these cases, the row reference is unchanged by model changes so the `TreeIter`s could be flagged as persistent. The PyGTK `GenericTreeModel` application interface will extract your row references from `TreeIter`s and wrap your row references in `TreeIter`s as needed.

In the following methods `rowref` refers to an internal row reference.

The on_get_iter() method should return an rowref for the tree path specified by *path*. The tree path will always be represented using a tuple. Our example uses the file name string as the rowref. The file names are kept in a list in the model so we take the first index of the path as an index to the file name:

```
def on_get_iter(self, path):
    return self.files[path[0]]
```

You have to be consistent in your row reference usage since you'll get a row reference back in method calls from the `GenericTreeModel` methods that take `TreeIter` arguments: on_get_path(), on_get_value(), on_iter_next(), on_iter_children(), on_iter_has_child(), on_iter_n_children(), on_iter_nth_child() and on_iter_parent().

The on_get_path() method should return a tree path given a rowref. For example, continuing the above example where the file name is used as the rowref, you could define the on_get_path() method as:

```
def on_get_path(self, rowref):
    return self.files.index(rowref)
```

This method finds the index of the list containing the file name in *rowref*. It's obvious from this example that a judicious choice of row reference will make the implementation more efficient. You could, for example, use a Python dict to map *rowref* to a path.

The on_get_value() method should return the data stored at the row and column specified by *rowref* and *column*. For our example:

```
def on_get_value(self, rowref, column):
    fname = os.path.join(self.dirname, rowref)
    try:
        filestat = statcache.stat(fname)
    except OSError:
        return None
    mode = filestat.st_mode
    if column is 0:
        if stat.S_ISDIR(mode):
            return folderpb
        else:
            return filepb
    elif column is 1:
        return rowref
    elif column is 2:
        return filestat.st_size
    elif column is 3:
        return oct(stat.S_IMODE(mode))
    return time.ctime(filestat.st_mtime)
```

has to extract the associated file information and return the appropriate value depending on which column is specified.

The on_iter_next() method should return a row reference to the row (at the same level) after the row specified by *rowref*. For our example:

```
def on_iter_next(self, rowref):
    try:
        i = self.files.index(rowref)+1
```

```
            return self.files[i]
        except IndexError:
            return None
```

The index of the *rowref* file name is determined and the next file name is returned or None is returned if there is no next file.

The on_iter_children() method should return a row reference to the first child row of the row specified by *rowref*. If *rowref* is None, a reference to the first top level row is returned. If there is no child row None is returned. For our example:

```
    def on_iter_children(self, rowref):
        if rowref:
            return None
        return self.files[0]
```

Since the model is a list model only the top level (*rowref*=None) can have child rows. None is returned if *rowref* contains a file name.

The on_iter_has_child() method should return TRUE if the row specified by *rowref* has child rows; FALSE otherwise. Our example returns FALSE since no row can have a child:

```
    def on_iter_has_child(self, rowref):
        return False
```

The on_iter_n_children() method should return the number of child rows that the row specified by *rowref* has. If *rowref* is None, the number of top level rows is returned. Our example returns 0 if *rowref* is not None:

```
    def on_iter_n_children(self, rowref):
        if rowref:
            return 0
        return len(self.files)
```

The on_iter_nth_child() method should return a row reference to the nth child row of the row specified by *parent*. If *parent* is None, a reference to the nth top level row is returned. Our example returns the nth top level row reference if *parent* is None. Otherwise None is returned:

```
    def on_iter_nth_child(self, rowref, n):
        if rowref:
            return None
        try:
            return self.files[n]
        except IndexError:
            return None
```

The on_iter_parent() method should return a row reference to the parent row of the row specified by *rowref*. If *rowref* points to a top level row, None should be returned. Our example always returns None assuming that *rowref* must point to a top level row:

```
    def on_iter_parent(child):
        return None
```

This example is put together in the filelisting-gtm.py program. Figure 14.11 shows the result of running the program.

Figure 14.11 Generic TreeModel Example Program

/usr/java/j2re1.4.2_04

Name	Size	Mode	Last Changed
COPYRIGHT-jai.txt	4675	0644	Thu Jul 10 19:08:13 2003
ControlPanel.html	446	0644	Tue Feb 24 07:53:03 2004
INSTALL-jai.txt	19512	0444	Thu Jul 10 19:08:12 2003
LICENSE	14380	0444	Tue Feb 24 07:46:45 2004
LICENSE-jai.txt	10351	0444	Thu Jul 10 19:08:13 2003
README	10088	0444	Tue Feb 24 07:46:45 2004
README-jai.txt	70632	0444	Thu Jul 10 19:08:13 2003
THIRDPARTYLICENSEREADME.txt	10129	0444	Tue Feb 24 07:46:45 2004
UNINSTALL-jai	596	0444	Thu Jul 10 19:08:12 2003
Welcome.html	969	0444	Tue Feb 24 07:46:45 2004
bin	4096	0755	Wed Apr 28 13:05:48 2004
jai-1_1_2-lib-linux-i586-jre.bin	2674187	0644	Wed Apr 28 13:06:57 2004
javaws	4096	0755	Wed Apr 28 13:06:30 2004
lib	4096	0755	Wed Apr 28 13:06:30 2004
man	4096	0755	Wed Apr 28 13:05:54 2004
plugin	4096	0755	Wed Apr 28 13:05:54 2004

14.11.3 Adding and Removing Rows

The filelisting-gtm.py program calculates the list of file names while creating a `FileListModel` instance. If you want to check for new files periodically and add or remove files from the model you could either create a new `FileListModel` for the same folder or you could add methods to add and remove rows in the model. Depending on the type of model you are creating you would need to add methods similar to those in the `TreeStore` and `ListStore` models:

- insert()
- insert_before()
- insert_after()
- prepend()
- append()
- remove()
- clear()

Of course not all or any of these need to be implemented. You can create your own methods that are more closely related to your model.

Using the above example program to illustrate adding methods for removing and adding files, let's implement the methods: `def remove(`*`iter`*`)`

`def add(`*`filename`*`)`

The remove() method removes the file specified by *iter*. In addition to removing the row from the model the method also should remove the file from the folder. Of course, if the user doesn't have the permissions to remove the file then the row shouldn't be removed either. For example:

```
    def remove(self, iter):
        path = self.get_path(iter)
        pathname = self.get_pathname(path)
        try:
            if os.path.exists(pathname):
                os.remove(pathname)
            del self.files[path[0]]
            self.row_deleted(path)
        except OSError:
            pass
        return
```

The method is passed a `TreeIter` that has to be converted to a path to use to retrieve the file path using the method get_pathname(). It's possible that the file has already been removed so we check if it exists before trying to remove it. If an OSError exception is thrown during the file removal it's probably because the file is a directory or the user doesn't have sufficient privilege to remove it. Finally, the file is removed and the "row-deleted" signal is emitted from the rows_deleted() method. The "file-deleted" signal notifies the `TreeView`s using the model that the model has changed so that they can update their internal state and display the revised model.

The add() method needs to create a file with the given name in the current folder. If the file was created its name is added to the list of files in the model. For example:

```
    def add(self, filename):
        pathname = os.path.join(self.dirname, filename)
        if os.path.exists(pathname):
            return
        try:
            fd = file(pathname, 'w')
            fd.close()
            self.dir_ctime = os.stat(self.dirname).st_ctime
            files = self.files[1:] + [filename]
            files.sort()
            self.files = ['..'] + files
            path = (self.files.index(filename),)
            iter = self.get_iter(path)
            self.row_inserted(path, iter)
        except OSError:
            pass
        return
```

This simple example makes sure that the file doesn't exist then tries to open the file for writing. If successful, the file is closed and the file name sorted into the list of files. The path and `TreeIter` for the added file row are retrieved to use in the row_inserted() method that emits the "row-inserted" signal. The "row-inserted" signal is used to notify the `TreeView`s using the model that they need to update their internal state and revise their display.

The other methods mentioned above (for example, append and prepend) don't make sense for the example since the model keeps the file list sorted.

Other methods that may be worth implementing in a `TreeModel` subclassing the `GenericTreeModel` are:

- set_value()
- reorder()
- swap()
- move_after()
- move_before()

Implementing these methods is similar to the above methods. You have to synchronize the model with the external state and then notify the `TreeView`s if the model has changed. The following methods are used to notify the `TreeView`s of model changes by emitting the appropriate signal: `def row_changed(`*`path`*`, `*`iter`*`)`

```
def row_inserted(path, iter)
def row_has_child_toggled(path, iter)
def row_deleted(path)
def rows_reordered(path, iter, new_order)
```

14.11.4 Memory Management

One of the problems with the GenericTreeModel is that TreeIters hold a reference to a Python object returned from your custom tree model. Since the TreeIter may be created and initialized in C code and live on the stack, it's not possible to know when the TreeIter has been destroyed and the Python object reference is no longer being used. Therefore, the Python object referenced in a TreeIter has by default its reference count incremented but it is not decremented when the TreeIter is destroyed. This ensures that the Python object will not be destroyed while being used by a TreeIter and possibly cause a segfault. Unfortunately the extra reference counts lead to the situation that, at best, the Python object will have an excessive reference count and, at worst, it will never be freed even when it is not being used. The latter case leads to memory leaks and the former to reference leaks.

To provide for the situation where the custom TreeModel holds a reference to the Python object until it is no longer available (i.e. the TreeIter is invalid because the model has changed) and there is no need to leak references, the GenericTreeModel has the "leak-references" property. By default "leak-references" is TRUE to indicate that the GenericTreeModel will leak references. If "leak-references" is set to FALSE, the reference count of the Python object will not be incremented when referenced in a TreeIter. This means that your custom TreeModel must keep a reference to all Python objects used in TreeIters until the model is destroyed. Unfortunately, even this cannot protect against buggy code that attempts to use a saved TreeIter on a different GenericTreeModel. To protect against that case your application would have to keep references to all Python objects referenced from a TreeIter for any GenericTreeModel instance. Of course, this ultimately has the same result as leaking references.

In PyGTK 2.4 and above the invalidate_iters() and iter_is_valid() methods are available to help manage the TreeIters and their Python object references:

```
generictreemodel.invalidate_iters()

result = generictreemodel.iter_is_valid(iter)
```

These are particularly useful when the "leak-references" property is set to FALSE. Tree models derived from GenericTreeModel are protected from problems with out of date TreeIters because the iters are automatically checked for validity with the tree model.

If a custom tree model doesn't support persistent iters (i.e. gtk.TREE_MODEL_ITERS_PERSIST is not set in the return from the TreeModel.get_flags() method), it can call the invalidate_iters() method to invalidate all its outstanding TreeIters when it changes the model (e.g. after inserting a new row). The tree model can also dispose of any Python objects, that were referenced by TreeIters, after calling the invalidate_iters() method.

Applications can use the iter_is_valid() method to determine if a TreeIter is still valid for the custom tree model.

14.11.5 Other Interfaces

The ListStore and TreeStore models support the TreeSortable, TreeDragSource and TreeDragDest interfaces in addition to the TreeModel interface. The GenericTreeModel only supports the TreeModel interface. I believe that this is because of the direct reference of the model at the C level by TreeViews and the TreeModelSort and TreeModelFilter models. To create and use TreeIters requires C glue code to interface with the Python custom tree model that has the data. That glue code is provided by the GenericTreeModel and there appears to be no alternative purely Python way of doing it because the TreeViews and the other models call the GtkTreeModel functions in C passing their reference to the custom tree model.

The TreeSortable interface would need C glue code as well to work with the default TreeViewColumn sort mechanism as explained in Section 14.2.9. However a custom model can do its own sorting and an application can manage the use of sort criteria by handling the TreeViewColumn header clicks and calling the custom tree model sort methods. The model completes the update of the TreeViews by emitting the "rows-reordered" signal using the TreeModel's rows_reordered() method. Thus the GenericTreeModel probably doesn't need to implement the TreeSortable interface.

Likewise, the `GenericTreeModel` doesn't have to implement the `TreeDragSource` and `TreeDragDest` interfaces because the custom tree model can implement its own drag and drop interfaces and the application can handle the appropriate `TreeView` signals and call the custom tree model methods as needed.

14.11.6 Applying The GenericTreeModel

I believe that the `GenericTreeModel` should only be used as a last resort. There are powerful mechanisms in the standard group of `TreeView` objects that should be sufficient for most applications. Undoubtedly there are applications which may require the use of the `GenericTreeModel` but you should attempt to first use the following instead:

Cell Data Functions As illustrated in Section 14.4.5, cell data functions can be used to modify and even synthesize the data for a `TreeView` column display. You can effectively create as many display columns with generated data as you wish. This gives you a great deal of control over the presentation of data from an underlying data source.

TreeModelFilter In PyGTK 2.4, the `TreeModelFilter` as described in Section 14.10.2 provides a great degree of control over the display of the columns and rows of a child `TreeModel` including presenting just the child rows of a row. Data columns can be synthesized similar to using Cell Data Functions but here the model appears to be a `TreeModel` with the number and type of columns specified whereas a cell data function leaves the model columns unchanged and just modifies the display in a `TreeView`.

If a `GenericTreeModel` must be used you should be aware that:

- the entire `TreeModel` interface must be created and made to work as documented. There are subtleties that can lead to bugs. By contrast, the standard `TreeModel`s are thoroughly tested.
- managing the references of Python objects used by `TreeIter`s can be difficult especially for long running programs with lots of variety of display.
- an interface has to be developed for adding, deleting and changing the contents of rows. There is some awkwardness with the mapping of `TreeIter`s to the Python objects and model rows in this interface.
- there is significant effort in developing sortable and drag and drop interfaces. The application probably needs to be involved in making these interfaces fully functional.

14.12 The Generic CellRenderer

Chapter 15

New Widgets in PyGTK 2.2

The Clipboard object was added in PyGTK 2.2. The GtkClipboard was available in GTK+ 2.0 but was not wrapped by PyGTK 2.0 because it was not a complete GObject. Some new objects were added to the gtk.gdk module in PyGTK 2.2 but they will not be described in this tutorial. See the PyGTK 2 Reference Manual for more information on the gtk.gdk.Display, gtk.gdk.DisplayManager and gtk.gdk.Screen objects.

15.1 Clipboards

A Clipboard provides a storage area for sharing data between processes or between different widgets in the same process. Each Clipboard is identified by a string name encoded as a gdk.Atom. You can use any name you want to identify a Clipboard and a new one will be created if it doesn't exist. If you want to share a Clipboard with other processes each process will need to know the Clipboard's name.

Clipboards are built on the SelectionData and selection interfaces. The default Clipboard used by the TextView, Label and Entry widgets is "CLIPBOARD". Other common clipboards are "PRIMARY" and "SECONDARY" that correspond to the primary and secondary selections (Win32 ignores these). These can also be specified using the gtk.gdk.Atom objects: gtk.gdk.SELECTION_CLIPBOARD, gtk.gdk.SELECTION_PRIMARY and gtk.gdk.SELECTION_SECONDARY. See the gtk.gdk.Atom reference documentation for more information.

15.1.1 Creating A Clipboard

A Clipboard is created using the constructor:

```
clipboard = gtk.Clipboard(display, selection)
```

where *display* is the gtk.gdk.Display associated with the Clipboard named by *selection*. The following convenience function creates a Clipboard using the default gtk.gdk.Display:

```
clipboard = gtk.clipboard_get(selection)
```

Finally, a Clipboard can also be created using the Widget method:

```
clipboard = widget.get_clipboard(selection)
```

The widget must be realized and be part of a toplevel window hierarchy.

15.1.2 Using Clipboards with Entry, Spinbutton and TextView

Entry, SpinButton and TextView widgets have popup menus that provide the ability to cut and copy the selected text to and paste from the "CLIPBOARD" clipboard. In addition key bindings are set to allow keyboard accelerators to cut, copy and paste. Cut is activated by Control+X; copy, by Control+C; and, paste, by Control+V.

The widgets (Entry and SpinButton) implement the Editable interface that has the following methods to cut, copy and paste to and from the "CLIPBOARD" clipboard:

```
editable.cut_clipboard()
editable.copy_clipboard()
editable.paste_clipboard()
```

A `Label` that is selectable (the "selectable" property is `TRUE`) also supports copying the selected text to the "CLIPBOARD" clipboard using a popup menu or the Control+C keyboard accelerator.

`TextBuffer`s have similar methods though they also allow specifying the clipboard to use:

```
textbuffer.copy_clipboard(clipboard)
```

The selection text will be copied to the `Clipboard` specified by *clipboard*.

```
textbuffer.cut_clipboard(clipboard, default_editable)
```

The selected text will be copied to *clipboard*. If *default_editable* is `TRUE` the selected text will also be deleted from the `TextBuffer`. Otherwise, cut_clipboard() will act like the copy_clipboard() method.

```
textbuffer.paste_clipboard(clipboard, override_location, default_editable)
```

If *default_editable* is `TRUE`, the contents of *clipboard* will be inserted into the `TextBuffer` at the location specified by the `TextIter` *override_location*. If *default_editable* is `FALSE`, paste_clipboard() will not insert the contents of *clipboard*. If *override_location* is `None` the contents of *clipboard* will be inserted at the cursor location.

`TextBuffer`s also have two methods to manage a set of `Clipboard`s that are automatically set with the contents of the current selection:

```
textbuffer.add_selection_clipboard(clipboard)
textbuffer.remove_selection_clipboard(clipboard)
```

When a `TextBuffer` is added to a `TextView` the "PRIMARY" clipboard is automatically added to the selection clipboards. Your application can add other clipboards (for example, the "CLIPBOARD" clipboard).

15.1.3 Setting Data on a Clipboard

You can set the `Clipboard` data programmatically using either of:

```
clipboard.set_with_data(targets, get_func, clear_func, user_data)

clipboard.set_text(text, len=-1)
```

The set_with_data() method indicates which selection data targets are supported and provides functions (*get_func* and *clear_func*) that are called when the data is asked for or the clipboard data is changed. *user_data* is passed to *get_func* or *clear_func* when called. *targets* is a list of 3-tuples containing:

- a string representing a target supported by the clipboard.
- a flags value used for drag and drop - use 0.
- an application assigned integer that is passed as a parameter to a signal handler to help identify the target type.

The signatures of *get_func* and *clear_func* are:

```
def get_func(clipboard, selectiondata, info, data):

def clear_func(clipboard, data):
```

where *clipboard* is the `Clipboard`, *selectiondata* is a `SelectionData` object to set the data in, *info* is the application assigned integer associated with a target, and *data* is *user_data*.

set_text() is a convenience method that uses the set_with_data() method to set text data on a `Clipboard` with the targets: "STRING", "TEXT", "COMPOUND_TEXT", and "UTF8_STRING". It uses internal get and clear functions to manage the data. set_text() is equivalent to the following:

```
def my_set_text(self, text, len=-1):
    targets = [ ("STRING", 0, 0),
                ("TEXT", 0, 1),
                ("COMPOUND_TEXT", 0, 2),
                ("UTF8_STRING", 0, 3) ]
    def text_get_func(clipboard, selectiondata, info, data):
        selectiondata.set_text(data)
        return
    def text_clear_func(clipboard, data):
        del data
        return
    self.set_with_data(targets, text_get_func, text_clear_func, text)
    return
```

Once data is set on a clipboard, it will be available until the application is finished or the clipboard data is changed.

To provide the behavior typical of cut to a clipboard, your application will have to delete the selected text or object after copying it to the clipboard.

15.1.4 Retrieving the Clipboard Contents

The contents of a `Clipboard` can be retrieved using the following method:

```
clipboard.request_contents(target, callback, user_data=None)
```

The contents specified by *target* are retrieved asynchronously in the function specified by *callback* which is called with *user_data*. The signature of *callback* is:

```
def callback(clipboard, selectiondata, data):
```

where *selectiondata* is a `SelectionData` object containing the contents of *clipboard*. *data* is *user_data*. The request_contents() method is the most general way of retrieving the contents of a `Clipboard`. The following convenience method retrieves the text contents of a `Clipboard`:

```
clipboard.request_text(callback, user_data=None)
```

The text string is returned to the callback function instead of a `Selectiondata` object. You can check which targets are available on the `Clipboard` by using the method:

```
clipboard.request_targets(callback, user_data=None)
```

The targets are returned as a tuple of `gtk.gdk.Atom` objects to the callback function.

Two convenience methods are provided to return the `Clipboard` contents synchronously:

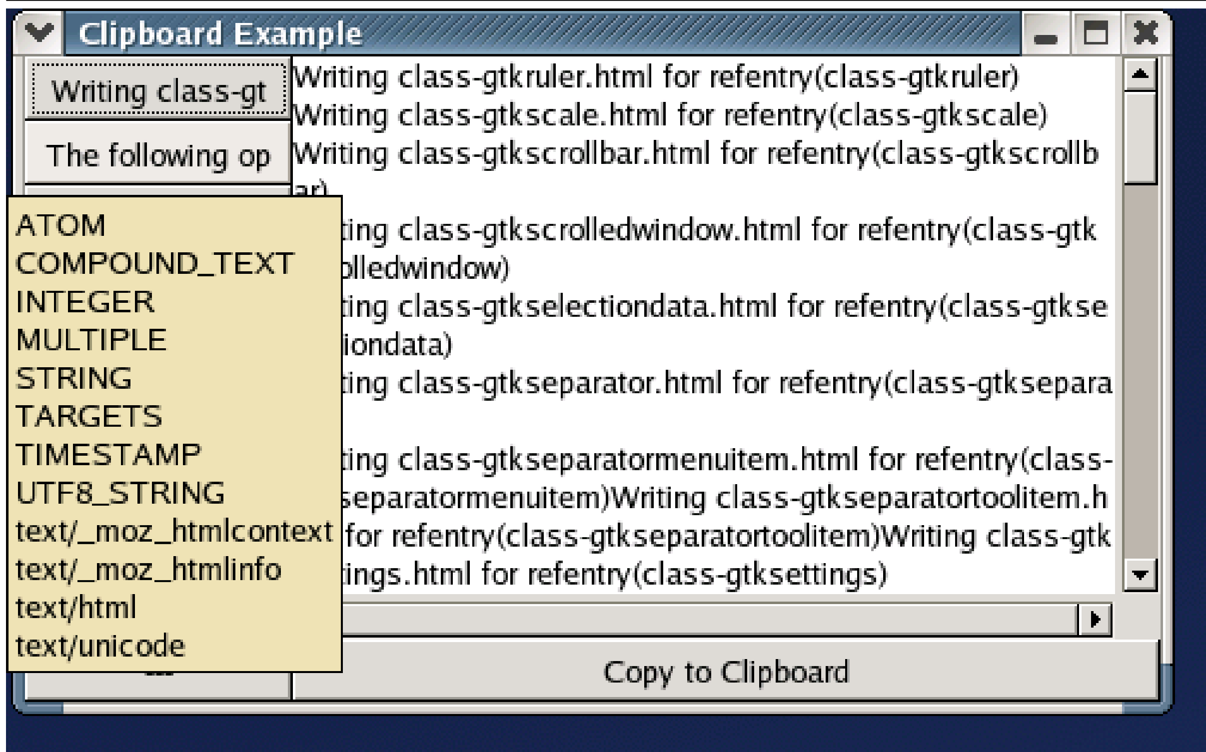
```
selectiondata = clipboard.wait_for_contents(target)

text = clipboard.wait_for_text()
```

15.1.5 A Clipboard Example

To illustrate the use of a `Clipboard` the clipboard.py example program tracks the text items that are cut or copied to the "CLIPBOARD" clipboard and saves the last ten clipboard entries. There are ten buttons that provide access to the text of the saved entries. The button label display the first sixteen characters of the saved text and the tooltips display the targets that the entry originally had. When an entry button is clicked the text window is loaded with the associated saved text which is editable. The button below the text window saves the current text window contents to the clipboard.

Figure 15.1 illustrates the clipboard.py example program in operation:

Figure 15.1 Clipboard Example Program



The example program polls the clipboard every 1.5 seconds to see if the contents have changed. The program could be changed to duplicate the complete set of target contents and then take ownership of the clipboard using the set_with_data() method. Later, when another program sets the contents of the clipboard, the `clear_func` will be called and it can be used to reload the clipboard contents and retake the clipboard ownership.

Chapter 16

New Widgets in PyGTK 2.4

Quite a few new widgets and support objects were added in PyGTK 2.4 including:

- `Action`, `RadioAction`, `ToggleAction` - objects that represent actions that a user can take. Actions contain information to be used to create proxy widgets (for example, icons, menu items and toolbar items).
- `ActionGroup` - an object containing Actions that have some relationship, for example, actions to open, close and print a document.
- `Border` - an object containing the values for a border.
- `ColorButton` - a button used to launch a ColorSelectionDialog.
- `ComboBox` - a widget providing a list of items to choose from. It replaces the `OptionMenu`.
- `ComboBoxEntry` - a widget providing a text entry field with a dropdown list of items to choose from. It replaces the `Combo`.
- `EntryCompletion` - an object providing completion for an `Entry` widget.
- `Expander` - a container that can show and hide its child in response to its button click.
- `FileChooser` - an interface for choosing files.
- `FileChooserWidget` - a widget implementing the `FileChooser` interface. It replaces the `FileSelection` widget.
- `FileChooserDialog` - a dialog used for "File/Open" and "File/Save" actions. It replaces the `FileSelectionDialog`.
- `FileFilter` - an object used to filter files based on an internal set of rules.
- `FontButton` - a button that launches the `FontSelectionDialog`.
- `IconInfo` - an object containing information about an icon in an `IconTheme`.
- `IconTheme` - an object providing lookup of icons by name and size.
- `ToolItem`, `ToolButton`, `RadioToolButton`, `SeparatorToolItem`, `ToggleToolButton` - widgets that can be added to a `Toolbar`. These replace the previous `Toolbar` items.
- `TreeModelFilter` - an object providing a powerful mechanism for revising the representation of an underlying `TreeModel`. This is described in Section 14.10.2.
- `UIManager` - an object providing a way to construct menus and toolbars from an XML UI description. It also has methods to manage the merging and separation of multiple UI descriptions.

16.1 The Action and ActionGroup Objects

The `Action` and `ActionGroup` objects work together to provide the images, text, callbacks and accelerators for your application menus and toolbars. The `UIManager` uses `Action`s and `ActionGroup`s to build the menubars and toolbars automatically based on a XML specification. It's much easier to create and populate menus and toolbars using the `UIManager` described in a later section. The following sections on the `Action` and `ActionGroup` objects describe how to directly apply these objects but I recommend using the `UIManager` whenever possible.

16.1.1 Actions

An `Action` object represents an action that the user can take using an application user interface. It contains information used by proxy UI elements (for example, `MenuItem`s or `Toolbar` items) to present the action to the user. There are two subclasses of `Action`:

ToggleAction An `Action` that can be toggled between two states.

RadioAction An `Action` that can be grouped so that only one can be active.

For example, the standard File → Quit menu item can be represented with an icon, mnemonic text and accelerator. When activated, the menu item triggers a callback that could exit the application. Likewise a `Toolbar` Quit button could share the icon, mnemonic text and callback. Both of these UI elements could be proxies of the same `Action`.

Ordinary `Button`, `ToggleButton` and `RadioButton` widgets can also act as proxies for an `Action` though there is no support for these in the `UIManager`.

16.1.1.1 Creating Actions

An `Action` can be created using the constructor:

```
action = gtk.Action(name, label, tooltip, stock_id)
```

name is a string used to identify the `Action` in an `ActionGroup` or in a `UIManager` specification. *label* and *tooltip* are strings used as the label and tooltip in proxy widgets. If *label* is `None` then the *stock_id* must be a string specifying a Stock Item to get the label from. If *tooltip* is `None` the `Action` will not have a tooltip.

As we'll see in Section 16.1.2 it's much easier to create Action objects using the `ActionGroup` convenience methods:

```
actiongroup.add_actions(entries, user_data=None)
actiongroup.add_toggle_actions(entries, user_data=None)
actiongroup.add_radio_actions(entries, value=0, on_change=None, user_data=None)
```

More about these later but first I'll describe how to use an `Action` with a `Button` to illustrate the basic operations of connecting an `Action` to a proxy widget.

16.1.1.2 Using Actions

The basic procedure for using an `Action` with a `Button` proxy is illustrated by the simpleaction.py example program. The `Button` is connected to the `Action` using the method:

```
action.connect_proxy(proxy)
```

where proxy is a `MenuItem`, `ToolItem` or `Button` widget.

An `Action` has one signal the "activate" signal that is triggered when the `Action` is activated usually as the result of a proxy widget being activated (for example a `ToolButton` is clicked). You just have connect a callback to this signal to handle the activation of any of the proxy widgets.

The source code for the simpleaction.py example program is:

```
1    #!/usr/bin/env python
2
3    import pygtk
4    pygtk.require('2.0')
5    import gtk
```

```
 6
 7  class SimpleAction:
 8      def __init__(self):
 9          # Create the toplevel window
10          window = gtk.Window()
11          window.set_size_request(70, 30)
12          window.connect('destroy', lambda w: gtk.main_quit())
13
14          # Create an accelerator group
15          accelgroup = gtk.AccelGroup()
16          # Add the accelerator group to the toplevel window
17          window.add_accel_group(accelgroup)
18
19          # Create an action for quitting the program using a stock item
20          action = gtk.Action('Quit', None, None, gtk.STOCK_QUIT)
21          # Connect a callback to the action
22          action.connect('activate', self.quit_cb)
23
24          # Create an ActionGroup named SimpleAction
25          actiongroup = gtk.ActionGroup('SimpleAction')
26          # Add the action to the actiongroup with an accelerator
27          # None means use the stock item accelerator
28          actiongroup.add_action_with_accel(action, None)
29
30          # Have the action use accelgroup
31          action.set_accel_group(accelgroup)
32
33          # Connect the accelerator to the action
34          action.connect_accelerator()
35
36          # Create the button to use as the action proxy widget
37          quitbutton = gtk.Button()
38          # add it to the window
39          window.add(quitbutton)
40
41          # Connect the action to its proxy widget
42          action.connect_proxy(quitbutton)
43
44          window.show_all()
45          return
46
47      def quit_cb(self, b):
48          print 'Quitting program'
49          gtk.main_quit()
50
51  if __name__ == '__main__':
52      sa = SimpleAction()
53      gtk.main()
```

The example creates an `Action` (line 20) that uses a Stock Item to provide the label text with mnemonic, icon, accelerator and translation domain. If a Stock Item is not used you'll need to specify a label instead. Line 22 connects the "activate" signal of *action* to the self.quit_cb() method so that it is invoked when the `Action` is activated by *quitbutton*. Line 42 connects *quitbutton* to *action* as a proxy widget. When *quitbutton* is clicked it will activate *action* and thereby invoke the self.quit_cb() method. The simpleaction.py example uses quite a bit of code (lines 15, 17, 31 and 34 to setup the accelerator for the `Button`. The procedure is similar for `MenuItem`s and `Toolbar` `ToolItem`s.

Figure 16.1 shows the simpleaction.py example in operation.

Figure 16.1 Simple Action Example

16.1.1.3 Creating Proxy Widgets

In the previous section we saw that an existing widget could be connected to an `Action` as a proxy. In this section we'll see how a proxy widget can be created using the `Action` methods:

```
menuitem = action.create_menu_item()

toolitem = action.create_tool_item()
```

The basicaction.py example illustrates a `MenuItem`, `ToolButton` and a `Button` sharing an `Action`. The `MenuItem` and the `ToolButton` are created using the above methods. The basicaction.py example program source code is:

```
 1   #!/usr/bin/env python
 2
 3   import pygtk
 4   pygtk.require('2.0')
 5   import gtk
 6
 7   class BasicAction:
 8       def __init__(self):
 9           # Create the toplevel window
10           window = gtk.Window()
11           window.connect('destroy', lambda w: gtk.main_quit())
12           vbox = gtk.VBox()
13           vbox.show()
14           window.add(vbox)
15
16           # Create an accelerator group
17           accelgroup = gtk.AccelGroup()
18           # Add the accelerator group to the toplevel window
19           window.add_accel_group(accelgroup)
20
21           # Create an action for quitting the program using a stock item
22           action = gtk.Action('Quit', '_Quit me!', 'Quit the Program',
23                               gtk.STOCK_QUIT)
24           action.set_property('short-label', '_Quit')
25           # Connect a callback to the action
26           action.connect('activate', self.quit_cb)
27
28           # Create an ActionGroup named BasicAction
29           actiongroup = gtk.ActionGroup('BasicAction')
30           # Add the action to the actiongroup with an accelerator
31           # None means use the stock item accelerator
32           actiongroup.add_action_with_accel(action, None)
33
34           # Have the action use accelgroup
35           action.set_accel_group(accelgroup)
36
37           # Create a MenuBar
38           menubar = gtk.MenuBar()
39           menubar.show()
40           vbox.pack_start(menubar, False)
41
42           # Create the File Action and MenuItem
43           file_action = gtk.Action('File', '_File', None, None)
```

```
44          actiongroup.add_action(file_action)
45          file_menuitem = file_action.create_menu_item()
46          menubar.append(file_menuitem)
47
48          # Create the File Menu
49          file_menu = gtk.Menu()
50          file_menuitem.set_submenu(file_menu)
51
52          # Create a proxy MenuItem
53          menuitem = action.create_menu_item()
54          file_menu.append(menuitem)
55
56          # Create a Toolbar
57          toolbar = gtk.Toolbar()
58          toolbar.show()
59          vbox.pack_start(toolbar, False)
60
61          # Create a proxy ToolItem
62          toolitem = action.create_tool_item()
63          toolbar.insert(toolitem, 0)
64
65          # Create and pack a Label
66          label = gtk.Label('''
67  Select File->Quit me! or
68  click the toolbar Quit button or
69  click the Quit button below or
70  press Control+q
71  to quit.
72  ''')
73          label.show()
74          vbox.pack_start(label)
75
76          # Create a button to use as another proxy widget
77          quitbutton = gtk.Button()
78          # add it to the window
79          vbox.pack_start(quitbutton, False)
80
81          # Connect the action to its proxy widget
82          action.connect_proxy(quitbutton)
83          # Have to set tooltip after toolitem is added to toolbar
84          action.set_property('tooltip', action.get_property('tooltip'))
85          tooltips = gtk.Tooltips()
86          tooltips.set_tip(quitbutton, action.get_property('tooltip'))
87
88          window.show()
89          return
90
91      def quit_cb(self, b):
92          print 'Quitting program'
93          gtk.main_quit()
94
95  if __name__ == '__main__':
96      ba = BasicAction()
97      gtk.main()
```

This example introduces an `ActionGroup` to hold the `Action`s used in the program. Section 16.1.2 will go into more detail on the use of `ActionGroup`s.

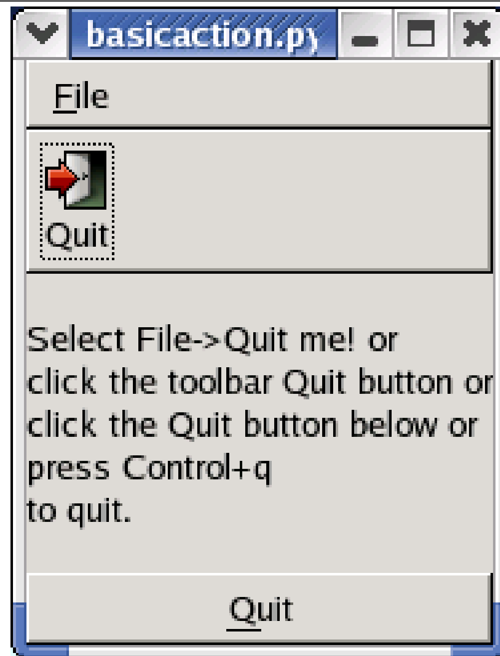
The code in lines 9-14 sets up a top level window containing a `VBox`. Lines 16-35 set up the "Quit" `Action` similar to that in the simpleaction.py example program and add it with the `gtk.STOCK_QUIT` Stock Item accelerator (line 32) to the "BasicAction" `ActionGroup` (created in line 29). Note that, unlike the simpleaction.py example program, you don't have to call the connect_accelerator() method for the action since it is called automatically when the create_menu_item() method is called in line 53.

Lines 38-40 create a `MenuBar` and pack it into the `VBox`. Lines 43-44 create an `Action` (*file_action*) for the File menu and add it to *actiongroup*. The File and Quit menu items are created in lines 45 and 53 and added to *menubar* and *file_menu* respectively in lines 46 and 54.

Likewise a `Toolbar` is created and added to the `VBox` in lines 57-59. The proxy `ToolItem` is created and added to *toolbar* in lines 62-63. Note the `Action` tooltip must be set (line 84) after the `ToolItem` is added to the `Toolbar` for it to be used. Also the `Button` tooltip must be added manually (lines 84-86).

Figure 16.2 displays the basicaction.py example program in operation:

Figure 16.2 Basic Action Example



A proxy widget can be disconnected from an `Action` by using the method:

```
action.disconnect_proxy(proxy)
```

16.1.1.4 Action Properties

An `Action` has a number of properties that control the display and function of its proxy widgets. The most important of these are the "sensitive" and "visible" properties. The "sensitive" property determines the sensitivity of the proxy widgets. If "sensitive" is `FALSE` the proxy widgets are not activatable and will usually be displayed "grayed out". Likewise, the "visible" property determines whether the proxy widgets will be visible. If an `Action`'s "visible" property is `FALSE` its proxy widgets will be hidden.

As we'll see in the next section, an `Action`'s sensitivity or visibility is also controlled by the sensitivity or visibility of the `ActionGroup` it belongs to. Therefore, for an `Action` to be sensitive (or visible) both it and its `ActionGroup` must be sensitive (or visible). To determine the effective sensitivity or visibility of an `Action` you should use the following methods:

```
result = action.is_sensitive()

result = action.is_visible()
```

The name assigned to an `Action` is contained in its "name" property which is set when the `Action` is created. You can retrieve that name using the method:

```
name = action.get_name()
```

Other properties that control the display of the proxy widgets of an `Action` include:

"hide-if-empty" If `TRUE`, empty menu proxies for this action are hidden.

"is-important" If `TRUE`, `ToolItem` proxies for this action show text in `gtk.TOOLBAR_BOTH_HORIZ` mode.

"visible-horizontal" If `TRUE`, the `ToolItem` is visible when the toolbar is in a horizontal orientation.

"visible-vertical" If `TRUE`, the `ToolItem` is visible when the toolbar is in a vertical orientation.

Other properties of interest include:

"label" The label used for menu items and buttons that activate this action.

"short-label" A shorter label that may be used on toolbar buttons and buttons.

"stock-id" The Stock Item to be used to retrieve the icon, label and accelerator to be used in widgets representing this action.

"tooltip" A tooltip for this action.

Note that the basicaction.py example program overrides the `gtk.STOCK_QUIT` label with "_Quit me!" and sets the "short-label" property to "_Quit". The short label is used for the `ToolButton` and the `Button` labels but the full label is used for the `MenuItem` label. Also note that the tooltip cannot be set on a `ToolItem` until it is added to a `Toolbar`.

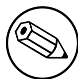
16.1.1.5 Actions and Accelerators

An `Action` has three methods that are used to set up an accelerator:

```
action.set_accel_group(accel_group)

action.set_accel_path(accel_path)

action.connect_accelerator()
```

These, in conjunction with the gtk.ActionGroup.add_action_with_accel() method, should cover most cases of accelerator set up.

An `AccelGroup` must always be set for an `Action`. The set_accel_path() method is called by the gtk.ActionGroup.add_action_with_accel() method. If set_accel_path() is used the accelerator path should match the default format: "<Actions>/actiongroup_name/action_name". Finally, the connect_accelerator() method is called to complete the accelerator set up.

> **NOTE**
>
> 
>
> An `Action` must have an `AccelGroup` and an accelerator path associated with it before connect_accelerator() is called.

Since the connect_accelerator() method can be called several times (i.e. once for each proxy widget), the number of calls is counted so that an equal number of disconnect_accelerator() calls must be made before removing the accelerator.

As illustrated in the previous example programs, an `Action` accelerator can be used by all the proxy widgets. An `Action` should be part of an `ActionGroup` in order to use the default accelerator path that has the format: "<Actions>/actiongroup_name/action_name". The easiest way to add an accelerator is to use the gtk.ActionGroup.add_action_with_accel() method and the following general procedure:

- Create an `AccelGroup` and add it to the top level window.
- Create a new `ActionGroup`
- Create an `Action` specifying a Stock Item with an accelerator.
- Add the `Action` to the `ActionGroup` using the gtk.ActionGroup.add_action_with_accel() method specifying `None` to use the Stock Item accelerator or an accelerator string acceptable to `gtk.accelerator_parse()`.

- Set the AccelGroup for the Action using the gtk.Action.set_accel_group() method.
- Complete the accelerator set up using the gtk.Action.connect_accelerator() method.

Any proxy widgets created by or connected to the Action will use the accelerator.

16.1.1.6 Toggle Actions

As mentioned previously a ToggleAction is a subclass of Action that can be toggled between two states. The constructor for a ToggleAction takes the same parameters as an Action:

```
toggleaction = gtk.ToggleAction(name, label, tooltip, stock_id)
```

In addition to the Action methods the following ToggleAction methods:

```
toggleaction.set_active(is_active)
is_active = toggleaction.get_active()
```

set and get the current state of *toggleaction*. *is_active* is a boolean value.
You can connect to the "toggled" signal specifying a callback with the signature:

```
def toggled_cb(toggleaction, user_data)
```

The "toggled" signal is emitted when the ToggleAction changes state.
A MenuItem proxy widget of a ToggleAction will be displayed like a CheckMenuItem by default. To have the proxy MenuItem displayed like a RadioMenuItem set the "draw-as-radio" property to TRUE using the method:

```
toggleaction.set_draw_as_radio(draw_as_radio)
```

You can use the following method to determine whether the ToggleAction MenuItems will be displayed like RadioMenuItems:

```
draw_as_radio = toggleaction.get_draw_as_radio()
```

16.1.1.7 Radio Actions

A RadioAction is a subclass of ToggleAction that can be grouped so that only one RadioAction is active at a time. The corresponding proxy widgets are the RadioMenuItem and RadioToolButton.
The constructor for a RadioAction takes the same arguments as an Action with the addition of a unique integer value that is used to identify the active RadioAction in a group:

```
radioaction = gtk.RadioAction(name, label, tooltip, stock_id, value)
```

The group for a RadioAction can be set using the method:

```
radioaction.set_group(group)
```

where *group* is another RadioAction that *radioaction* should be grouped with. The group containing a RadioAction can be retrieved using the method:

```
group = radioaction.get_group()
```

that returns a list of the group of RadioAction objects that includes *radioaction*.
The value of the currently active group member can retrieved using the method:

```
active_value = radioaction.get_current_value()
```

You can connect a callback to the "changed" signal to be notified when the active member of the RadioAction group has been changed. Note that you only have to connect to one of the RadioAction objects to track changes. The callback signature is:

```
def changed_cb(radioaction, current, user_data)
```

where *current* is the currently active RadioAction in the group.

16.1.1.8 An Actions Example

The actions.py example program illustrates the use of the `Action`, `ToggleAction` and `RadioAction` objects. Figure 16.3 displays the example program in operation:

Figure 16.3 Actions Example

This example is similar enough to the basicaction.py example program that a detailed description is not necessary.

16.1.2 ActionGroups

As mentioned in the previous section, related `Action` objects should be added to an `ActionGroup` to provide common control over their visibility and sensitivity. For example, in a text processing application the menu items and toolbar buttons for specifying the text justification could be contained in an `ActionGroup`. A user interface is expected to have multiple `ActionGroup` objects that cover various aspects of the application. For example, global actions like creating new documents, opening and saving a document and quitting the application likely form one `ActionGroup` while actions such as modifying the view of the document would form another.

16.1.2.1 Creating ActionGroups

An `ActionGroup` is created using the constructor:

```
actiongroup = gtk.ActionGroup(name)
```

where *name* is a unique name for the `ActionGroup`. The name should be unique because it is used to form the default accelerator path for its `Action` objects.

The *ActionGroup* name can be retrieved using the method:

```
name = actiongroup.get_name()
```

or by retrieving the contents of the "name" property.

16.1.2.2 Adding Actions

As illustrated in Section 16.1.1 an existing `Action` can be added to an `ActionGroup` using one of the methods:

```
actiongroup.add_action(action)

actiongroup.add_action_with_accel(action, accelerator)
```

where *action* is the `Action` to be added and *accelerator* is a string accelerator specification acceptable to `gtk.accelerator_parse`(). If *accelerator* is `None` the accelerator (if any) associated with the "stock-id" property of *action* will be used. As previously noted the add_action_wih_accel() method is preferred if you want to use accelerators.

The `ActionGroup` offers three convenience methods that make the job of creating and adding `Action` objects to an `ActionGroup` much easier:

```
actiongroup.add_actions(entries, user_data=None)

actiongroup.add_toggle_actions(entries, user_data=None)

actiongroup.add_radio_actions(entries, value=0, on_change=None, user_data=None)
```

The *entries* parameter is a sequence of action entry tuples that provide the information used to create the actions that are added to the `ActionGroup`. The `RadioAction` with the value of *value* is initially set active. *on_change* is a callback that is connected to the "changed" signal of the first `RadioAction` in the group. The signature of *on_changed* is:

```
def on_changed_cb(radioaction, current, user_data)
```

The entry tuples for `Action` objects contain:

- The name of the action. Must be specified.
- The stock id for the action. Optional with a default value of `None` if a label is specified.
- The label for the action. Optional with a default value of `None` if a stock id is specified.
- The accelerator for the action, in the format understood by the `gtk.accelerator_parse`() function. Optional with a default value of `None`.
- The tooltip for the action. Optional with a default value of `None`.
- The callback function invoked when the action is activated. Optional with a default value of `None`.

You must minimally specify a value for the *name* field and a value in either the *stock id* field or the *label* field. If you specify a label then you can specify `None` for the stock id if you aren't using one. For example the following method call:

```
actiongroup.add_actions([('quit', gtk.STOCK_QUIT, '_Quit me!', None,
                          'Quit the Program', quit_cb)])
```

adds an action to *actiongroup* for exiting a program.

The entry tuples for the `ToggleAction` objects are similar to the `Action` entry tuples except there is an additional optional *flag* field containing a boolean value indicating whether the action is active. The default value for the *flag* field is `FALSE`. For example the following method call:

```
actiongroup.add_toggle_actions([('mute, None, '_Mute', '<control>m',
                                 'Mute the volume', mute_cb, True)])
```

adds a `ToggleAction` to *actiongroup* and sets it to be initially active.

The entry tuples for the `RadioAction` objects are similar to the `Action` entry tuples but specify a *value* field instead of a *callback* field:

- The name of the action. Must be specified.
- The stock id for the action. Optional with a default value of `None` if a label is specified.
- The label for the action.Optional with a default value of `None` if a stock id is specified.
- The accelerator for the action, in the format understood by the `gtk.accelerator_parse`() function. Optional with a default value of `None`.
- The tooltip for the action. Optional with a default value of `None`.
- The value to set on the radio action. Optional with a default value of `0`. Should always be specified in applications.

For example the following code fragment:

```
radioactionlist = [('am', None, '_AM', '<control>a', 'AM Radio', 0)
                   ('fm', None, '_FM', '<control>f', 'FM Radio', 1)
                   ('ssb', None, '_SSB', '<control>s', 'SSB Radio', 2)]
actiongroup.add_radio_actions(radioactionlist, 0, changed_cb)
```

creates three `RadioAction` objects and sets the initial active action to 'am' and the callback that is invoked when any of the actions is activated to `changed_cb`.

16.1.2.3 Retrieving Actions

An `Action` can be retrieved by name from an `ActionGroup` by using the method:

```
action = actiongroup.get_action(action_name)
```

A list of all the `Action` objects contained in an `ActionGroup` can be retrieved using the method:

```
actionlist = actiongroup.list_actions()
```

16.1.2.4 Controlling Actions

The sensitivity and visibility of all `Action` objects in an `ActionGroup` can be controlled by setting the associated property values. The following convenience methods get and set the properties:

```
is_sensitive = actiongroup.get_sensitive()
actiongroup.set_sensitive(sensitive)

is_visible = actiongroup.get_visible()
actiongroup.set_visible(visible)
```

Finally you can remove an `Action` from an `ActionGroup` using the method:

```
actiongroup.remove_action(action)
```

16.1.2.5 An ActionGroup Example

The actiongroup.py example program duplicates the menubar and toolbar of the actions.py example program using the `ActionGroup` methods. In addition the program provides buttons to control the sensitivity and visibility of the menu items and toolbar items. Figure 16.4 illustrates the program in operation:

Figure 16.4 ActionGroup Example

16.1.2.6 ActionGroup Signals

Your application can track the connection and removal of proxy widgets to the Action objects in an ActionGroup using the "connect-proxy" and disconnect-proxy" signals. The signatures of your signal handler callbacks should be:

```
def connect_proxy_cb(actiongroup, action, proxy, user_params)

def disconnect_proxy_cb(actiongroup, action, proxy, user_params)
```

For example, you might want to track these changes to make some additional changes to the properties of the new proxy widget when it is connected or to update some other part of the user interface when a proxy widget is disconnected.

The "pre-activate" and "post-activate" signals allow your application to do some additional processing immediately before or after an action is activated. The signatures of the signal handler callbacks should be:

```
def pre_activate_cb(actiongroup, action, user_params)

def post_activate_cb(actiongroup, action, user_params)
```

These signals are mostly used by the UIManager to provide global notification for all Action objects in ActionGroup objects used by it.

16.2 ComboBox and ComboBoxEntry Widgets

16.2.1 ComboBox Widgets

The ComboBox replaces the OptionMenu with a powerful widget that uses a TreeModel (usually a ListStore) to provide the list items to display. The ComboBox implements the CellLayout interface that provides a number of methods for managing the display of the list items. One or more CellRenderers can be packed into a ComboBox to customize the list item display.

16.2.1.1 Basic ComboBox Use

The easy way to create and populate a ComboBox is to use the convenience function:

```
combobox = gtk.combo_box_new_text()
```

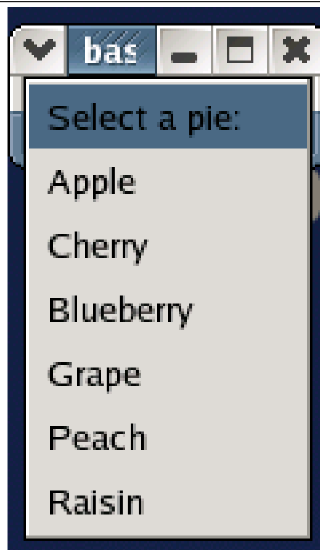
This function creates a ComboBox and its associated ListStore and packs it with a CellRendererText. The following convenience methods are used to populate or remove the contents of the ComboBox and its ListStore:

```
combobox.append_text(text)
combobox.prepend_text(text)
combobox.insert_text(position, text)
combobox.remove_text(position)
```

where *text* is the string to be added to the ComboBox and *position* is the index where *text* is to be inserted or removed. In most cases the convenience function and methods are all you need.

The example program comboboxbasic.py demonstrates the use of the above function and methods. Figure 16.5 illustrates the program in operation:

Figure 16.5 Basic ComboBox



The active text can be retrieved using the method:

```
text = combobox.get_active_text()
```

Prior to version 2.6, the GTK+ developers did not provide such a convenience method to retrieve the active text, so you'd have to create your own implementation, similar to:

```
def get_active_text(combobox):
    model = combobox.get_model()
    active = combobox.get_active()
    if active < 0:
        return None
    return model[active][0]
```

The index of the active item is retrieved using the method:

```
active = combobox.get_active()
```

The active item can be set using the method:

```
combobox.set_active(index)
```

where *index* is an integer larger than -2. If *index* is -1 there is no active item and the ComboBox display will be blank. If *index* is less than -1, the call will be ignored. If *index* is greater than -1 the list item with that index value will be displayed.

You can connect to the "changed" signal of a ComboBox to be notified when the active item has been changed. The signature of the "changed" handler is:

```
def changed_cb(combobox, ...):
```

where ... represents the zero or more arguments passed to the GObject.connect() method.

16.2.1.2 Advanced ComboBox Use

Creating a ComboBox using the gtk.combo_box_new_text() function is roughly equivalent to the following code:

```
liststore = gtk.ListStore(str)
combobox = gtk.ComboBox(liststore)
cell = gtk.CellRendererText()
combobox.pack_start(cell, True)
combobox.add_attribute(cell, 'text', 0)
```

To make use of the power of the various TreeModel and CellRenderer objects you need to construct a ComboBox using the constructor:

```
combobox = gtk.ComboBox(model=None)
```

where *model* is a TreeModel. If you create a ComboBox without associating a TreeModel, you can add one later using the method:

```
combobox.set_model(model)
```

The associated TreeModel can be retrieved using the method:

```
model = combobox.get_model()
```

Some of the things you can do with a ComboBox are:

- Share the same TreeModel with other ComboBoxes and TreeViews.
- Display images and text in the ComboBox list items.
- Use an existing TreeStore or ListStore as the model for the ComboBox list items.
- Use a TreeModelSort to provide a sorted ComboBox list.
- Use a TreeModelFilter to use a subtree of a TreeStore as the source for a ComboBox list items.
- Use a TreeModelFilter to use a subset of the rows in a TreeStore or ListStore as the ComboBox list items.
- Use a cell data function to modify or synthesize the display for list items.

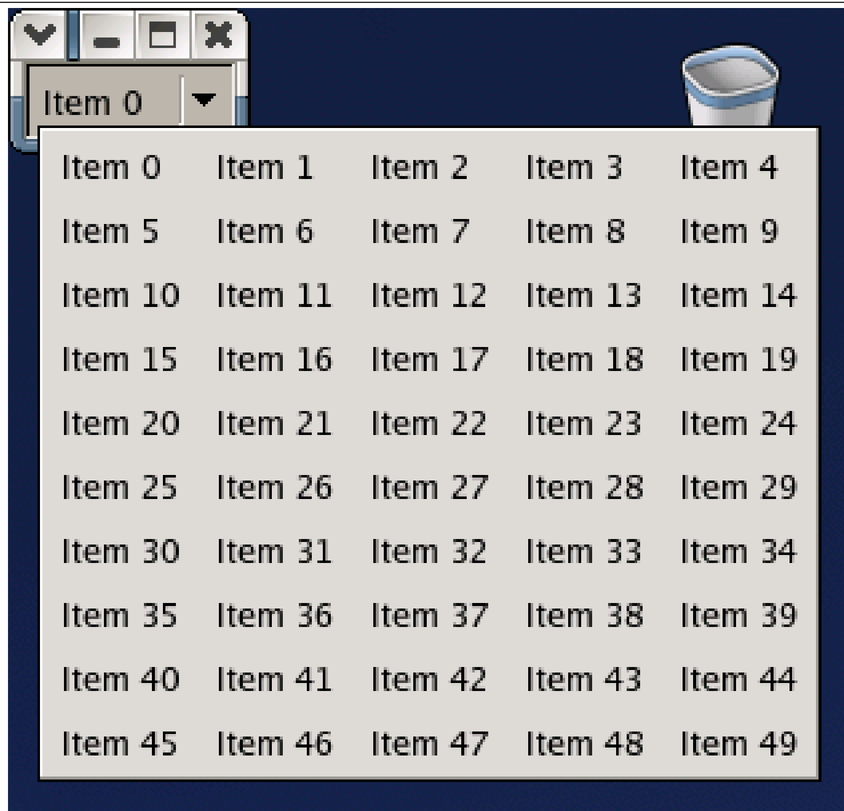
The use of the TreeModel and CellRenderer objects is detailed in Chapter 14.

The ComboBox list items can be displayed in a grid if you have a large number of items to display. Otherwise the list will have scroll arrows if the entire list cannot be displayed. The following method is used to set the number of columns to display:

```
combobox.set_wrap_width(width)
```

where *width* is the number of columns of the grid displaying the list items. For example, the comboboxwrap.py program displays a list of 50 items in 5 columns. Figure 16.6 illustrates the program in operation:

Figure 16.6 ComboBox with Wrapped Layout



With a large number of items, say more than 50, the use of the set_wrap_width() method will have poor performance because of the computation for the grid layout. To get a feel for the affect modify the comboboxwrap.py program line 18 to display 150 items.

```
        for n in range(150):
```

Run the program and get a time estimate for startup. Then modify it by commenting out line 17:

```
        #combobox.set_wrap_width(5)
```

Run and time it again. It should start up significantly faster. My experience is about 20 times faster.

In addition to the get_active() method described above, you can retrieve a `TreeIter` pointing at the active row by using the method:

```
  iter = combobox.get_active_iter()
```

You can also set the active list item using a `TreeIter` with the method:

```
  combobox.set_active_iter(iter)
```

The set_row_span_column() and set_column_span_column() methods are supposed to allow the specification of a `TreeModel` column number that contains the number of rows or columns that the list item is supposed to span in a grid layout. Unfortunately, in GTK+ 2.4 these methods are broken.

Since the `ComboBox` implements the `CellLayout` interface which has similar capabilities as the `TreeViewColumn` (see Section 14.5 for more information). Briefly, the interface provides:

```
  combobox.pack_start(cell, expand=True)
  combobox.pack_end(cell, expand=True)
  combobox.clear()
```

The first two methods pack a `CellRenderer` into the `ComboBox` and the clear() method clears all attributes from all `CellRenderer`s.

The following methods:

```
comboboxentry.add_attribute(cell, attribute, column)

comboboxentry.set_attributes(cell, ...)
```

set attributes for the `CellRenderer` specified by *`cell`*. The add_attribute() method takes a string *`attribute`* name (e.g. 'text') and an integer *`column`* number of the column in the `TreeModel` to use to set *`attribute`*. The additional arguments to the set_attributes() method are `attribute=column` pairs (e.g text=1).

16.2.2 ComboBoxEntry Widgets

The `ComboBoxEntry` widget replaces the `Combo` widget. It is subclassed from the `ComboBox` widget and contains a child `Entry` widget that has its contents set by selecting an item in the dropdown list or by direct text entry either from the keyboard or by pasting from a `Clipboard` or a selection.

16.2.2.1 Basic ComboBoxEntry Use

Like the `ComboBox`, the `ComboBoxEntry` can be created using the convenience function:

```
comboboxentry = gtk.combo_box_entry_new_text()
```

The `ComboBoxEntry` should be populated using the `ComboBox` convenience methods described in Section 16.2.1.1.

Since a `ComboBoxEntry` widget is a `Bin` widget its child `Entry` widget is available using the "child" attribute or the get_child() method:

```
entry = comboboxentry.child
entry = comboboxentry.get_child()
```

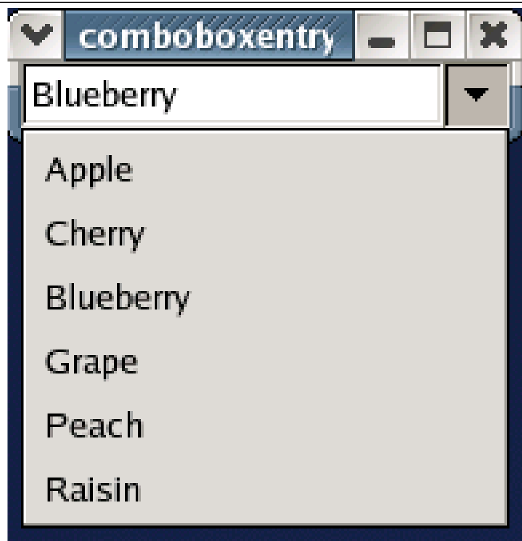
You can retrieve the `Entry` text using its get_text() method.

Like the `ComboBox`, you can track changes in the active list item by connecting to the "changed" signal. Unfortunately, this doesn't help track changes to the text in the `Entry` child that are direct entry. When a direct entry is made to the child `Entry` widget the "changed" signal will be emitted but the index returned by the get_active() method will be -1. To track all changes to the `Entry` text, you'll have to use the `Entry` "changed" signal. For example:

```
def changed_cb(entry):
    print entry.get_text()

comboboxentry.child.connect('changed', changed_cb)
```

will print out the text after every change in the child `Entry` widget. For example, the comboboxentrybasic.py program demonstrates the use of the convenience API. Figure 16.7 illustrates the program in operation:

Figure 16.7 Basic ComboBoxEntry



Note that when the `Entry` text is changed due to the selection of a dropdown list item the "changed" handler is called twice: once when the text is cleared; and, once when the text is set with the selected list item text.

16.2.2.2 Advanced ComboBoxEntry Use

The constructor for a ComboBoxEntry is:

```
comboboxentry = gtk.ComboBoxEntry(model=None, column=-1)
```

where *`model`* is a `TreeModel` and *`column`* is the number of the column in *`model`* to use for setting the list items. If column is not specified the default value is -1 which means the text column is unset.

Creating a `ComboBoxEntry` using the convenience function `gtk.combo_box_entry_new_text()` is equivalent to the following:

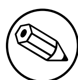
```
liststore = gtk.ListStore(str)
comboboxentry = gtk.ComboBoxEntry(liststore, 0)
```

The `ComboBoxEntry` adds a couple of methods that are used to set and retrieve the `TreeModel` column number to use for setting the list item strings:

```
comboboxentry.set_text_column(text_column)
text_column = comboboxentry.get_text_column()
```

The text column can also be retrieved and set using the "text-column" property. See Section 16.2.1.2 for more information on the advanced use of the `ComboBoxEntry`.

NOTE



Your application must set the text column for the `ComboBoxEntry` to set the `Entry` contents from the dropdown list. The text column can only be set once, either by using the constructor or by using the set_text_column() method.

When a `ComboBoxEntry` is created it is packed with a new `CellRendererText` which is not accessible. The 'text' attribute for the `CellRendererText` has to be set as a side effect of setting the text column using the set_text_column() method. You can pack additional `CellRenderers` into a `ComboBoxEntry` for display in the dropdown list. See Section 16.2.1.2 for more information.

16.3 ColorButton and FontButton Widgets

16.3.1 ColorButton Widgets

A `ColorButton` widget provides a convenient way of displaying a color in a button that can be clicked to open a `ColorSelectionDialog`. It's useful for displaying and setting colors in a user preference dialog. A `ColorButton` takes care of setting up, displaying and retrieving the result of a `ColorSelectionDialog`. A `ColorButton` is created using the constructor:

```
colorbutton = gtk.ColorButton(color=gtk.gdk.Color(0,0,0))
```

The initial color can be specified using the *color* parameter or set later using the method:

```
colorbutton.set_color(color)
```

The title for the `ColorSelectionDialog` that is displayed when the button is clicked can be set and retrieved using the methods:

```
colorbutton.set_title(title)

title = colorbutton.get_title()
```

The opacity of the color is set using the alpha channel. The following methods get and set the color opacity in the range from 0 (transparent) to 65535 (opaque):

```
alpha = colorbutton.get_alpha()

colorbutton.set_alpha(alpha)
```

By default the alpha is ignored because the "use_alpha" property is `FALSE`. The value of the "use_alpha" property can be set and retrieved using the method:

```
colorbutton.set_use_alpha(use_alpha)

use_alpha = colorbutton.get_use_alpha()
```

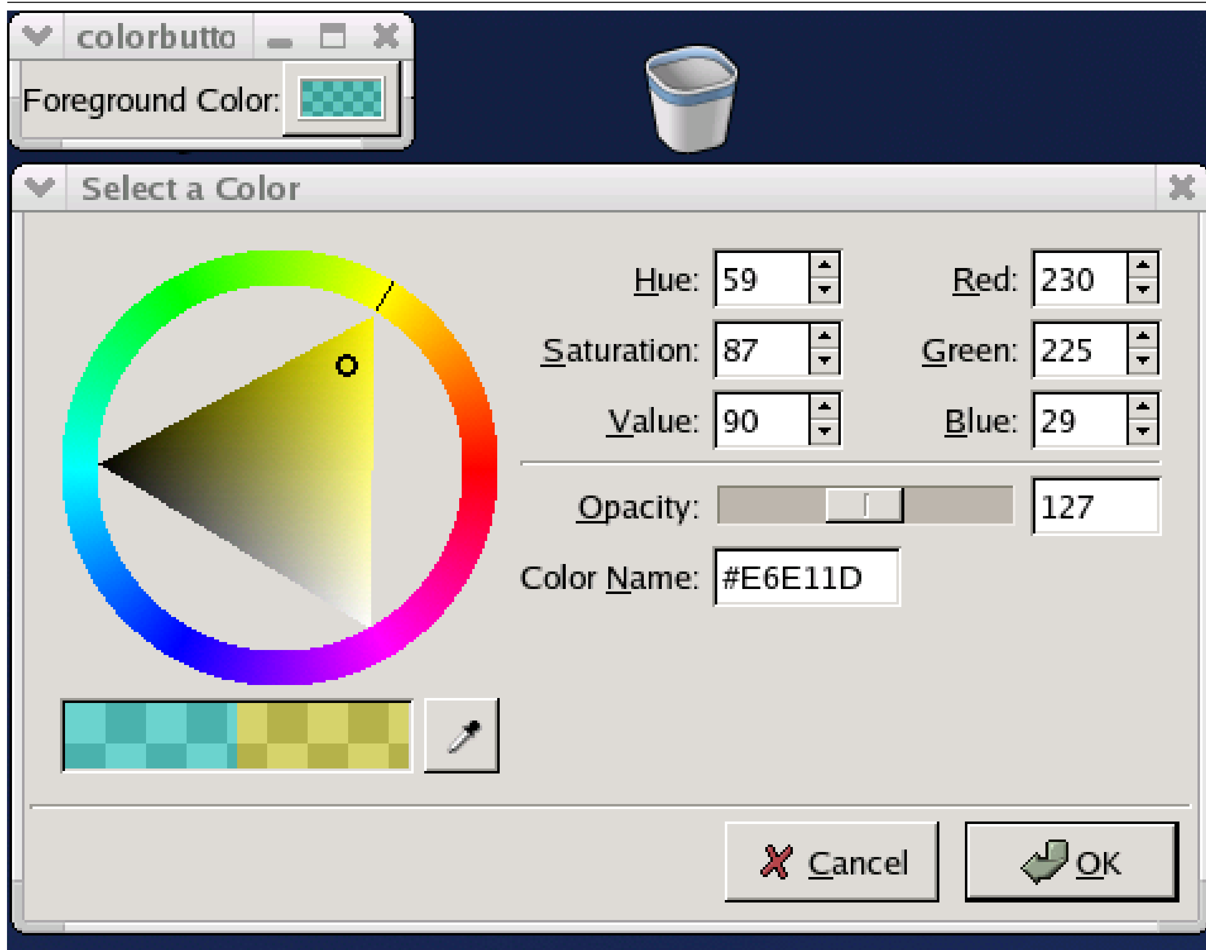
If "use_alpha" is `TRUE` the `ColorSelectionDialog` displays a slider for setting the opacity and displays the color using a checkerboard background.

You can track changes in the selected color by connecting to the "color-set" signal that is emitted when the user sets the color. The signal callback signature is:

```
def color_set_cb(colorbutton, user_data):
```

The example program colorbutton.py illustrates the use of a `ColorButton`. Figure 16.8 shows the program in operation.

Figure 16.8 ColorButton Example



16.3.2 FontButton Widgets

Like the `ColorButton`, the `FontButton` is a convenience widget that provides a display of the currently selected font and, when clicked, opens a `FontSelectionDialog`. A `FontButton` takes care of setting up, displaying and retrieving the result of a `FontSelectionDialog`. A FontButton is created using the constructor:

```
fontbutton = gtk.FontButton(fontname=None)
```

where *`fontname`* is a string specifying the current font for the `FontSelectionDialog`. For example the font name can be specified like 'Sans 12', 'Sans Bold 14', or 'Monospace Italic 14'. You need to specify the font family and size at minimum.

The current font can also be set and retrieved using the following methods:

```
result = fontbutton.set_font_name(fontname)

fontname = fontbutton.get_font_name()
```

where *`result`* returns `TRUE` or `FALSE` to indicate whether the font was successfully set. The `FontButton` has a number of properties and associated methods that affect the display of the current font in the `FontButton`. The "show-size" and show-style" properties contain boolean values that control whether the font size and style will be displayed in the button label. The following methods set and retrieve the value of these properties:

```
fontbutton.set_show_style(show_style)
show_style = fontbutton.get_show_style()

fontbutton.set_show_size(show_size)
show_size = fontbutton.get_show_size()
```

Alternatively, you can have the current font size and style used by the label to directly illustrate the font selection. The "use-size" and "use-font" properties and the associated methods:

```
fontbutton.set_use_font(use_font)
use_font = fontbutton.get_use_font()

fontbutton.set_use_size(use_size)
use_size = fontbutton.get_use_size()
```

Using the current font in the label seems like a useful illustration technique in spite of the inevitable changes in size of the button but using the selected size doesn't seem as useful especially when using really large or small font sizes. Note if you set "use-font" or "use-size" to `TRUE` and later set them to `FALSE`, the last set font and size will be retained. For example, if "use-font" and "use-size" are `TRUE` and the current font is `Monospace Italic 20`, the `FontButton` label is displayed using `Monospace Italic 20`; then if "use-font" and "use-size" are set to `FALSE` and then the current font is changed to `Sans 12` the label will still be displayed in `Monospace Italic 20`. Use the example program fontbutton.py to see how this works.

Finally, the title of the `FontSelectionDialog` can be set and retrieved using the methods:

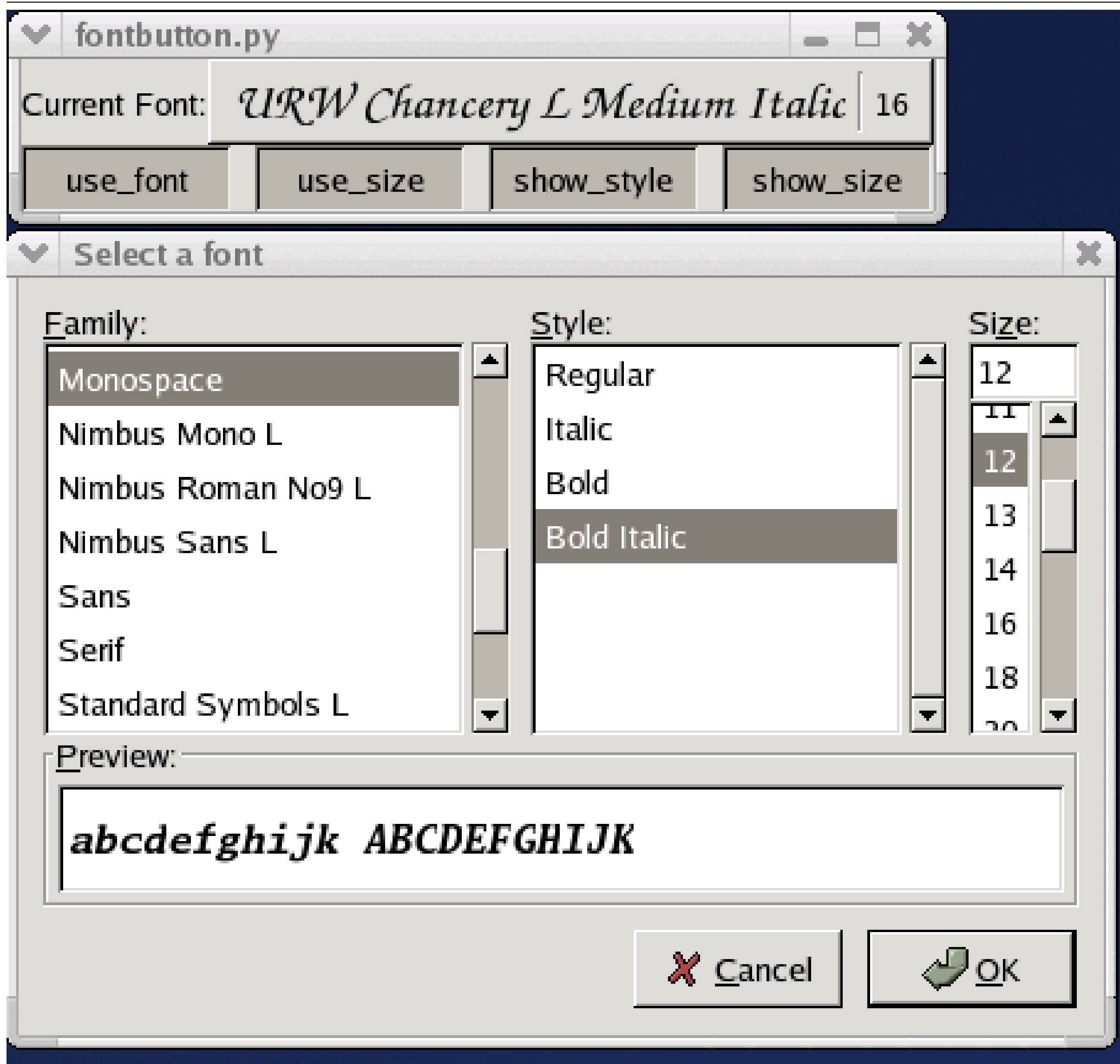
```
fontbutton.set_title(title)

title = fontbutton.get_title()
```

Like the `ColorButton`, you can track changes in the current font by connecting to the "font-set" signal that is emitted when the user sets the font. The signal callback signature is:

```
def font_set_cb(fontbutton, user_data):
```

The example program fontbutton.py illustrates the use of a `FontButton`. You can set the "use-font", "use-size", "show-size" and "show-style" properties using toggle buttons. Figure 16.9 shows the program in operation.

Figure 16.9 FontButton Example



16.4 EntryCompletion Objects

An `EntryCompletion` is an object that is used with an `Entry` widget to provide completion functionality. As the user types into the `Entry` the `EntryCompletion` will popup a window with a set of strings matching the `Entry` text.

An `EntryCompletion` is created using the constructor:

```
completion = gtk.EntryCompletion()
```

You can use the `Entry` method set_completion() to associate an `EntryCompletion` with an `Entry`:

```
entry.set_completion(completion)
```

The strings used by the `EntryCompletion` for matching are retrieved from a `TreeModel` (usually a `ListStore`) that must be set using the method:

```
completion.set_model(model)
```

The `EntryCompletion` implements the `CellLayout` interface that is similar to the `TreeViewColumn` in managing the display of the `TreeModel` data. The following convenience method sets up an `EntryCompletion` in the most common configuration - a list of strings:

```
completion.set_text_column(column)
```

This method is equivalent to the following:

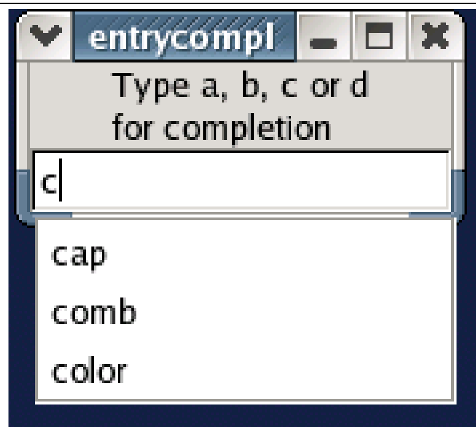
```
cell = CellRendererText()
completion.pack_start(cell)
completion.add_attribute(cell, 'text', column)
```

To set the number of characters that must be entered before the `EntryCompletion` starts matching you can use the method:

```
completion.set_minimum_key_length(length)
```

The example program entrycompletion.py demonstrates the use of the `EntryCompletion`. Figure 16.10 illustrates the program in operation.

Figure 16.10 EntryCompletion



The example program starts with a small number of completion strings that can be increased by typing into the entry field and pressing the **Enter** key. If the string is unique it is added to the list of completion strings.

The built-in match function is a case insensitive string comparison function. If you need a more specialized match function, you can use the following method to install your own match function:

```
completion.set_match_func(func, user_data)
```

The signature of *func* is:

```
def func(completion, key_string, iter, data):
```

where *key_string* contains the current contents of the `Entry`, *iter* is a `TreeIter` pointing at a row in the associated `TreeModel`, and *data* is *user_data*. *func* should return `TRUE` if the row's completion string should be displayed.

The simple example code snippet below uses a match function to display completion names that begin with the entry contents and have the given suffix, in this case, a name ending in `.png` for a `PNG` file.

```
...
completion.set_match_func(end_match, (0, '.png'))
...
def end_match(completion, entrystr, iter, data):
    column, suffix = data
    model = completion.get_model()
    modelstr = model[iter][column]
    return modelstr.startswith(entrystr) and modelstr.endswith(suffix)
...
```

For example if the user types 'foo' and the completion model contains strings like 'foobar.png', smiley.png', 'foot.png' and 'foo.tif', the 'foobar.png' and 'foot.png' strings would be displayed as completions.

16.5 Expander Widgets

The Expander widget is a fairly simple container widget that can reveal or hide its child widget by clicking on a triangle similar to the triangle in a TreeView. A new Expander is created using the constructor:

```
expander = gtk.Expander(label=None)
```

where *label* is a string to be used as the expander label. If *label* is None or not specified, no label is created. Alternatively, you can use the function:

```
expander = gtk.expander_new_with_mnemonic(label=None)
```

that sets the character in label preceded by an underscore as a mnemonic keyboard accelerator.
The Expander widget uses the Container API to add and remove its child widget:

```
expander.add(widget)

expander.remove(widget)
```

The child widget can be retrieved using the Bin "child" attribute or the get_child() method.
The setting that controls the interpretation of label underscores can be retrieved and changed using the methods:

```
use_underline = expander.get_use_underline()

expander.set_use_underline(use_underline)
```

If you want to use Pango markup (see the Pango Markup reference for more detail) in the label string, use the following methods to set and retrieve the setting of the "use-markup" property:

```
expander.set_use_markup(use_markup)

use_markup = expander.get_use_markup()
```

Finally, you can use any widget as the label widget using the following method:

```
expander.set_label_widget(label_widget)
```

This allows you, for example, to use an HBox packed with an Image and a Label as the Expander label.
The state of the Expander can be retrieved and set using the methods:

```
expanded = expander.get_expanded()

expander.set_expanded(expanded)
```

If *expanded* is TRUE the child widget is revealed.
In most cases the Expander automatically does exactly what you want when revealing and hiding the child widget. In some cases your application might want to create a child widget at expansion time. The "notify::expanded" signal can be used to track changes in the state of the expander triangle. The signal handler can then create or change the child widget as needed.
The example program expander.py demonstrates the use of the Expander. Figure 16.11 illustrates the program in operation:

Figure 16.11 Expander Widget

The program creates a Label containing the current time and shows it when the expander is expanded.

16.6 File Selections using FileChooser-based Widgets

The new way to select files in PyGTK 2.4 is to use the variants of the `FileChooser` widget. The two objects that implement this new interface in PyGTK 2.4 are `FileChooserWidget` and `FileChooserDialog`. The latter is the complete dialog with the window and easily defined buttons. The former is a widget useful for embedding within another widget.

Both the `FileChooserWidget` and `FileChooserDialog` possess the means for navigating the filesystem tree and selecting files. The view of the widgets depends on the action used to open a widget.

To create a new file chooser dialog to select an existing file (as in typical File → Open option of a typical application), use:

```
chooser = gtk.FileChooserDialog(title=None,action=gtk.FILE_CHOOSER_ACTION_OPEN,
                                buttons=(gtk.STOCK_CANCEL,gtk.RESPONSE_CANCEL, ↩
                                    gtk.STOCK_OPEN,gtk.RESPONSE_OK))
```

To create a new file chooser dialog to select a new file name (as in the typical File → Save as option of a typical application), use:

```
chooser = gtk.FileChooserDialog(title=None,action=gtk.FILE_CHOOSER_ACTION_SAVE,
                                buttons=(gtk.STOCK_CANCEL,gtk.RESPONSE_CANCEL, ↩
                                    gtk.STOCK_OPEN,gtk.RESPONSE_OK))
```

In the above examples, the two buttons (the stock Cancel and Open items) are created and connected to their respective responses (stock Cancel and OK responses).

To set the folder displayed in the file chooser, use the method:

```
chooser.set_current_folder(pathname)
```

To set the suggested file name as if it was typed by a user (the typical File → Save Assituation), use the method:

```
chooser.set_current_name(name)
```

The above method does not require the filename to exist. If you want to preselect a particular existing file (as in the File → Open situation), you should use the method:

```
chooser.set_filename(filename)
```

To obtain the filename that the user has entered or clicked on, use this method:

```
filename = chooser.get_filename()
```

It is possible to allow multiple file selections (only for the `gtk.FILE_CHOOSER_ACTION_OPEN` action) by using the method:

```
chooser.set_select_multiple(select_multiple)
```

where *`select_mutiple`* should be `TRUE` to allow multiple selections. In this case, you will need to use the following method to retrieve a list of the selected filenames:

```
filenames = chooser.get_filenames()
```

An important feature of all file choosers is the ability to add file selection filters. The filter may be added by the method:

```
chooser.add_filter(filter)
```

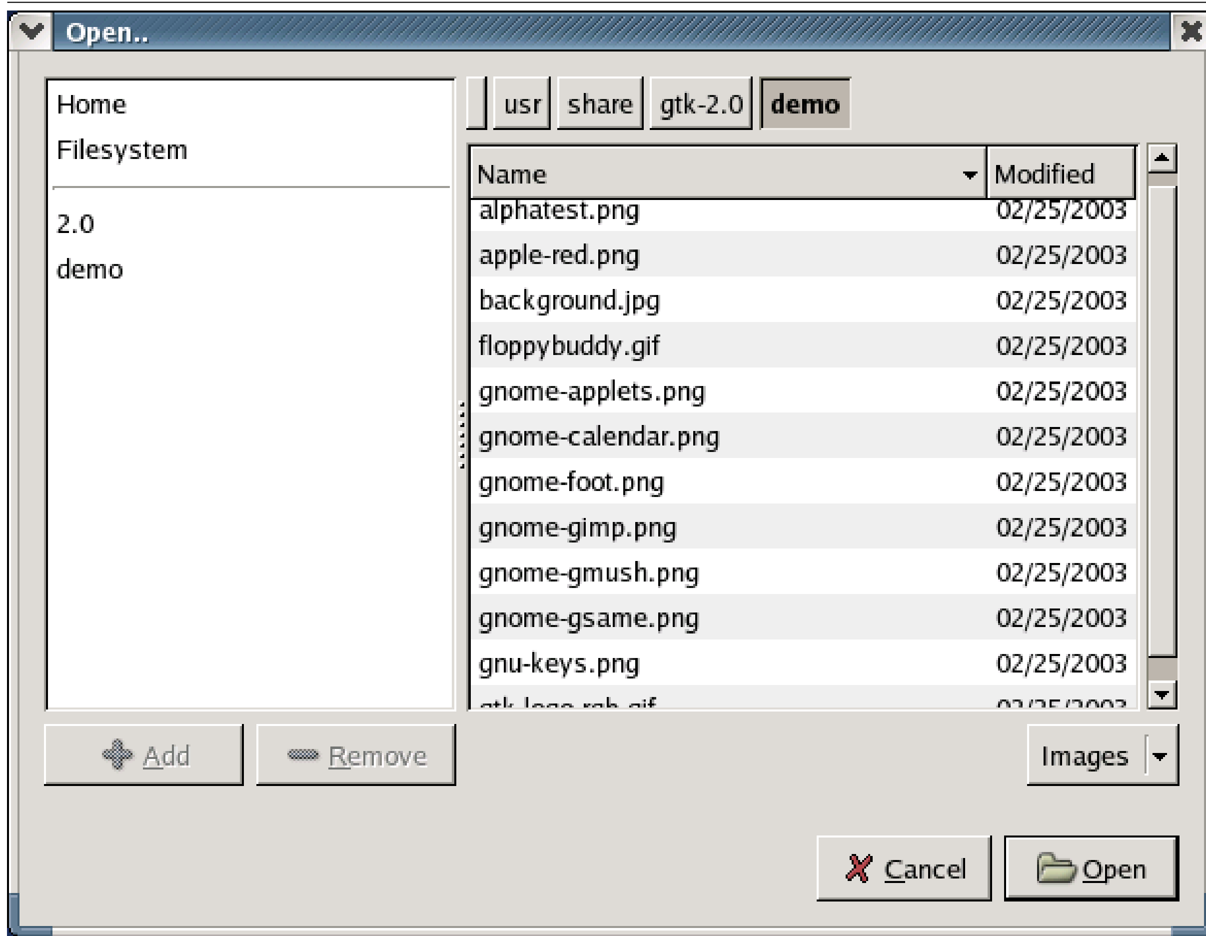
In the example above, *`filter`* must be an instance of the `FileFilter`class.

The left panel of the file chooser lists some shortcut folders such as Home, Filesystem, CDROM, etc. You may add a folder to the list of these shortcuts and remove it from the list by using these methods:

```
chooser.add_shortcut_folder(folder)
chooser.remove_shortcut_folder(folder)
```

where *`folder`* is the pathname of folder. The filechooser.py example program illustrates the use of the filechooser widget. Figure 16.12 shows the resulting display:

Figure 16.12 File Selection Example



The source code for the filechooser.py example program is:

```
1   #!/usr/bin/env python
2
3   # example filechooser.py
4
5   import pygtk
6   pygtk.require('2.0')
7
8   import gtk
9
10  # Check for new pygtk: this is new class in PyGtk 2.4
11  if gtk.pygtk_version < (2,3,90):
12     print "PyGtk 2.3.90 or later required for this example"
13     raise SystemExit
14
15  dialog = gtk.FileChooserDialog("Open..",
16                                 None,
17                                 gtk.FILE_CHOOSER_ACTION_OPEN,
18                                 (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
19                                  gtk.STOCK_OPEN, gtk.RESPONSE_OK))
20  dialog.set_default_response(gtk.RESPONSE_OK)
21
22  filter = gtk.FileFilter()
23  filter.set_name("All files")
24  filter.add_pattern("*")
25  dialog.add_filter(filter)
26
27  filter = gtk.FileFilter()
28  filter.set_name("Images")
```

```
29   filter.add_mime_type("image/png")
30   filter.add_mime_type("image/jpeg")
31   filter.add_mime_type("image/gif")
32   filter.add_pattern("*.png")
33   filter.add_pattern("*.jpg")
34   filter.add_pattern("*.gif")
35   filter.add_pattern("*.tif")
36   filter.add_pattern("*.xpm")
37   dialog.add_filter(filter)
38
39   response = dialog.run()
40   if response == gtk.RESPONSE_OK:
41       print dialog.get_filename(), 'selected'
42   elif response == gtk.RESPONSE_CANCEL:
43       print 'Closed, no files selected'
44   dialog.destroy()
```

16.7 The UIManager

16.7.1 Overview

The `UIManager` provides a way to create menus and toolbars from an XML-like description. The `UIManager` uses `ActionGroup` objects to manage the `Action` objects providing the common substructure for the menu and toolbar items.

Using the `UIManager` you can dynamically merge and demerge multiple UI descriptions and actions. This allows you to modify the menus and toolbars when the mode changes in the application (for example, changing from text editing to image editing), or when new plug-in features are added or removed from your application.

A `UIManager` can be used to create the menus and toolbars for an application user interface as follows:

- Create a `UIManager` instance
- Extract the `AccelGroup` from the `UIManager` and add it to the top level `Window`
- Create the `ActionGroup` instances and populate them with the appropriate `Action` instances.
- Add the `ActionGroup` instances to the `UIManager` in the order that the `Action` instances should be found.
- Add the UI XML descriptions to the `UIManager`. Make sure that all `Actions` referenced by the descriptions are available in the `UIManager` `ActionGroup` instances.
- Extract references to the menubar, menu and toolbar widgets by name for use in building the user interface.
- Dynamically modify the user interface by adding and removing UI descriptions and by adding, rearranging and removing the associated `ActionGroup` instances.

16.7.2 Creating a UIManager

A `UIManager` instance is created by the constructor:

```
uimamager = gtk.UIManager()
```

A new `UIManager` is created with an associated `AccelGroup` that can be retrieved using the method:

```
accelgroup = uimanager.get_accel_group()
```

The `AccelGroup` should be added to the top level window of the application so that the `Action` accelerators can be used by your users. For example:

```
window = gtk.Window()
...
uimanager = gtk.UIManager()
accelgroup = uimanager.get_accel_group()
window.add_accel_group(accelgroup)
```

16.7.3 Adding and Removing ActionGroups

As described in Section 16.1.2, `ActionGroups` can be populated with `Actions` by using the add_actions(), add_toggle_actions() and add_radio_actions() convenience methods. An `ActionGroup` can be used by a `UIManager` after it has been added to its `ActionGroup` list by using the method:

```
uimanager.insert_action_group(action_group, pos)
```

where *pos* is the index of the position where *action_group* should be inserted. A `UIManager` may contain several `ActionGroups` with duplicate `Action` names. The order of the `ActionGroup` objects is important because the lookup of an `Action` stops when the first `Action` with the given name is encountered. This means that actions in earlier `ActionGroup` objects mask those in later `ActionGroup` objects.

The actions referenced in a UI XML description must be added to a `UIManager` before the description can be added to the `UIManager`.

An `ActionGroup` can be removed from a `UIManager` by using the method:

```
uimanager.remove_action_group(action_group)
```

A list of the `ActionGroup` objects associated with a `UIManager` can be retrieved using the method:

```
actiongrouplist = uimanager.get_action_groups()
```

16.7.4 UI Descriptions

The UI descriptions accepted by `UIManager` are simple XML definitions with the following elements:

ui The root element of a UI description. It can be omitted. Can contain **menubar**, **popup**, **toolbar** and **accelerator** elements.

menubar A top level element describing a `MenuBar` structure that can contain **MenuItem**, **separator**, **placeholder** and **menu** elements. It has an optional *name* attribute. If *name* is not specified, "menubar" is used as the name.

popup A top level element describing a popup `Menu` structure that can contain **menuitem**, **separator**, **placeholder**, and **menu** elements. It has an optional *name* attribute. If *name* is not specified, "popup" is used as the name.

toolbar A top level element describing a `Toolbar` structure that can contain **toolitem**, **separator** and **placeholder** elements. It has an optional *name* attribute. If *name* is not specified, "toolbar" is used as the name.

placeholder An element identifying a position in a **menubar**, **toolbar**, **popup** or **menu**. A placeholder can contain **menuitem**, **separator**, **placeholder**, and **menu** elements. **Placeholder** elements are used when merging UI descriptions to allow, for example, a menu to be built up from UI descriptions using common **placeholder** names. It has an optional *name* attribute. If *name* is not specified, "placeholder" is used as the name.

menu An element describing a `Menu` structure that can contain **menuitem**, **separator**, **placeholder**, and **menu** elements. A **menu** element has a required attribute *action* that names an `Action` object to be used to create the `Menu`. It also has optional *name* and *position* attributes. If *name* is not specified, the *action* name is used as the name. The *position* attribute can have either the value "top" or "bottom" with "bottom" the default if *position* is not specified.

menuitem An element describing a `MenuItem`. A **menuitem** element has a required attribute *action* that names an `Action` object to be used to create the `MenuItem`. It also has optional *name* and *position* attributes. If *name* is not specified, the *action* name is used as the name. The *position* attribute can have either the value "top" or "bottom" with "bottom" the default if *position* is not specified.

toolitem An element describing a toolbar `ToolItem`. A **toolitem** element has a required attribute *action* that names an `Action` object to be used to create the `Toolbar`. It also has optional *name* and *position* attributes. If *name* is not specified, the *action* name is used as the name. The *position* attribute can have either the value "top" or "bottom" with "bottom" the default if *position* is not specified.

separator An element describing a `SeparatorMenuItem` or a `SeparatorToolItem` as appropriate.

accelerator An element describing a keyboard accelerator. An **accelerator** element has a required attribute *action* that names an `Action` object that defines the accelerator key combination and is activated by the accelerator. It also has an optional *name* attribute. If *name* is not specified, the *action* name is used as the name.

For example, a UI description that could be used to create an interface similar that in Figure 16.4 is:

```
<ui>
  <menubar name="MenuBar">
    <menu action="File">
      <menuitem action="Quit"/>
    </menu>
    <menu action="Sound">
      <menuitem action="Mute"/>
    </menu>
    <menu action="RadioBand">
      <menuitem action="AM"/>
      <menuitem action="FM"/>
      <menuitem action="SSB"/>
    </menu>
  </menubar>
  <toolbar name="Toolbar">
    <toolitem action="Quit"/>
    <separator/>
    <toolitem action="Mute"/>
    <separator name="sep1"/>
    <placeholder name="RadioBandItems">
      <toolitem action="AM"/>
      <toolitem action="FM"/>
      <toolitem action="SSB"/>
    </placeholder>
  </toolbar>
</ui>
```

Note that this description just uses the **action** attribute names for the names of most elements rather than specifying *name* attributes. Also I would recommend not specifying the **ui** element as it appears to be unnecessary.

The widget hierarchy created using a UI description is very similar to the XML element hierarchy except that **placeholder** elements are merged into their parents.

A widget in the hierarchy created by a UI description can be accessed using its path which is composed of the name of the widget element and its ancestor elements joined by slash ("/") characters. For example using the above description the following are valid widget paths:

```
/MenuBar
/MenuBar/File/Quit
/MenuBar/RadioBand/SSB
/Toolbar/Mute
/Toolbar/RadioBandItems/FM
```

Note that the **placeholder** name must be included in the path. Usually you just access the top level widgets (for example, "/MenuBar" and "/Toolbar") but you may need to access a lower level widget to, for example, change a property.

16.7.5 Adding and Removing UI Descriptions

Once a `UIManager` is set up with an `ActionGroup` a UI description can be added and merged with the existing UI by using one of the following methods:

```
merge_id = uimanager.add_ui_from_string(buffer)

merge_id = uimanager.add_ui_from_file(filename)
```

where *buffer* is a string containing a UI description and *filename* is the file containing a UI description. Both methods return a *merge_id* which is a unique integer value. If the method fails, the `GError` exception is raised. The *merge_id* can be used to remove the UI description from the `UIManager` by using the method:

```
uimanager.remove_ui(merge_id)
```

The same methods can be used more than once to add additional UI descriptions that will be merged to provide a combined XML UI description. Merged UIs will be discussed in more detail in Section 16.7.8 section.

A single UI element can be added to the current UI description by using the method:

```
uimanager.add_ui(merge_id, path, name, action, type, top)
```

where *merge_id* is a unique integer value, *path* is the path where the new element should be added, *action* is the name of an `Action` or `None` to add a **separator**, *type* is the element type to be added and *top* is a boolean value. If *top* is `TRUE` the element will be added before its siblings, otherwise it is added after.

merge_id should be obtained from the method:

```
merge_id = uimanager.new_merge_id()
```

The integer values returned from the new_merge_id() method are monotonically increasing.

path is a string composed of the name of the element and the names of its ancestor elements separated by slash ("/") characters but not including the optional root node "/ui". For example, "/MenuBar/RadioBand" is the path of the **menu** element named "RadioBand" in the following UI description:

```
<menubar name="MenuBar">
  <menu action="RadioBand">
  </menu>
</menubar>
```

The value of *type* must be one of:

`gtk.UI_MANAGER_AUTO` The type of the UI element (menuitem, toolitem or separator) is set according to the context.

`gtk.UI_MANAGER_MENUBAR` A menubar.

`gtk.UI_MANAGER_MENU` A menu.

`gtk.UI_MANAGER_TOOLBAR` A toolbar.

`gtk.UI_MANAGER_PLACEHOLDER` A placeholder.

`gtk.UI_MANAGER_POPUP` A popup menu.

`gtk.UI_MANAGER_MENUITEM` A menuitem.

`gtk.UI_MANAGER_TOOLITEM` A toolitem.

`gtk.UI_MANAGER_SEPARATOR` A separator.

`gtk.UI_MANAGER_ACCELERATOR` An accelerator.

add_ui() fails silently if the element is not added. Using add_ui() is so low level that you should always try to use the convenience methods add_ui_from_string() and add_ui_from_file() instead.

Adding a UI description or element causes the widget hierarchy to be updated in an idle function. You can make sure that the widget hierarchy has been updated before accessing it by calling the method:

```
uimanager.ensure_update()
```

16.7.6 Accessing UI Widgets

You access a widget in the UI widget hierarchy by using the method:

```
widget = uimanager.get_widget(path)
```

where *path* is a string containing the name of the widget element and it's ancestors as described in Section 16.7.4.

For example, given the following UI description:

```
<menubar name="MenuBar">
  <menu action="File">
    <menuitem action="Quit"/>
  </menu>
  <menu action="Sound">
    <menuitem action="Mute"/>
  </menu>
  <menu action="RadioBand">
    <menuitem action="AM"/>
    <menuitem action="FM"/>
    <menuitem action="SSB"/>
  </menu>
</menubar>
<toolbar name="Toolbar">
  <toolitem action="Quit"/>
  <separator/>
  <toolitem action="Mute"/>
  <separator name="sep1"/>
  <placeholder name="RadioBandItems">
    <toolitem action="AM"/>
    <toolitem action="FM"/>
    <toolitem action="SSB"/>
  </placeholder>
</toolbar>
```

added to the `UIManager` *uimanager*, you can access the `MenuBar` and `Toolbar` for use in an application `Window` by using the following code fragment:

```
window = gtk.Window()
vbox = gtk.VBox()
menubar = uimanager.get_widget('/MenuBar')
toolbar = uimanager.get_widget('/Toolbar')
vbox.pack_start(meunbar, False)
vbox.pack_start(toolbar, False)
```

Likewise the lower level widgets in the hierarchy are accessed by using their paths. For example the `RadioToolButton` named "SSB" is accessed as follows:

```
ssb = uimanager.get_widget('/Toolbar/RadioBandItems/SSB')
```

As a convenience all the top level widgets of a type can be retrieved using the method:

```
toplevels = uimanager.get_toplevels(type)
```

where *type* specifies the type of widgets to return using a combination of the flags: `gtk.UI_MANAGER_MENUBAR`, `gtk.UI_MANAGER_TOOLBAR` and `gtk.UI_MANAGER_POPUP`. You can use the gtk.Widget.get_name() method to determine which top level widget you have.

You can retrieve the `Action` that is used by the proxy widget associated with a UI element by using the method:

```
action = uimanager_get_action(path)
```

where *path* is a string containing the path to a UI element in *uimanager*. If the element has no associated Action, None is returned.

16.7.7 A Simple UIManager Example

A simple example program illustrating the use of UIManager is uimanager.py. Figure 16.13 illustrates the program in operation.

Figure 16.13 Simple UIManager Example

The uimanager.py example program uses the XML description of Section 16.7.6. The text of the two labels are changed in response to the activation of the "Mute" ToggleAction and "AM", "FM" and "SSB" RadioActions. All the actions are contained in a single ActionGroup allowing the sensitivity and visibility of all the action proxy widgets to be toggled on and off by using the "Sensitive" and "Visible" toggle buttons. The use of the **placeholder** element will be described in Section 16.7.8.

16.7.8 Merging UI Descriptions

The merging of UI descriptions is done based on the name of the XML elements. As noted above the individual elements in the hierarchy can be accessed using a pathname consisting of the element name and the names of its ancestors. For example, using the UI description in Section 16.7.4 the "AM" **toolitem** element has the pathname "/Toolbar/RadioBandItems/AM" while the "FM" **menuitem** element has the pathname "/MenuBar/RadioBand/FM".

If a UI description is merged with that UI description the elements are added as siblings to the existing elements. For example, if the UI description:

```
<menubar name="MenuBar">
  <menu action="File">
    <menuitem action="Save" position="top"/>
    <menuitem action="New" position="top"/>
  </menu>
  <menu action="Sound">
    <menuitem action="Loudness"/>
  </menu>
  <menu action="RadioBand">
    <menuitem action="CB"/>
    <menuitem action="Shortwave"/>
  </menu>
</menubar>
<toolbar name="Toolbar">
  <toolitem action="Save" position="top"/>
  <toolitem action="New" position="top"/>
  <separator/>
```

```
  <toolitem action="Loudness"/>
  <separator/>
  <placeholder name="RadioBandItems">
    <toolitem action="CB"/>
    <toolitem action="Shortwave"/>
  </placeholder>
</toolbar>
```

is added to our example UI description:

```
<menubar name="MenuBar">
  <menu action="File">
    <menuitem action="Quit"/>
  </menu>
  <menu action="Sound">
    <menuitem action="Mute"/>
  </menu>
  <menu action="RadioBand">
    <menuitem action="AM"/>
    <menuitem action="FM"/>
    <menuitem action="SSB"/>
  </menu>
</menubar>
<toolbar name="Toolbar">
  <toolitem action="Quit"/>
  <separator/>
  <toolitem action="Mute"/>
  <separator name="sep1"/>
  <placeholder name="RadioBandItems">
    <toolitem action="AM"/>
    <toolitem action="FM"/>
    <toolitem action="SSB"/>
  </placeholder>
</toolbar>
```

the following merged UI description will be created:

```
<menubar name="MenuBar">
  <menu name="File" action="File">
    <menuitem name="New" action="New"/>
    <menuitem name="Save" action="Save"/>
    <menuitem name="Quit" action="Quit"/>
  </menu>
  <menu name="Sound" action="Sound">
    <menuitem name="Mute" action="Mute"/>
    <menuitem name="Loudness" action="Loudness"/>
  </menu>
  <menu name="RadioBand" action="RadioBand">
    <menuitem name="AM" action="AM"/>
    <menuitem name="FM" action="FM"/>
    <menuitem name="SSB" action="SSB"/>
    <menuitem name="CB" action="CB"/>
    <menuitem name="Shortwave" action="Shortwave"/>
  </menu>
</menubar>
<toolbar name="Toolbar">
  <toolitem name="New" action="New"/>
  <toolitem name="Save" action="Save"/>
  <toolitem name="Quit" action="Quit"/>
  <separator/>
  <toolitem name="Mute" action="Mute"/>
  <separator name="sep1"/>
  <placeholder name="RadioBandItems">
    <toolitem name="AM" action="AM"/>
    <toolitem name="FM" action="FM"/>
```

```
      <toolitem name="SSB" action="SSB"/>
      <toolitem name="CB" action="CB"/>
      <toolitem name="Shortwave" action="Shortwave"/>
    </placeholder>
    <separator/>
    <toolitem name="Loudness" action="Loudness"/>
    <separator/>
  </toolbar>
```

Examining the merged XML you can see that the "New" and "Save" **menuitem** elements have been merged before the "Quit" element as a result of the "position" attribute being set to "top" which means the element should be prepended. Likewise, the "New" and "Save" **toolitem** elements have been prepended to "Toolbar". Note that the "New" and "Save" elements are reversed by the merging process.

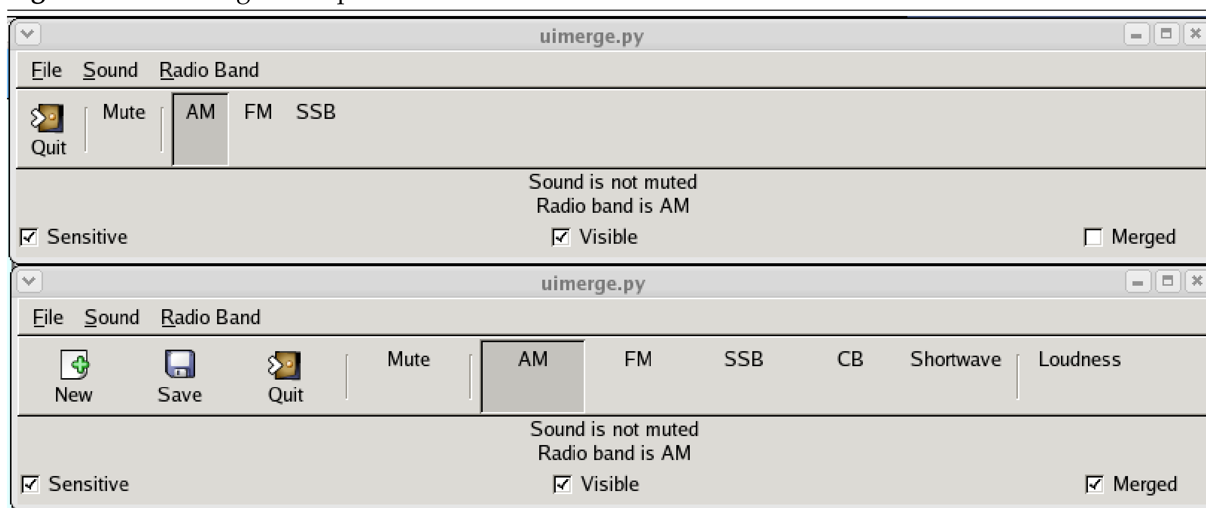
The "Loudness" **toolitem** element is appended to the "Toolbar" elements and appears last in the merged UI description even though it's not last in its UI description. The "RadioBandItems" **placeholder** element in both UI descriptions combines the "CB" and "Shortwave" **toolitem** elements with the "AM", "FM", and "SSB" elements. If the "RadioBandItems" **placeholder** element was not used the "CB" and "Shortwave" elements would have been placed after the "Loudness" element.

A representation of the UI description used by a `UIManager` can be retrieved using the method:

```
uidesc = uimanager.get_ui()
```

The uimerge.py example program demonstrates the merging of the above UI descriptions. Figure 16.14 illustrates the unmerged and merged UIs:

Figure 16.14 UIMerge Example



The example program uses three `ActionGroup` objects:

- `Action` objects for the "File", "Sound" and "Radio Band" menus
- `Action` objects for the "Quit", "Mute", "AM", "FM", "SSB" and "Radio Band" menus
- `Action` objects for the "Loudness", "CB" and "Shortwave" elements

The "Sensitive" and Visible" `ToggleButton` widgets control the sensitivity and visibility of only the second `ActionGroup`.

16.7.9 UIManager Signals

The `UIManager` has a couple of interesting signals that your application can connect to. The "actions-changed" signal is emitted when an `ActionGroup` is added or removed from a `UIManager`. The signature of the callback is:

```
def callback(uimanager, ...)
```

The "add-widget" signal is emitted when a proxy `MenuBar` or `Toolbar` widget is created. The callback signature is:

```
def callback(uimanager, widget, ...)
```

where *`widget`* is the newly created widget.

Chapter 17

Undocumented Widgets

These all require authors! :) Please consider contributing to our tutorial.

If you must use one of these widgets that are undocumented, I strongly suggest you take a look at the *.c files in the PyGTK distribution. PyGTK's method names are very descriptive. Once you have an understanding of how things work, it's not difficult to figure out how to use a widget simply by looking at its method definitions. This, along with a few examples from others' code, and it should be no problem.

When you do come to understand all the methods of a new undocumented widget, please consider writing a tutorial on it so others may benefit from your time.

17.1 Accel Label

17.2 Option Menu

17.3 Menu Items

17.3.1 Check Menu Item

17.3.2 Radio Menu Item

17.3.3 Separator Menu Item

17.3.4 Tearoff Menu Item

17.4 Curves

17.5 Gamma Curve

Chapter 18

Setting Widget Attributes

This describes the methods used to operate on widgets (and objects). These can be used to set style, padding, size, etc.

The method:

```
widget.activate()
```

causes the widget to emit the "activate" signal.

The method:

```
widget.set_sensitive(sensitive)
```

sets the sensitivity of the widget (i.e. does it react to events). if *sensitive* is TRUE the widget will receive events; if FALSE the widget will not receive events. A widget that is insensitive is usually displayed "grayed out".

The method:

```
widget.set_size_request(width, height)
```

sets the widget size to the given *width* and *height*.

18.1 Widget Flag Methods

The methods:

```
widget.set_flags(flags)

widget.unset_flags(flags)

flags = widget.flags()
```

set, unset and get the gtk.Object and gtk.Widget flags. *flags* can be any of the standard flags:

```
IN_DESTRUCTION
FLOATING
RESERVED_1
RESERVED_2
TOPLEVEL
NO_WINDOW
REALIZED
MAPPED
VISIBLE
SENSITIVE
PARENT_SENSITIVE
CAN_FOCUS
HAS_FOCUS
CAN_DEFAULT
HAS_DEFAULT
HAS_GRAB
```

```
RC_STYLE
COMPOSITE_CHILD
NO_REPARENT
APP_PAINTABLE
RECEIVES_DEFAULT
DOUBLE_BUFFERED
```

The method:

```
widget.grab_focus()
```

allows a widget to grab the focus assuming that it has the `CAN_FOCUS` flag set.

18.2 Widget Display Methods

The methods:

```
widget.show()

widget.show_all()

widget.hide()

widget.hide_all()

widget.realize()

widget.unrealize()

widget.map()

widget.unmap()
```

manage the display of the *widget*.

The show() method arranges to display the widget by using the realize() and map() methods.

The hide() method arranges to remove the widget from the display and also unmaps it using the unmap() method if necessary.

The show_all() and hide_all() methods arrange to show or hide the widget and all its children.

The realize() method arranges to allocate resources to the widget including its window.

The unrealize() method releases the widget window and other resources. Unrealizing a widget will also hide and unmap it.

The map() method arranges to allocate space on the display for the widget; this only applies to widgets that need to be handled by the window manager. Mapping a widget will also cause it to be realized if necessary.

The unmap() method removes a widget from the display and will also hide it if necessary.

18.3 Widget Accelerators

The following methods:

```
widget.add_accelerator(accel_signal, accel_group, accel_key, accel_mods,  ↩
    accel_flags)

widget.remove_accelerator(accel_group, accel_key, accel_mods)
```

add and remove accelerators from a `gtk.AcceleratorGroup` that must be attached to the top level widget to handle the accelerators.

The *accel_signal* is a signal that is valid for the *widget* to emit.

The *accel_key* is a keyboard key to use as the accelerator.

The *accel_mods* are modifiers to add to the *accel_key* (e.g. **Shift**, **Control**, etc.):

```
SHIFT_MASK
LOCK_MASK
CONTROL_MASK
MOD1_MASK
MOD2_MASK
MOD3_MASK
MOD4_MASK
MOD5_MASK
BUTTON1_MASK
BUTTON2_MASK
BUTTON3_MASK
BUTTON4_MASK
BUTTON5_MASK
RELEASE_MASK
```

The *accel_flags* set options about how the accelerator information is displayed. Valid values are:

```
ACCEL_VISIBLE          # display the accelerator key in the widget display

ACCEL_LOCKED           # do not allow the accelerator display to change
```

An accelerator group is created by the function:

```
accel_group = gtk.AccelGroup()
```

The *accel_group* is attached to a top level widget with the following method:

```
window.add_accel_group(accel_group)
```

An example of adding an accelerator:

```
menu_item.add_accelerator("activate", accel_group,
                          ord('Q'), gtk.gdk.CONTROL_MASK, gtk.ACCEL_VISIBLE)
```

18.4 Widget Name Methods

The following widget methods set and get the name of a widget:

```
widget.set_name(name)

name = widget.get_name()
```

name is the string that will be associated with the *widget*. This is useful for specifying styles to be used with specific widgets within an application. The name of the widget can be used to narrow the application of the style as opposed to using the widget's class. See Chapter 23 for more details.

18.5 Widget Styles

The following methods get and set the style associated with a widget:

```
widget.set_style(style)

style = widget.get_style()
```

The following function:

```
style = get_default_style()
```

gets the default style.

A style contains the graphics information needed by a widget to draw itself in its various states:

```
STATE_NORMAL         # The state during normal operation.
STATE_ACTIVE         # The widget is currently active, such as a button pushed
STATE_PRELIGHT       # The mouse pointer is over the widget.
STATE_SELECTED       # The widget is selected
STATE_INSENSITIVE    # The widget is disabled
```

A style contains the following attributes:

```
fg          # a list of 5 foreground colors - one for each state
bg          # a list of 5 background colors
light       # a list of 5 colors - created during set_style() method
dark        # a list of 5 colors - created during set_style() method
mid         # a list of 5 colors - created during set_style() method
text        # a list of 5 colors
base        # a list of 5 colors
text_aa     # a list of 5 colors halfway between text/base

black       # the black color
white       # the white color
font_desc   # the default pango font description

xthickness  #
ythickness  #

fg_gc       # a list of 5 graphics contexts - created during set_style() method
bg_gc       # a list of 5 graphics contexts - created during set_style() method
light_gc    # a list of 5 graphics contexts - created during set_style() method
dark_gc     # a list of 5 graphics contexts - created during set_style() method
mid_gc      # a list of 5 graphics contexts - created during set_style() method
text_gc     # a list of 5 graphics contexts - created during set_style() method
base_gc     # a list of 5 graphics contexts - created during set_style() method
black_gc    # a list of 5 graphics contexts - created during set_style() method
white_gc    # a list of 5 graphics contexts - created during set_style() method

bg_pixmap   # a list of 5 GdkPixmaps
```

Each attribute can be accessed directly similar to *style.black* and *style.fg_gc[gtk.STATE_NORMAL]*. All attributes are read-only except for *style.black*, *style.white*, *style.black_gc* and *style.white_gc*.

An existing style can be copied for later modification by using the method:

```
new_style = style.copy()
```

which copies the *style* attributes except for the graphics context lists and the light, dark and mid color lists.

The current style of a widget can be retrieved with:

```
style = widget.get_style()
```

To change the style of a widget (e.g. to change the widget foreground color), the following widget methods should be used:

```
widget.modify_fg(state, color)
widget.modify_bg(state, color)
widget.modify_text(state, color)
widget.modify_base(state, color)
widget.modify_font(font_desc)
widget.set_style(style)
```

Setting the *style* will allocate the style colors and create the graphics contexts. Most widgets will automatically redraw themselves after the style is changed. If *style* is `None` then the widget style will revert to the default style.

Not all style changes will affect the widget. For example, changing the `Label` (see Section 9.1) widget background color will not change the label background color because the `Label` widget does not have its own `gtk.gdk.Window`. The background of the label is dependent on the background color of the

label's parent. The use of an `EventBox` to hold a Label will allow the Label background color to be set. See Section 10.1 for an example.

Chapter 19

Timeouts, IO and Idle Functions

19.1 Timeouts

You may be wondering how you make GTK do useful work when in `main()`. Well, you have several options. Using the following gobject module function you can create a timeout function that will be called every "interval" milliseconds.

```
source_id = gobject.timeout_add(interval, function, ...)
```

The *interval* argument is the number of milliseconds between calls to your function. The *function* argument is the callback you wish to have called. Any arguments after the second are passed to the function as data. The return value is an integer "source_id" which may be used to stop the timeout by calling:

```
gobject.source_remove(source_id)
```

You may also stop the timeout callback function from being called again by returning zero or `FALSE` from your callback. If you want your callback to be called again, it should return `TRUE`.

Your callback should look something like this:

```
def timeout_callback(...):
```

The number of arguments to the callback should match the number of data arguments specified in `timeout_add()`.

19.2 Monitoring IO

You can check for the ability to read from or write to a file (either a Python file or a lower level OS file) and then automatically invoke a callback. This is especially useful for networking applications. The gobject module function:

```
source_id = gobject.io_add_watch(source, condition, callback)
```

where the first argument (*source*) is the open file (Python file object or lower level file descriptor integer) you wish to have watched. The `gobject.io_add_watch()` function uses the lower level file descriptor integer internally but the function will extract it from the Python file object using the fileno() method as needed. The second argument (*condition*) specifies what you want to look for. This may be one of:

```
gobject.IO_IN - There is data ready for reading from your file.

gobject.IO_OUT - The file is ready for writing.

gobject.IO_PRI - There is urgent data to read.

gobject.IO_ERR - Error condition.

gobject.IO_HUP - Hung up (the connection has been broken, usually for
                 pipes and sockets).
```

These are defined in the gobject module. As I'm sure you've figured out already, the third argument is the *callback* you wish to have called when the above conditions are satisfied.

The return value *source_id* may be used to stop the monitoring of the file by using the following function:

```
gobject.source_remove(source_id)
```

The callback function should be similar to:

```
def input_callback(source, condition):
```

where *source* and *condition* are as specified above. The source value will be the lower level file descriptor integer and not the Python file object (i.e. the value that is returned from the Python file method fileno()).

You may also stop the callback function from being called again by returning zero or `FALSE` from your callback. If you want your callback to be called again, it should return `TRUE`.

19.3 Idle Functions

What if you have a function which you want to be called when nothing else is happening ? Use the function:

```
source_id = gobject.idle_add(callback, ...)
```

Any arguments beyond the first (indicated with ...) are passed to the *callback* in order. The *source_id* is returned to provide a reference to the handler.

This function causes GTK to call the specified *callback* function whenever nothing else is happening.

The *callback* signature is:

```
def callback(...):
```

where the arguments passed to the `callback` are the same as those specified in the `gobject.idle_add`() function. As with the other callback functions, returning `FALSE` will stop the idle callback from being called and returning `TRUE` causes the callback function to be run at the next idle time.

An idle function can be removed from the queue by calling the function:

```
gobject.source_remove(source_id)
```

with the *source_id* returned from the `gobject.idle_add`() function.

Chapter 20

Advanced Event and Signal Handling

20.1 Signal Methods

The signal methods are `gobject.GObject` methods that are inherited by the `gtk.Objects` including all the GTK+ widgets.

20.1.1 Connecting and Disconnecting Signal Handlers

```
handler_id = object.connect(name, cb, cb_args)

handler_id = object.connect_after(name, cb, cb_args)

handler_id = object.connect_object(name, cb, slot_object, cb_args)

handler_id = object.connect_object_after(name, cb, slot_object, cb_args)

object.disconnect(handler_id)
```

The first four methods connect a signal handler (*cb*) to a `gtk.Object` (*object*) for the given signal name. and return a *handler_id* that identifies the connection. *cb_args* is zero or more arguments that will be passed last (in order) to *cb*. The connect_after() and connect_object_after() methods will have their signal handlers called after other signal handlers (including the default handlers) connected to the same object and signal name. Each object signal handler has its own set of arguments that it expects. You have to refer to the GTK+ documentation to figure out what arguments need to be handled by a signal handler though information for the common widgets is available in Appendix A. The general signal handler is similar to:

```
def signal_handler(object, ...., cb_args):
```

Signal handlers that are defined as part of a Python object class (specified in the connect() methods as *self.cb*) will have an additional argument passed as the first argument - the object instance *self*:

```
signal_handler(self, object, ...., cb_args)
```

The connect_object() and connect_object_after() methods call the signal handler with the *slot_object* substituted in place of the *object* as the first argument:

```
def signal_handler(slot_object, ...., func_args):

def signal_handler(self, slot_object, ...., func_args):
```

The disconnect() method destroys the connection between a signal handler and an object signal. The *handler_id* specifies which connection to destroy.

20.1.2 Blocking and Unblocking Signal Handlers

The following methods:

```
object.handler_block(handler_id)

object.handler_unblock(handler_id)
```

block and unblock the signal handler specified by *handler_id*. When a signal handler is blocked it will not be invoked when the signal occurs.

20.1.3 Emitting and Stopping Signals

The following methods:

```
object.emit(name, ...)

object.emit_stop_by_name(name)
```

emit and stop the signal *name* respectively. Emitting the signal causes its default and user defined handlers to be run. The emit_stop_by_name() method will abort the current signal name emission.

20.2 Signal Emission and Propagation

Signal emission is the process whereby GTK+ runs all handlers for a specific object and signal.

First, note that the return value from a signal emission is the return value of the last handler executed. Since event signals are all of type RUN_LAST, this will be the default (GTK+ supplied) handler, unless you connect with the connect_after() method.

The way an event (say "button_press_event") is handled, is:

- Start with the widget where the event occurred.
- Emit the generic "event" signal. If that signal handler returns a value of TRUE, stop all processing.
- Otherwise, emit a specific, "button_press_event" signal. If that returns TRUE, stop all processing.
- Otherwise, go to the widget's parent, and repeat the above two steps.
- Continue until some signal handler returns TRUE, or until the top-level widget is reached.

Some consequences of the above are:

- Your handler's return value will have no effect if there is a default handler, unless you connect with connect_after().
- To prevent the default handler from being run, you need to connect with connect() and use emit_stop_by_name() - the return value only affects whether the signal is propagated, not the current emission.

Chapter 21

Managing Selections

21.1 Selection Overview

One type of interprocess communication supported by X and GTK+ is selections. A selection identifies a chunk of data, for instance, a portion of text, selected by the user in some fashion, for instance, by dragging with the mouse. Only one application on a display (the owner) can own a particular selection at one time, so when a selection is claimed by one application, the previous owner must indicate to the user that selection has been relinquished. Other applications can request the contents of a selection in different forms, called targets. There can be any number of selections, but most X applications only handle one, the primary selection.

In most cases, it isn't necessary for a PyGTK application to deal with selections itself. The standard widgets, such as the `Entry` (see Section 9.9) widget, already have the capability to claim the selection when appropriate (e.g., when the user drags over text), and to retrieve the contents of the selection owned by another widget or another application (e.g., when the user clicks the second mouse button). However, there may be cases in which you want to give other widgets the ability to supply the selection, or you wish to retrieve targets not supported by default.

A fundamental concept needed to understand selection handling is that of the atom. An atom is an integer that uniquely identifies a string (on a certain display). Certain atoms are predefined by the X server, GTK.

21.2 Retrieving the Selection

Retrieving the selection is an asynchronous process. To start the process, you call:

```
result = widget.selection_convert(selection, target, time=0)
```

This converts the *selection* into the form specified by *target*. *selection* is an atom corresponding to the selection type; the common selections are the strings:

```
PRIMARY

SECONDARY
```

If at all possible, the *time* field should be the time from the event that triggered the *selection*. This helps make sure that events occur in the order that the user requested them. However, if it is not available (for instance, if the conversion was triggered by a "clicked" signal), then you can use 0 which means use the current time. *result* is `TRUE` if the conversion succeeded, `FALSE` otherwise.

When the selection owner responds to the request, a "selection_received" signal is sent to your application. The handler for this signal receives a `gtk.SelectionData` object, which has the following attributes:

```
selection
target
type
format
data
```

`selection` and `target` are the values you gave in your selection_convert() method.

`type` is an atom that identifies the type of data returned by the selection owner. Some possible values are "STRING", a string of latin-1 characters, "ATOM", a series of atoms, "INTEGER", an integer, "image/x-xpixmap", etc. Most targets can only return one type.

The list of standard atoms in X and GTK+ is:

```
PRIMARY
SECONDARY
ARC
ATOM
BITMAP
CARDINAL
COLORMAP
CURSOR
CUT_BUFFER0
CUT_BUFFER1
CUT_BUFFER2
CUT_BUFFER3
CUT_BUFFER4
CUT_BUFFER5
CUT_BUFFER6
CUT_BUFFER7
DRAWABLE
FONT
INTEGER
PIXMAP
POINT
RECTANGLE
RESOURCE_MANAGER
RGB_COLOR_MAP
RGB_BEST_MAP
RGB_BLUE_MAP
RGB_DEFAULT_MAP
RGB_GRAY_MAP
RGB_GREEN_MAP
RGB_RED_MAP
STRING
VISUALID
WINDOW
WM_COMMAND
WM_HINTS
WM_CLIENT_MACHINE
WM_ICON_NAME
WM_ICON_SIZE
WM_NAME
WM_NORMAL_HINTS
WM_SIZE_HINTS
WM_ZOOM_HINTS
MIN_SPACE
NORM_SPACE
MAX_SPACE  END_SPACE,
SUPERSCRIPT_X
SUPERSCRIPT_Y
SUBSCRIPT_X
SUBSCRIPT_Y
UNDERLINE_POSITION
UNDERLINE_THICKNESS
STRIKEOUT_ASCENT
STRIKEOUT_DESCENT
ITALIC_ANGLE
X_HEIGHT
QUAD_WIDTH
WEIGHT
POINT_SIZE
```

```
RESOLUTION
COPYRIGHT
NOTICE
FONT_NAME
FAMILY_NAME
FULL_NAME
CAP_HEIGHT
WM_CLASS
WM_TRANSIENT_FOR
CLIPBOARD
```

format gives the length of the units (for instance characters) in bits. Usually, you don't care about this when receiving data.

data is the returned data in the form of a string.

PyGTK wraps all received data into a string. This makes it easy to handle string targets. To retrieve targets of other types (e.g. ATOM or INTEGER) the program must extract the information from the returned string. PyGTK provides two methods to retrieve text and a list of targets from the selection data:

```
text = selection_data.get_text()

targets = selection_data.get_targets()
```

where *text* is a string containing the text of the selection and *targets* is a list of the targets supported by the selection.

Given a `gtk.SelectionData` containing a list of targets the method:

```
has_text = selection_data.targets_include_text()
```

will return `TRUE` if one or more of the targets can provide text.

The getselection.py example program demonstrates the retrieving of a "STRING" or "TARGETS" target from the primary selection and printing the corresponding data to the console when the associated button is "clicked". Figure 21.1 illustrates the program display:

Figure 21.1 Get Selection Example

The source code for the getselection.py program is:

```
 1 #!/usr/bin/env python
 2
 3 # example getselection.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class GetSelectionExample:
10     # Signal handler invoked when user clicks on the
11     # "Get String Target" button
12     def get_stringtarget(self, widget):
13         # And request the "STRING" target for the primary selection
14         ret = widget.selection_convert("PRIMARY", "STRING")
15         return
16
17     # Signal handler invoked when user clicks on the "Get Targets" button
```

```
18     def get_targets(self, widget):
19         # And request the "TARGETS" target for the primary selection
20         ret = widget.selection_convert("PRIMARY", "TARGETS")
21         return
22 
23     # Signal handler called when the selections owner returns the data
24     def selection_received(self, widget, selection_data, data):
25         # Make sure we got the data in the expected form
26         if str(selection_data.type) == "STRING":
27             # Print out the string we received
28             print "STRING TARGET: %s" % selection_data.get_text()
29 
30         elif str(selection_data.type) == "ATOM":
31             # Print out the target list we received
32             targets = selection_data.get_targets()
33             for target in targets:
34                 name = str(target)
35                 if name != None:
36                     print "%s" % name
37                 else:
38                     print "(bad target)"
39         else:
40             print "Selection was not returned as \"STRING\" or \"ATOM\"!"
41 
42         return False
43 
44 
45     def __init__(self):
46         # Create the toplevel window
47         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
48         window.set_title("Get Selection")
49         window.set_border_width(10)
50         window.connect("destroy", lambda w: gtk.main_quit())
51 
52         vbox = gtk.VBox(False, 0)
53         window.add(vbox)
54         vbox.show()
55 
56         # Create a button the user can click to get the string target
57         button = gtk.Button("Get String Target")
58         eventbox = gtk.EventBox()
59         eventbox.add(button)
60         button.connect_object("clicked", self.get_stringtarget, eventbox)
61         eventbox.connect("selection_received", self.selection_received)
62         vbox.pack_start(eventbox)
63         eventbox.show()
64         button.show()
65 
66         # Create a button the user can click to get targets
67         button = gtk.Button("Get Targets")
68         eventbox = gtk.EventBox()
69         eventbox.add(button)
70         button.connect_object("clicked", self.get_targets, eventbox)
71         eventbox.connect("selection_received", self.selection_received)
72         vbox.pack_start(eventbox)
73         eventbox.show()
74         button.show()
75 
76         window.show()
77 
78 def main():
79     gtk.main()
80     return 0
81 
```

```
82 if __name__ == "__main__":
83     GetSelectionExample()
84     main()
```

Lines 30-38 handle the retrieval of the "TARGETS" selection data and print the list of target names. The buttons are enclosed in their own eventboxes because a selection must be associated with a `gtk.gdkWindow` and buttons are "windowless" widgets in GTK+2.0.

21.3 Supplying the Selection

Supplying the selection is a bit more complicated. You must register handlers that will be called when your selection is requested. For each selection-target pair you will handle, you make a call to:

```
widget.selection_add_target(selection, target, info)
```

widget, *selection*, and *target* identify the requests this handler will manage. When a request for a selection is received, the "selection_get" signal will be called. *info* is an integer that can be used as an enumerator to identify the specific target within the callback.

The callback has the signature:

```
def selection_get(widget, selection_data, info, time):
```

The `gtk.SelectionData` is the same as above, but this time, we're responsible for filling in the fields *type*, *format* and *data*. (The *format* field is actually important here - the X server uses it to figure out whether the *data* needs to be byte-swapped or not. Usually it will be 8 - i.e. a character - or 32 - i.e. a integer.) This is done by calling the method:

```
selection_data.set(type, format, data)
```

This PyGTK method can only handle string data so the *data* must be loaded into a Python string but *format* will be whatever the appropriate size is (e.g. 32 for atoms and integers, 8 for strings). The Python `struct` or `StringIO` modules can be used to convert non-string data to string data. For example, you can convert a list of integers to a string and set the *selection_data* by:

```
ilist = [1, 2, 3, 4, 5]

data = apply(struct.pack, ['%di'%len(ilist)] + ilist)

selection_data.set("INTEGER", 32, data)
```

The following method sets the selection data from the given string:

```
selection_data.set_text(str, len)
```

When prompted by the user, you claim ownership of the selection by calling:

```
result = widget.selection_owner_set(selection, time=0L)
```

result will be `TRUE` if program successfully claimed the *selection*. If another application claims ownership of the *selection*, you will receive a "selection_clear_event".

As an example of supplying the selection, the setselection.py program adds selection functionality to a toggle button enclosed in a `gtk.EventBox`. (The `gtk.Eventbox` is needed because the selection must be associated with a `gtk.gdk.Window` and a `gtk.Button` is a "windowless" object in GTK+ 2.0.) When the toggle button is depressed, the program claims the primary selection. The only target supported (aside from certain targets like "TARGETS" supplied by GTK+ itself), is the "STRING" target. When this target is requested, a string representation of the time is returned. Figure 21.2 illustrates the program display when the program has taken the primary selection ownership:

Figure 21.2 Set Selection Example

The setselection.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example setselection.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8 import time
 9
10 class SetSelectionExample:
11     # Callback when the user toggles the selection
12     def selection_toggled(self, widget, window):
13         if widget.get_active():
14             self.have_selection = window.selection_owner_set("PRIMARY")
15             # if claiming the selection failed, we return the button to
16             # the out state
17             if not self.have_selection:
18                 widget.set_active(False)
19         else:
20             if self.have_selection:
21                 # Not possible to release the selection in PyGTK
22                 # just mark that we don't have it
23                 self.have_selection = False
24         return
25
26     # Called when another application claims the selection
27     def selection_clear(self, widget, event):
28         self.have_selection = False
29         widget.set_active(False)
30         return True
31
32     # Supplies the current time as the selection.
33     def selection_handle(self, widget, selection_data, info, time_stamp):
34         current_time = time.time()
35         timestr = time.asctime(time.localtime(current_time))
36
37         # When we return a single string, it should not be null terminated.
38         # That will be done for us
39         selection_data.set_text(timestr, len(timestr))
40         return
41
42     def __init__(self):
43         self.have_selection = False
44         # Create the toplevel window
45         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
46         window.set_title("Set Selection")
47         window.set_border_width(10)
48         window.connect("destroy", lambda w: gtk.main_quit())
49         self.window = window
50         # Create an eventbox to hold the button since it no longer has
51         # a GdkWindow
52         eventbox = gtk.EventBox()
53         eventbox.show()
```

```
54         window.add(eventbox)
55 
56         # Create a toggle button to act as the selection
57         selection_button = gtk.ToggleButton("Claim Selection")
58         eventbox.add(selection_button)
59 
60         selection_button.connect("toggled", self.selection_toggled,  ←
   eventbox)
61         eventbox.connect_object("selection_clear_event", self. ←
   selection_clear,
62                                 selection_button)
63 
64         eventbox.selection_add_target("PRIMARY", "STRING", 1)
65         eventbox.selection_add_target("PRIMARY", "COMPOUND_TEXT", 1)
66         eventbox.connect("selection_get", self.selection_handle)
67         selection_button.show()
68         window.show()
69 
70 def main():
71     gtk.main()
72     return 0
73 
74 if __name__ == "__main__":
75     SetSelectionExample()
76     main()
```

Chapter 22

Drag-and-drop (DND)

PyGTK has a high level set of functions for doing inter-process communication via the drag-and-drop system. PyGTK can perform drag-and-drop on top of the low level Xdnd and Motif drag-and-drop protocols.

22.1 DND Overview

An application capable of drag-and-drop first defines and sets up the widget(s) for drag-and-drop. Each widget can be a source and/or destination for drag-and-drop. Note that these widgets must have an associated X Window.

Source widgets can send out drag data, thus allowing the user to drag things off of them, while destination widgets can receive drag data. Drag-and-drop destinations can limit who they accept drag data from, e.g. the same application or any application (including itself).

Sending and receiving drop data makes use of signals. Dropping an item to a destination widget requires both a data request (for the source widget) and data received signal handler (for the target widget). Additional signal handers can be connected if you want to know when a drag begins (at the very instant it starts), to when a drop is made, and when the entire drag-and-drop procedure has ended (successfully or not).

Your application will need to provide data for source widgets when requested, that involves having a drag data request signal handler. For destination widgets they will need a drop data received signal handler.

So a typical drag-and-drop cycle would look as follows:

- Drag begins. Source can get "drag-begin" signal. Can set up drag icon, etc.
- Drag moves over a drop area. Destination can get "drag-motion" signal.
- Drop occurs. Destination can get "drag-drop" signal. Destination should ask for source data.
- Drag data request (when a drop occurs). Source can get "drag-data-get" signal.
- Drop data received (may be on same or different application). Destination can get "drag-data-received" signal.
- Drag data delete (if the drag was a move). Source can get "drag-data-delete" signal
- Drag-and-drop procedure done. Source can receive "drag-end" signal.

There are a few minor steps that go in between here and there, but we will get into detail about that later.

22.2 DND Properties

Drag data has the following properties:

- Drag action type (ie `ACTION_COPY`, `ACTION_MOVE`).

- Client specified arbitrary drag-and-drop type (a name and number pair).
- Sent and received data format type.

Drag actions are quite obvious, they specify if the widget can drag with the specified action(s), e.g. `gtk.gdk.ACTION_COPY` and/or `gtk.gdk.ACTION_MOVE`. An `gtk.gdk.ACTION_COPY` would be a typical drag-and-drop without the source data being deleted while `gtk.gdk.ACTION_MOVE` would be just like `gtk.gdk.ACTION_COPY` but the source data will be 'suggested' to be deleted after the received signal handler is called. There are additional drag actions including `gtk.gdk.ACTION_LINK` which you may want to look into when you get to more advanced levels of drag-and-drop.

The client specified arbitrary drag-and-drop type is much more flexible, because your application will be defining and checking for that specifically. You will need to set up your destination widgets to receive certain drag-and-drop types by specifying a name and/or number. It would be more reliable to use a name since another application may just happen to use the same number for an entirely different meaning.

Sent and received data format types (*selection target*) come into play only in your request and received data handler functions. The term *selection target* is somewhat misleading. It is a term adapted from GTK+ selection (cut/copy and paste). What *selection target* actually means is the data's format type (i.e. `gtk.gdk.Atom`, integer, or string) that is being sent or received. Your request data handler function needs to specify the type (*selection target*) of data that it sends out and your received data handler needs to handle the type (*selection target*) of data received.

22.3 DND Methods

22.3.1 Setting Up the Source Widget

The method drag_source_set() specifies a set of target types for a drag operation on a widget.

```
widget.drag_source_set(start_button_mask, targets, actions)
```

The parameters signify the following:

- *`widget`* specifies the drag source widget
- *`start_button_mask`* specifies a bitmask of buttons that can start the drag (e.g. `BUTTON1_MASK`)
- *`targets`* specifies a list of target data types the drag will support
- *`actions`* specifies a bitmask of possible actions for a drag from this window

The *`targets`* parameter is a list of tuples each similar to:

```
(target, flags, info)
```

`target` specifies a string representing the drag type.

`flags` restrict the drag scope. *`flags`* can be set to 0 (no limitation of scope) or the following flags:

```
gtk.TARGET_SAME_APP    # Target will only be selected for drags within a single ↩
    application.

gtk.TARGET_SAME_WIDGET # Target will only be selected for drags within a single ↩
    widget.
```

`info` is an application assigned integer identifier.

If a widget is no longer required to act as a source for drag-and-drop operations, the method drag_source_unset() can be used to remove a set of drag-and-drop target types.

```
widget.drag_source_unset()
```

Table 22.1 Source Widget Signals

drag_begin	def drag_begin_cb(widget, drag_context, data):
drag_data_get	def drag_data_get_cb(widget, drag_context, selection_data, info, time, data):
drag_data_delete	def drag_data_delete_cb(widget, drag_context, data):
drag_end	def drag_end_cb(widget, drag_context, data):

22.3.2 Signals On the Source Widget

The source widget is sent the following signals during a drag-and-drop operation.

The "drag-begin" signal handler can be used to set up some inital conditions such as a drag icon using one of the `Widget` methods: drag_source_set_icon(), drag_source_set_icon_pixbuf(), drag_source_set_icon_stock(). The "drag-end' signal handler can be used to undo the actions of the "drag-begin" signal ahndler.

The "drag-data-get" signal handler should return the drag data matching the target specified by *info*. It fills in the `gtk.gdk.SelectionData` with the drag data.

The "drag-delete" signal handler is used to delete the drag data for a `gtk.gdk.ACTION_MOVE` action after the data has been copied.

22.3.3 Setting Up a Destination Widget

The drag_dest_set() method specifies that this widget can receive drops and specifies what types of drops it can receive.

drag_dest_unset() specifies that the widget can no longer receive drops.

```
widget.drag_dest_set(flags, targets, actions)

widget.drag_dest_unset()
```

flags specifies what actions GTK+ should take on behalf of widget for drops on it. The possible values of *flags* are:

gtk.DEST_DEFAULT_MOTION If set for a widget, GTK+, during a drag over this widget will check if the drag matches this widget's list of possible targets and actions. GTK+ will then call drag_status() as appropriate.

gtk.DEST_DEFAULT_HIGHLIGHT If set for a widget, GTK+ will draw a highlight on this widget as long as a drag is over this widget and the widget drag format and action is acceptable.

gtk.DEST_DEFAULT_DROP If set for a widget, when a drop occurs, GTK+ will check if the drag matches this widget's list of possible targets and actions. If so, GTK+ will call drag_get_data() on behalf of the widget. Whether or not the drop is succesful, GTK+ will call drag_finish(). If the action was a move and the drag was succesful, then `TRUE` will be passed for the *delete* parameter to drag_finish().

gtk.DEST_DEFAULT_ALL If set, specifies that all default actions should be taken.

targets is a list of target information tuples as described above.

actions is a bitmask of possible actions for a drag onto this widget. The possible values that can be or'd for actions are:

```
gtk.gdk.ACTION_DEFAULT
gtk.gdk.ACTION_COPY
gtk.gdk.ACTION_MOVE
gtk.gdk.ACTION_LINK
gtk.gdk.ACTION_PRIVATE
gtk.gdk.ACTION_ASK
```

targets and *actions* are ignored if *flags* does not contain `gtk.DEST_DEFAULT_MOTION` or `gtk.DEST_DEFAULT_DROP`. In that case the application must handle the "drag-motion" and "drag-drop" signals.

The "drag-motion" handler must determine if the drag data is appropriate by matching the destination targets with the `gtk.gdk.DragContext` targets and optionally by examining the drag data

by calling the drag_get_data() method. The `gtk.gdk.DragContext`. drag_status() method must be called to update the *drag_context* status.

The "drag-drop" handler must determine the matching target using the Widget drag_dest_find_target() method and then ask for the drag data using the Widget drag_get_data() method. The data will be available in the "drag-data-received" handler.

The dragtargets.py program prints out the targets of a drag operation in a label:

```
 1  #!/usr/local/env python
 2
 3  import pygtk
 4  pygtk.require('2.0')
 5  import gtk
 6
 7  def motion_cb(wid, context, x, y, time):
 8      context.drag_status(gtk.gdk.ACTION_COPY, time)
 9      return True
10
11  def drop_cb(wid, context, x, y, time):
12      l.set_text('\n'.join([str(t) for t in context.targets]))
13      context.finish(True, False, time)
14      return True
15
16  w = gtk.Window()
17  w.set_size_request(200, 150)
18  w.drag_dest_set(0, [], 0)
19  w.connect('drag_motion', motion_cb)
20  w.connect('drag_drop', drop_cb)
21  w.connect('destroy', lambda w: gtk.main_quit())
22  l = gtk.Label()
23  w.add(l)
24  w.show_all()
25
26  gtk.main()
```

The program creates a window and then sets it as a drag destination for no targets and actions by setting the flags to zero. The `motion_cb`() and `drop_cb`() handlers are connected to the "drag-motion" and "drag-drop" signals respectively. The `motion_cb`() handler just sets the drag status for the drag context so that a drop will be enabled. The `drop_cb`() sets the label text to a string containing the drag targets and finishes the drop leaving the source data intact.

22.3.4 Signals On the Destination Widget

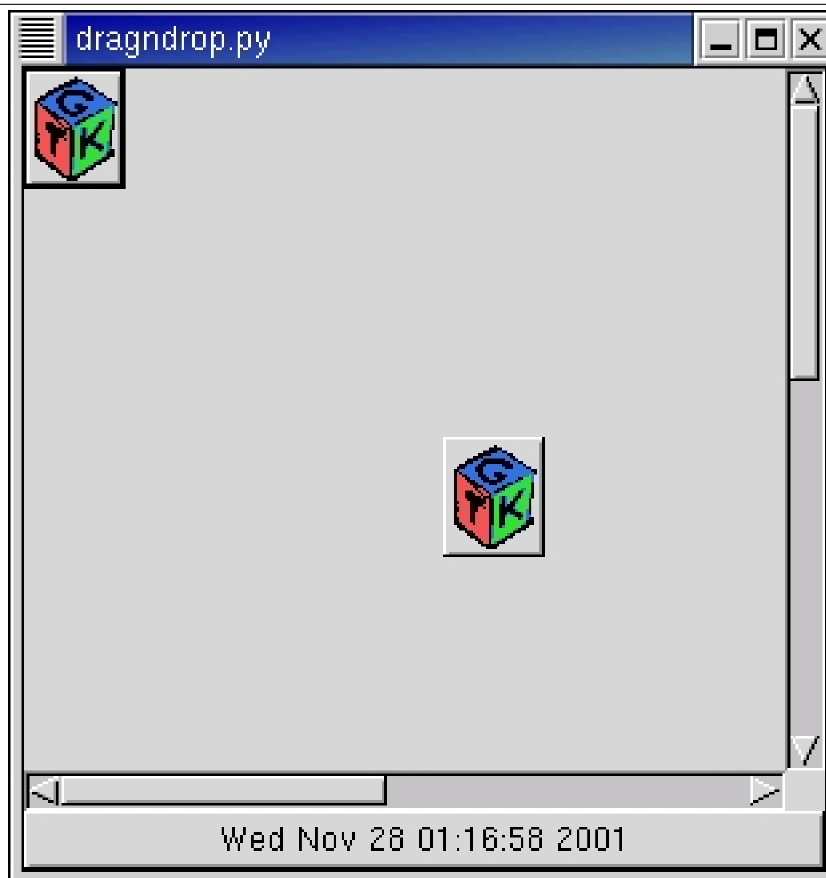
The destination widget is sent the following signals during a drag-and-drop operation.

Table 22.2 Destination Widget Signals

drag_motion	def drag_motion_cb(widget, drag_context, x, y, time, data):
drag_drop	def drag_drop_cb(widget, drag_context, x, y, time, data):
drag_data_received	def drag_data_received_cb(widget, drag_context, x, y, selection_data, info, time, data):

The dragndrop.py example program demonstrates the use of drag and drop in one application. A button with a xpm pixmap (in gtkxpm.py) is the source for the drag; it provides both text and xpm data. A layout widget is the destination for the xpm drop while a button is the destination for the text drop. Figure 22.1 illustrates the program display after an xpm drop has been made on the layout and a text drop has been made on the button:

Figure 22.1 Drag and Drop Example



The dragndrop.py source code is:

```
 1 #!/usr/bin/env python
 2
 3 # example dragndrop.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8 import string, time
 9
10 import gtkxpm
11
12 class DragNDropExample:
13     HEIGHT = 600
14     WIDTH = 600
15     TARGET_TYPE_TEXT = 80
16     TARGET_TYPE_PIXMAP = 81
17     fromImage = [ ( "text/plain", 0, TARGET_TYPE_TEXT ),
18                   ( "image/x-xpixmap", 0, TARGET_TYPE_PIXMAP ) ]
19     toButton = [ ( "text/plain", 0, TARGET_TYPE_TEXT ) ]
20     toCanvas = [ ( "image/x-xpixmap", 0, TARGET_TYPE_PIXMAP ) ]
21
22     def layout_resize(self, widget, event):
23         x, y, width, height = widget.get_allocation()
24         if width > self.lwidth or height > self.lheight:
25             self.lwidth = max(width, self.lwidth)
26             self.lheight = max(height, self.lheight)
27             widget.set_size(self.lwidth, self.lheight)
28
29     def makeLayout(self):
```

```
30         self.lwidth = self.WIDTH
31         self.lheight = self.HEIGHT
32         box = gtk.VBox(False,0)
33         box.show()
34         table = gtk.Table(2, 2, False)
35         table.show()
36         box.pack_start(table, True, True, 0)
37         layout = gtk.Layout()
38         self.layout = layout
39         layout.set_size(self.lwidth, self.lheight)
40         layout.connect("size-allocate", self.layout_resize)
41         layout.show()
42         table.attach(layout, 0, 1, 0, 1, gtk.FILL|gtk.EXPAND,
43                      gtk.FILL|gtk.EXPAND, 0, 0)
44         # create the scrollbars and pack into the table
45         vScrollbar = gtk.VScrollbar(None)
46         vScrollbar.show()
47         table.attach(vScrollbar, 1, 2, 0, 1, gtk.FILL|gtk.SHRINK,
48                      gtk.FILL|gtk.SHRINK, 0, 0)
49         hScrollbar = gtk.HScrollbar(None)
50         hScrollbar.show()
51         table.attach(hScrollbar, 0, 1, 1, 2, gtk.FILL|gtk.SHRINK,
52                      gtk.FILL|gtk.SHRINK,
53                      0, 0)
54         # tell the scrollbars to use the layout widget's adjustments
55         vAdjust = layout.get_vadjustment()
56         vScrollbar.set_adjustment(vAdjust)
57         hAdjust = layout.get_hadjustment()
58         hScrollbar.set_adjustment(hAdjust)
59         layout.connect("drag_data_received", self.receiveCallback)
60         layout.drag_dest_set(gtk.DEST_DEFAULT_MOTION |
61                                   gtk.DEST_DEFAULT_HIGHLIGHT |
62                                   gtk.DEST_DEFAULT_DROP,
63                                   self.toCanvas, gtk.gdk.ACTION_COPY)
64         self.addImage(gtkxpm.gtk_xpm, 0, 0)
65         button = gtk.Button("Text Target")
66         button.show()
67         button.connect("drag_data_received", self.receiveCallback)
68         button.drag_dest_set(gtk.DEST_DEFAULT_MOTION |
69                              gtk.DEST_DEFAULT_HIGHLIGHT |
70                              gtk.DEST_DEFAULT_DROP,
71                              self.toButton, gtk.gdk.ACTION_COPY)
72         box.pack_start(button, False, False, 0)
73         return box
74 
75     def addImage(self, xpm, xd, yd):
76         hadj = self.layout.get_hadjustment()
77         vadj = self.layout.get_vadjustment()
78         style = self.window.get_style()
79         pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(
80             self.window.window, style.bg[gtk.STATE_NORMAL], xpm)
81         image = gtk.Image()
82         image.set_from_pixmap(pixmap, mask)
83         button = gtk.Button()
84         button.add(image)
85         button.connect("drag_data_get", self.sendCallback)
86         button.drag_source_set(gtk.gdk.BUTTON1_MASK, self.fromImage,
87                                gtk.gdk.ACTION_COPY)
88         button.show_all()
89         # have to adjust for the scrolling of the layout - event location
90         # is relative to the viewable not the layout size
91         self.layout.put(button, int(xd+hadj.value), int(yd+vadj.value))
92         return
93 
```

```
 94     def sendCallback(self, widget, context, selection, targetType,  ↩
    eventTime):
 95         if targetType == self.TARGET_TYPE_TEXT:
 96             now = time.time()
 97             str = time.ctime(now)
 98             selection.set(selection.target, 8, str)
 99         elif targetType == self.TARGET_TYPE_PIXMAP:
100             selection.set(selection.target, 8,
101                           string.join(gtkxpm.gtk_xpm, '\n'))
102 
103     def receiveCallback(self, widget, context, x, y, selection, targetType,
104                         time):
105         if targetType == self.TARGET_TYPE_TEXT:
106             label = widget.get_children()[0]
107             label.set_text(selection.data)
108         elif targetType == self.TARGET_TYPE_PIXMAP:
109             self.addImage(string.split(selection.data, '\n'), x, y)
110 
111     def __init__(self):
112         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
113         self.window.set_default_size(300, 300)
114         self.window.connect("destroy", lambda w: gtk.main_quit())
115         self.window.show()
116         layout = self.makeLayout()
117         self.window.add(layout)
118 
119 def main():
120     gtk.main()
121 
122 if __name__ == "__main__":
123     DragNDropExample()
124     main()
```

Chapter 23

GTK's rc Files

GTK+ has its own way of dealing with application defaults, by using rc files. These can be used to set the colors of just about any widget, and can also be used to tile pixmaps onto the background of some widgets.

23.1 Functions For rc Files

When your application starts, you should include a call to:

```
rc_parse(filename)
```

Passing in the *filename* of your rc file. This will cause GTK+ to parse this file, and use the style settings for the widget types defined there.

If you wish to have a special set of widgets that can take on a different style from others, or any other logical division of widgets, use a call to:

```
widget.set_name(name)
```

Your newly created *widget* will be assigned the *name* you give. This will allow you to change the attributes of this *widget* by name through the rc file.

If we use a call something like this:

```
button = gtk.Button("Special Button")

button.set_name("special button")
```

Then this *button* is given the name "special button" and may be addressed by name in the rc file as "special button.GtkButton". [--- Verify ME!]

Section 23.3 below, sets the properties of the main window, and lets all children of that main window inherit the style described by the "main button" style. The code used in the application is:

```
window = gtk.Window(gtk.WINDOW_TOPLEVEL)

window.set_name("main window")
```

And then the style is defined in the rc file using:

```
widget "main window.*GtkButton*" style "main_button"
```

Which sets all the `Button` (see Chapter 6 widgets in the "main window" to the "main_buttons" style as defined in the rc file.

As you can see, this is a fairly powerful and flexible system. Use your imagination as to how best to take advantage of this.

23.2 GTK's rc File Format

The format of the GTK+ rc file is illustrated in Section 23.3 below. This is the `testgtkrc` file from the GTK+ distribution, but I've added a few comments and things. You may wish to include this explanation in your application to allow the user to fine tune his application.

There are several directives to change the attributes of a widget.

- *fg* - Sets the foreground color of a widget.
- *bg* - Sets the background color of a widget.
- *bg_pixmap* - Sets the background of a widget to a tiled pixmap.
- *font* - Sets the font to be used with the given widget.

In addition to this, there are several states a widget can be in, and you can set different colors, pixmaps and fonts for each state. These states are:

NORMAL The normal state of a widget, without the mouse over top of it, and not being pressed, etc.

PRELIGHT When the mouse is over top of the widget, colors defined using this state will be in effect.

ACTIVE When the widget is pressed or clicked it will be active, and the attributes assigned by this tag will be in effect.

INSENSITIVE When a widget is set insensitive, and cannot be activated, it will take these attributes.

SELECTED When an object is selected, it takes these attributes.

When using the "fg" and "bg" keywords to set the colors of widgets, the format is:

```
fg[<STATE>] = { Red, Green, Blue }
```

Where `STATE` is one of the above states (PRELIGHT, ACTIVE, etc), and the `Red`, `Green` and `Blue` are values in the range of 0 - 1.0, { 1.0, 1.0, 1.0 } being white. They must be in float form, or they will register as 0, so a straight "1" will not work, it must be "1.0". A straight "0" is fine because it doesn't matter if it's not recognized. Unrecognized values are set to 0.

bg_pixmap is very similar to the above, except the colors are replaced by a filename.

pixmap_path is a list of paths separated by ":"'s. These paths will be searched for any pixmap you specify.

The "font" directive is simply:

```
font = "<font name>"
```

The only hard part is figuring out the *font* string. Using **xfontsel** or a similar utility should help.

The "widget_class" sets the style of a class of widgets. These classes are listed in the widget overview in Section 5.1.

The "widget" directive sets a specifically named set of widgets to a given style, overriding any style set for the given widget class. These widgets are registered inside the application using the set_name() method. This allows you to specify the attributes of a widget on a per widget basis, rather than setting the attributes of an entire widget class. I urge you to document any of these special widgets so users may customize them.

When the keyword *parent* is used as an attribute, the widget will take on the attributes of its parent in the application.

When defining a style, you may assign the attributes of a previously defined style to this new one.

```
style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

This example takes the "button" style, and creates a new "main_button" style simply by changing the font and prelight background color of the "button" style.

Of course, many of these attributes don't apply to all widgets. It's a simple matter of common sense really. Anything that could apply, should.

23.3 Example rc file

```
# pixmap_path "<dir 1>:<dir 2>:<dir 3>:..."
#
pixmap_path "/usr/include/X11R6/pixmaps:/home/imain/pixmaps"
#
# style <name> [= <name>]
# {
#   <option>
# }
#
# widget <widget_set> style <style_name>
# widget_class <widget_class_set> style <style_name>

# Here is a list of all the possible states.  Note that some do not apply to
# certain widgets.
#
# NORMAL - The normal state of a widget, without the mouse over top of
# it, and not being pressed, etc.
#
# PRELIGHT - When the mouse is over top of the widget, colors defined
# using this state will be in effect.
#
# ACTIVE - When the widget is pressed or clicked it will be active, and
# the attributes assigned by this tag will be in effect.
#
# INSENSITIVE - When a widget is set insensitive, and cannot be
# activated, it will take these attributes.
#
# SELECTED - When an object is selected, it takes these attributes.
#
# Given these states, we can set the attributes of the widgets in each of
# these states using the following directives.
#
# fg - Sets the foreground color of a widget.
# fg - Sets the background color of a widget.
# bg_pixmap - Sets the background of a widget to a tiled pixmap.
# font - Sets the font to be used with the given widget.
#

# This sets a style called "button".  The name is not really important, as
# it is assigned to the actual widgets at the bottom of the file.

style "window"
{
  #This sets the padding around the window to the pixmap specified.
  #bg_pixmap[<STATE>] = "<pixmap filename>"
  bg_pixmap[NORMAL] = "warning.xpm"
}

style "scale"
{
  #Sets the foreground color (font color) to red when in the "NORMAL"
  #state.

  fg[NORMAL] = { 1.0, 0, 0 }

  #Sets the background pixmap of this widget to that of its parent.
  bg_pixmap[NORMAL] = "<parent>"
}

style "button"
{
```

```
  # This shows all the possible states for a button.  The only one that
  # doesn't apply is the SELECTED state.

  fg[PRELIGHT] = { 0, 1.0, 1.0 }
  bg[PRELIGHT] = { 0, 0, 1.0 }
  bg[ACTIVE] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 0, 1.0, 0 }
  bg[NORMAL] = { 1.0, 1.0, 0 }
  fg[NORMAL] = { .99, 0, .99 }
  bg[INSENSITIVE] = { 1.0, 1.0, 1.0 }
  fg[INSENSITIVE] = { 1.0, 0, 1.0 }
}

# In this example, we inherit the attributes of the "button" style and then
# override the font and background color when prelit to create a new
# "main_button" style.

style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}

style "toggle_button" = "button"
{
  fg[NORMAL] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 1.0, 0, 0 }

  # This sets the background pixmap of the toggle_button to that of its
  # parent widget (as defined in the application).
  bg_pixmap[NORMAL] = "<parent>"
}

style "text"
{
  bg_pixmap[NORMAL] = "marble.xpm"
  fg[NORMAL] = { 1.0, 1.0, 1.0 }
}

style "ruler"
{
  font = "-adobe-helvetica-medium-r-normal--*-80-*-*-*-*-*-*"
}

# pixmap_path "~/.pixmaps"

# These set the widget types to use the styles defined above.
# The widget types are listed in the class hierarchy, but could probably be
# just listed in this document for the users reference.

widget_class "GtkWindow" style "window"
widget_class "GtkDialog" style "window"
widget_class "GtkFileSelection" style "window"
widget_class "*Gtk*Scale" style "scale"
widget_class "*GtkCheckButton*" style "toggle_button"
widget_class "*GtkRadioButton*" style "toggle_button"
widget_class "*GtkButton*" style "button"
widget_class "*Ruler" style "ruler"
widget_class "*GtkText" style "text"

# This sets all the buttons that are children of the "main window" to
# the main_button style.  These must be documented to be taken advantage of.
widget "main window.*GtkButton*" style "main_button"
```

Chapter 24

Scribble, A Simple Example Drawing Program

24.1 Scribble Overview

In this section, we will build a simple drawing program. In the process, we will examine how to handle mouse events, how to draw in a window, and how to do drawing better by using a backing pixmap.

Figure 24.1 Scribble Drawing Program Example

24.2 Event Handling

The GTK+ signals we have already discussed are for high-level actions, such as a menu item being selected. However, sometimes it is useful to learn about lower-level occurrences, such as the mouse being moved, or a key being pressed. There are also GTK+ signals corresponding to these low-level events. The handlers for these signals have an extra parameter which is a `gtk.gdk.Event` object containing information about the event. For instance, motion event handlers are passed a `gtk.gdk.Event` object containing EventMotion information which has (in part) attributes like:

```
type
window
time
```

```
  x
  y
    ...
  state
    ...
```

window is the window in which the event occurred.
x and *y* give the coordinates of the event.
type will be set to the event type, in this case MOTION_NOTIFY. The types (in module gtk.gdk) are:

```
NOTHING                   a special code to indicate a null event.

DELETE                    the window manager has requested that the toplevel window  ↩
    be
                          hidden or destroyed, usually when the user clicks on a  ↩
                              special
                          icon in the title bar.

DESTROY                   the window has been destroyed.

EXPOSE                    all or part of the window has become visible and needs to  ↩
    be
                          redrawn.

MOTION_NOTIFY             the pointer (usually a mouse) has moved.

BUTTON_PRESS              a mouse button has been pressed.

_2BUTTON_PRESS            a mouse button has been double-clicked (clicked twice  ↩
    within
                          a short period of time). Note that each click also  ↩
                              generates a
                          BUTTON_PRESS event.

_3BUTTON_PRESS            a mouse button has been clicked 3 times in a short period  ↩
    of
                          time. Note that each click also generates a BUTTON_PRESS  ↩
                              event.

BUTTON_RELEASE            a mouse button has been released.

KEY_PRESS                 a key has been pressed.

KEY_RELEASE               a key has been released.

ENTER_NOTIFY              the pointer has entered the window.

LEAVE_NOTIFY              the pointer has left the window.

FOCUS_CHANGE              the keyboard focus has entered or left the window.

CONFIGURE                 the size, position or stacking order of the window has  ↩
    changed.
                          Note that GTK+ discards these events for GDK_WINDOW_CHILD  ↩
                              windows.

MAP                       the window has been mapped.

UNMAP                     the window has been unmapped.

PROPERTY_NOTIFY           a property on the window has been changed or deleted.

SELECTION_CLEAR           the application has lost ownership of a selection.
```

```
SELECTION_REQUEST      another application has requested a selection.

SELECTION_NOTIFY       a selection has been received.

PROXIMITY_IN           an input device has moved into contact with a sensing  ←
    surface
                       (e.g. a touchscreen or graphics tablet).

PROXIMITY_OUT          an input device has moved out of contact with a sensing  ←
    surface.

DRAG_ENTER             the mouse has entered the window while a drag is in  ←
    progress.

DRAG_LEAVE             the mouse has left the window while a drag is in progress.

DRAG_MOTION            the mouse has moved in the window while a drag is in  ←
    progress.

DRAG_STATUS            the status of the drag operation initiated by the window  ←
    has changed.

DROP_START             a drop operation onto the window has started.

DROP_FINISHED          the drop operation initiated by the window has completed.

CLIENT_EVENT           a message has been received from another application.

VISIBILITY_NOTIFY      the window visibility status has changed.

NO_EXPOSE              indicates that the source region was completely available  ←
    when parts
                       of a drawable were copied. This is not very useful.

SCROLL                 ?

WINDOW_STATE           ?

SETTING                ?
```

state specifies the modifier state when the *event* occurred (that is, it specifies which modifier keys and mouse buttons were pressed). It is the bitwise OR of some of the following (in module gtk.gdk):

```
SHIFT_MASK
LOCK_MASK
CONTROL_MASK
MOD1_MASK
MOD2_MASK
MOD3_MASK
MOD4_MASK
MOD5_MASK
BUTTON1_MASK
BUTTON2_MASK
BUTTON3_MASK
BUTTON4_MASK
BUTTON5_MASK
```

As for other signals, to determine what happens when an event occurs we call the connect() method. But we also need to let GTK+ know which events we want to be notified about. To do this, we call the method:

```
widget.set_events(events)
```

The *events* argument specifies the events we are interested in. It is the bitwise OR of constants that specify different types of events. For future reference, the event types (in module gtk.gdk) are:

```
EXPOSURE_MASK
POINTER_MOTION_MASK
POINTER_MOTION_HINT_MASK
BUTTON_MOTION_MASK
BUTTON1_MOTION_MASK
BUTTON2_MOTION_MASK
BUTTON3_MOTION_MASK
BUTTON_PRESS_MASK
BUTTON_RELEASE_MASK
KEY_PRESS_MASK
KEY_RELEASE_MASK
ENTER_NOTIFY_MASK
LEAVE_NOTIFY_MASK
FOCUS_CHANGE_MASK
STRUCTURE_MASK
PROPERTY_CHANGE_MASK
VISIBILITY_NOTIFY_MASK
PROXIMITY_IN_MASK
PROXIMITY_OUT_MASK
SUBSTRUCTURE_MASK
```

There are a few subtle points that have to be observed when calling the set_events() method. First, it must be called before the X window for a PyGTK widget is created. In practical terms, this means you should call it immediately after creating the widget. Second, the widget must be one which will be realized with an associated X window. For efficiency, many widget types do not have their own window, but draw in their parent's window. These widgets include:

```
gtk.Alignment
gtk.Arrow
gtk.Bin
gtk.Box
gtk.Image
gtk.Item
gtk.Label
gtk.Layout
gtk.Pixmap
gtk.ScrolledWindow
gtk.Separator
gtk.Table
gtk.AspectFrame
gtk.Frame
gtk.VBox
gtk.HBox
gtk.VSeparator
gtk.HSeparator
```

To capture events for these widgets, you need to use an `EventBox` widget. See Section 10.1 widget for details.

The event attributes that are set by PyGTK for each type of event are:

```
every event              type
                         window
                         send_event

NOTHING
DELETE
DESTROY                                      # no additional attributes

EXPOSE                  area
                        count

MOTION_NOTIFY           time
                        x
                        y
```

```
                       pressure
                       xtilt
                       ytilt
                       state
                       is_hint
                       source
                       deviceid
                       x_root
                       y_root

BUTTON_PRESS
_2BUTTON_PRESS
_3BUTTON_PRESS
BUTTON_RELEASE         time
                       x
                       y
                       pressure
                       xtilt
                       ytilt
                       state
                       button
                       source
                       deviceid
                       x_root
                       y_root

KEY_PRESS
KEY_RELEASE            time
                       state
                       keyval
                       string

ENTER_NOTIFY
LEAVE_NOTIFY           subwindow
                       time
                       x
                       y
                       x_root
                       y_root
                       mode
                       detail
                       focus
                       state

FOCUS_CHANGE           _in

CONFIGURE              x
                       y
                       width
                       height

MAP
UNMAP                                  # no additional attributes

PROPERTY_NOTIFY        atom
                       time
                       state

SELECTION_CLEAR
SELECTION_REQUEST
SELECTION_NOTIFY       selection
                       target
                       property
                       requestor
```

```
                       time

PROXIMITY_IN
PROXIMITY_OUT          time
                       source
                       deviceid

DRAG_ENTER
DRAG_LEAVE
DRAG_MOTION
DRAG_STATUS
DROP_START
DROP_FINISHED          context
                       time
                       x_root
                       y_root

CLIENT_EVENT           message_type
                       data_format
                       data

VISIBILTY_NOTIFY       state

NO_EXPOSE                             # no additional attributes
```

24.2.1 Scribble - Event Handling

For our drawing program, we want to know when the mouse button is pressed and when the mouse is moved, so we specify POINTER_MOTION_MASK and BUTTON_PRESS_MASK. We also want to know when we need to redraw our window, so we specify EXPOSURE_MASK. Although we want to be notified via a Configure event when our window size changes, we don't have to specify the corresponding ST-RUCTURE_MASK flag, because it is automatically specified for all windows.

It turns out, however, that there is a problem with just specifying POINTER_MOTION_MASK. This will cause the server to add a new motion event to the event queue every time the user moves the mouse. Imagine that it takes us 0.1 seconds to handle a motion event, but the X server queues a new motion event every 0.05 seconds. We will soon get way behind the users drawing. If the user draws for 5 seconds, it will take us another 5 seconds to catch up after they release the mouse button! What we would like is to only get one motion event for each event we process. The way to do this is to specify POINTER_MOTION_HINT_MASK.

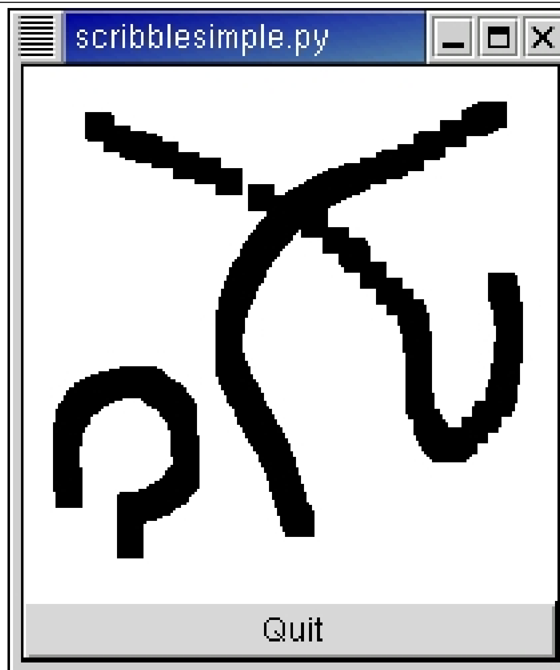
When we specify POINTER_MOTION_HINT_MASK, the server sends us a motion event the first time the pointer moves after entering our window, or after a button press or release event. Subsequent motion events will be suppressed until we explicitly ask for the position of the pointer using the gtk.gdk.W-indow method:

```
  x, y, mask = window.get_pointer()
```

window is a gtk.gdk.Window object. *x* and *y* are the coordinates of the pointer and *mask* is the modifier mask to detect which keys are pressed. (There is a gtk.Widget method, get_pointer() which provides the same information as the gtk.gdk.Window get_pointer() method but it does not return the mask information)

The scribblesimple.py example program demonstrates the basic use of events and event handlers. Figure 24.2 illustrates the program in action:

Figure 24.2 Simple Scribble Example



The event handlers are connected to the drawing_area by the following lines:

```
 92         # Signals used to handle backing pixmap
 93         drawing_area.connect("expose_event", expose_event)
 94         drawing_area.connect("configure_event", configure_event)
 95
 96         # Event signals
 97         drawing_area.connect("motion_notify_event", motion_notify_event)
 98         drawing_area.connect("button_press_event", button_press_event)
 99
100         drawing_area.set_events(gtk.gdk.EXPOSURE_MASK
101                                 | gtk.gdk.LEAVE_NOTIFY_MASK
102                                 | gtk.gdk.BUTTON_PRESS_MASK
103                                 | gtk.gdk.POINTER_MOTION_MASK
104                                 | gtk.gdk.POINTER_MOTION_HINT_MASK)
```

The `button_press_event()` and `motion_notify_event()` event handlers in scribblesimple.py are:

```
 57     def button_press_event(widget, event):
 58         if event.button == 1 and pixmap != None:
 59             draw_brush(widget, event.x, event.y)
 60         return True
 61
 62     def motion_notify_event(widget, event):
 63         if event.is_hint:
 64             x, y, state = event.window.get_pointer()
 65         else:
 66             x = event.x
 67             y = event.y
 68             state = event.state
 69
 70         if state & gtk.gdk.BUTTON1_MASK and pixmap != None:
 71             draw_brush(widget, x, y)
 72
 73         return True
```

The `expose_event()` and `configure_event()` handlers will be described later.

24.3 The DrawingArea Widget, And Drawing

We now turn to the process of drawing on the screen. The widget we use for this is the `DrawingArea` (see Chapter 12) widget. A drawing area widget is essentially an X window and nothing more. It is a blank canvas in which we can draw whatever we like. A drawing area is created using the call:

```
darea = gtk.DrawingArea()
```

A default size for the widget can be specified by calling:

```
darea.set_size_request(width, height)
```

This default size can be overridden, as is true for all widgets, by calling the set_size_request() method, and that, in turn, can be overridden if the user manually resizes the the window containing the drawing area.

It should be noted that when we create a `DrawingArea` widget, we are completely responsible for drawing the contents. If our window is obscured then uncovered, we get an exposure event and must redraw what was previously hidden.

Having to remember everything that was drawn on the screen so we can properly redraw it can, to say the least, be a nuisance. In addition, it can be visually distracting if portions of the window are cleared, then redrawn step by step. The solution to this problem is to use an offscreen backing pixmap. Instead of drawing directly to the screen, we draw to an image stored in server memory but not displayed, then when the image changes or new portions of the image are displayed, we copy the relevant portions onto the screen.

To create an offscreen pixmap, we call the function:

```
pixmap = gtk.gdk.Pixmap(window, width, height, depth=-1)
```

The *window* parameter specifies a `gtk.gdk.Window` that this *pixmap* takes some of its properties from. *width* and *height* specify the size of the *pixmap*. *depth* specifies the color depth, that is the number of bits per pixel, for the new window. If the *depth* is specified as -1 or omitted, it will match the depth of window.

We create the pixmap in our "configure_event" handler. This event is generated whenever the window changes size, including when it is originally created.

```
32  # Create a new backing pixmap of the appropriate size
33  def configure_event(widget, event):
34      global pixmap
35
36      x, y, width, height = widget.get_allocation()
37      pixmap = gtk.gdk.Pixmap(widget.window, width, height)
38      pixmap.draw_rectangle(widget.get_style().white_gc,
39                            True, 0, 0, width, height)
40
41      return True
```

The call to draw_rectangle() clears the pixmap initially to white. We'll say more about that in a moment.

Our exposure event handler then simply copies the relevant portion of the pixmap onto the drawing area (widget) using the draw_pixmap() method. (We determine the area we need to redraw by using the *event.area* attribute of the exposure event):

```
43  # Redraw the screen from the backing pixmap
44  def expose_event(widget, event):
45      x , y, width, height = event.area
46      widget.window.draw_drawable(widget.get_style().fg_gc[gtk.STATE_NORMAL ↩
    ],
47                                  pixmap, x, y, x, y, width, height)
48      return False
```

We've now seen how to keep the screen up to date with our pixmap, but how do we actually draw interesting stuff on our pixmap? There are a large number of calls in PyGTK for drawing on drawables. A drawable is simply something that can be drawn upon. It can be a window, a pixmap, or a bitmap (a black and white image). We've already seen two such calls above, draw_rectangle() and draw_pixmap(). The complete list is:

```
drawable.draw_point(gc, x, y)

drawable.draw_line(gc, x1, y1, x2, y2)

drawable.draw_rectangle(gc, fill, x, y, width, height)

drawable.draw_arc(gc, fill, x, y, width, height, angle1, angle2)

drawable.draw_polygon(gc, fill, points)

drawable.draw_drawable(gc, src, xsrc, ysrc, xdest, ydest, width, height)

drawable.draw_points(gc, points)

drawable.draw_lines(gc, points)

drawable.draw_segments(gc, segments)

drawable.draw_rgb_image(gc, x, y, width, height, dither, buffer, rowstride)

drawable.draw_rgb_32_image(gc, x, y, width, height, dither, buffer, rowstride)

drawable.draw_gray_image(gc, x, y, width, height, dither, buffer, rowstride)
```

The drawing area methods are the same as the drawable drawing methods so you can use the methods described in Section 12.2 for further details on these methods. These methods all share the same first arguments. The first argument is a graphics context (*gc*).

A graphics context encapsulates information about things such as foreground and background color and line width. PyGTK has a full set of functions for creating and modifying graphics contexts, but to keep things simple we'll just use predefined graphics contexts. See Section 12.1 section for more information on graphics contexts. Each widget has an associated style. (Which can be modified in a `gtkrc` file, see Chapter 23.) This, among other things, stores a number of graphics contexts. Some examples of accessing these graphics contexts are:

```
widget.get_style().white_gc

widget.get_style().black_gc

widget.get_style().fg_gc[STATE_NORMAL]

widget.get_style().bg_gc[STATE_PRELIGHT]
```

The fields *fg_gc*, *bg_gc*, *dark_gc*, and *light_gc* are indexed by a parameter which can take on the values:

```
STATE_NORMAL,
STATE_ACTIVE,
STATE_PRELIGHT,
STATE_SELECTED,
STATE_INSENSITIVE
```

For instance, for `STATE_SELECTED` the default foreground color is white and the default background color, dark blue.

Our function draw_brush(), which does the actual drawing on the pixmap, is then:

```
50   # Draw a rectangle on the screen
51   def draw_brush(widget, x, y):
52       rect = (int(x-5), int(y-5), 10, 10)
53       pixmap.draw_rectangle(widget.get_style().black_gc, True,
54                             rect[0], rect[1], rect[2], rect[3])
55       widget.queue_draw_area(rect[0], rect[1], rect[2], rect[3])
```

After we draw the rectangle representing the brush onto the pixmap, we call the function:

```
widget.queue_draw_area(x, y, width, height)
```

which notifies X that the area given needs to be updated. X will eventually generate an expose event (possibly combining the areas passed in several calls to draw()) which will cause our expose event handler to copy the relevant portions to the screen.

We have now covered the entire drawing program except for a few mundane details like creating the main window.

Chapter 25

Tips For Writing PyGTK Applications

This section is simply a gathering of wisdom, general style guidelines and hints to creating good PyGTK applications. Currently this section is very short, but I hope it will get longer in future editions of this tutorial.

25.1 The user should drive the interface, not the reverse

PyGTK, like other toolkits, gives you ways of invoking widgets, such as the `DIALOG_MODAL` flag passed to dialogs, that require a response from the user before the rest of the application can continue. In Python, as in other languages, it is good style to use modal interface elements as little as possible.

Every modal interaction is a place where your application is forcing a particular workflow on the user. While this is sometime unavoidable, as a general rule it is backwards; the application should be adapting itself to the user's preferred workflow instead.

A particularly common case of this, which ought to be much less so is confirmation prompts. Every confirmation prompt is a place where you should support an undo operation instead; the GIMP, the application GTK was originally built for, avoids many operations that would otherwise requite a stop-and-check with the user by having an undo command that can unwind any operation it does.

25.2 Separate your data model from your interface

Python's flexible, duck-typed object system lowers the cost of architectural options that are more difficult to exercise in more rigid languages (yes, we *are* thinking of C++). One of these is carefully separating your data model (the classes and data structures that represent whatever state your application is designed to manipulate) from your controller (the classes that implement your user interface.

In Python, a design pattern that frequently applies is to have one master editor/controller class that encapsulates your user interface (with, possibly, small helper classes for stateful widgets) and one master model class that encapsulates your application state (probably with some members that are themselves instances of small data-representation classes). The controller calls methods in the model to do all its data manipulation; the model delegates screen-painting and input-event processing to the controller.

Narrowing the interface between model and controller makes it easier to avoid being locked into early decisions about either part by adhesions with the other one. It also makes downstream maintainance and bug diagnosis easier.

25.3 How to Separate Callback Methods From Signal Handlers

25.3.1 Overview

You do not have to store all of your callback methods in one main program file. You can separate them into classes of their own, in separate files. This way your main program can derive the methods from those classes using inheritance. You end up having all the original functionality with the added benifits of easier maintenance, code reusability, and smaller file sizes, which means less of a burden for text editors.

25.3.2 Inheritance

Inheritance is a way to reuse code. A class can inherit all the functionality of other classes, and the nice thing about inheritance is that we can use it to divide a huge program into logical groups of smaller, more maintainable pieces.

Now lets spend a second on terminology. A derived class, some call this a subclass or a child class, is a class that derives some of its functionality from other classes. A base class, some call it a superclass or a parent class, is what the derived class inherits from.

Below is a short example to help you become familiar with inheritance. You can try this out in the python interpreter to gain some first hand experience.

Create two base classes:

```
class base1:
   base1_attribute = 1
   def base1_method(self):
     return "hello from base class 1"

class base2:
   base2_attribute = 2
   def base2_method(self):
     return "hello from base class 2"
```

Then create a derived class that inherits from these two base classes:

```
class derived(base1, base2):  #a class derived from two base classes
   var3 = 3
```

Now the derived class has all the functionality of the base classes.

```
x = derived()        # creates an instance of the derived class
x.base1_attribute    # 1
x.base2_attribute    # 2
x.var3               # 3
x.base1_method()     # hello from base class 1
x.base2_method()     # hello from base class 2
```

The object called x has the ability to access the variables and methods of the base classes because it has inherited their functionality. Now lets apply this concept of inheritance to a PyGTK application.

25.3.3 Inheritance Applied To PyGTK

Create a file called gui.py, then copy this code into it.

```
#A file called: gui.py

import pygtk
import gtk

# Create a class definition called gui
class gui:
  #
  #          CALLBACK METHODS
  #-----------------------------------
  def open(self, widget):
    print "opens stuff"
  def save(self, widget):
    print "save stuff"
  def undo(self, widget):
    print "undo stuff"
  def destroy(self, widget):
    gtk.main_quit()


  def __init__(self):
    #
```

```
    #        GUI CONSTRUCTION CODE
    #-----------------------------------------------
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.show()
    self.vbox1 = gtk.VBox(False, 25)
    self.window.add(self.vbox1)
    open_button = gtk.Button(label="Open Stuff")
    open_button.set_use_stock(True)
    self.vbox1.pack_start(open_button, False, False, 0)
    open_button.show()
    save_button = gtk.Button(label="Save Stuff")
    self.vbox1.pack_start(save_button, False, False, 0)
    save_button.show()
    undo_button = gtk.Button(label="Undo")
    self.vbox1.pack_start(undo_button, False, False, 0)
    undo_button.show()
    self.vbox1.show()
    #
    #        SIGNAL HANDLERS
    #------------------------------------------------
    open_button.connect("clicked", self.open)
    save_button.connect("clicked", self.save)
    undo_button.connect("clicked", self.undo)
    self.window.connect("destroy", self.destroy)

  def main(self):
   gtk.main()

if __name__ == "__main__":
 gui_instance = gui()       # create a gui object
 gui_instance.main()        # call the main method
```

If your run this program you will find it is just a simple window with some buttons. As the program is organized right now, all of the code is in one single file. But in a moment, you will find out how to break that program up into multiple files. The idea is to take those four callback methods out of the gui class and put them into classes of their own, in separate files. Now if you had hundreds of callback methods you would try and group them in some logical way, for example, you might put all of your methods that deal with input/output into the same class, and you would make other classes for other groups of methods as well.

The first thing we have to do is make some classes for the methods in the gui.py file. Create three new text files, and name them io.py, undo.py, and destroy.py, and put these files in the same directory as the gui.py file. Copy the code below into the io.py file.

```
class io:
  def open(self, widget):
    print "opens stuff"

  def save(self, widget):
    print "save stuff"
```

These are the two callback methods, open and save, from the gui.py program. Copy the next block of code into the undo.py file.

```
class undo:
  def undo(self, widget):
    print "undo stuff"
```

This is the undo_method from gui.py. And finally, copy the code below into destroy.py.

```
import gtk

class destroy:
  def destroy(self, widget):
    gtk.main_quit()
```

Now all the methods are separated into classes of their own.

IMPORTANT

In your future programs you will want to import things like gtk, pango, os ect... into your derived class(the one with all of your gui initialization code), but also, remember to import any modules or classes you need into your base classes too. Sometimes you might create an instance of a gtk widget in a base class method, in that case import gtk.

This is just an example of a base class where you would be required to import gtk.

```
import gtk

class Font_io
  def Font_Chooser(self,widget):
    self.fontchooser = gtk.FontSelectionDialog("Choose Font ↩
        ")
    self.fontchooser.show()
```

Notice it defines a gtk widget, a font selection dialog. You would normally import gtk in your main class(the derived class) and everything would be ok. But the second you take this Font_Chooser method out of your main class and put it into a class of its own, and then try to inherit from it, you would find you get an error. In this case, you would not even see any error until you were running the program. But when you try to use the Font_Chooser, you would find that gtk is not defined, even though you have imported it in your derived class. So just remember that when you create base classes, you need to add their proper imports too.

With your three classes in three separate py files, you now need to change the code in the gui.py file in three ways.

1. Import the classes you have created.
2. Change your class definition.
3. Delete your callback methods.

The updated code below shows how to do this.

```
#A file called:  gui.py
#(updated version)
#(with multiple inheritance)

import pygtk
import gtk

from io import file_io                        #
from undo import undo                         # 1. Import Your Classes
from destroy import destroy                   #

# Create a class definition called gui
class gui(io, undo, destroy):                 # 2. Changed Class Definition
                                              # 3. Deleted Callbacks
  def __init__(self):
    #
    #         GUI CONSTRUCTION CODE
    #-----------------------------------------------
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.show()
    self.vbox1 = gtk.VBox(False, 25)
```

```
    self.window.add(self.vbox1)
    open_button = gtk.Button(label="Open Stuff")
    open_button.set_use_stock(True)
    self.vbox1.pack_start(open_button, False, False, 0)
    open_button.show()
    save_button = gtk.Button(label="Save Stuff")
    self.vbox1.pack_start(save_button, False, False, 0)
    save_button.show()
    undo_button = gtk.Button(label="Undo")
    self.vbox1.pack_start(undo_button, False, False, 0)
    undo_button.show()
    self.vbox1.show()
    #
    #        SIGNAL HANDLERS
    #------------------------------------------------
    open_button.connect("clicked", self.open_method)
    save_button.connect("clicked", self.save_method)
    undo_button.connect("clicked", self.undo_method)
    self.window.connect("destroy", self.destroy)

  def main(self):
   gtk.main()

if __name__ == "__main__":
 gui_instance = gui()       # create a gui object
 gui_instance.main()        # call the main method
```

These three lines are new:

```
from io import io
from undo import undo
from destroy import destroy
```

The import statements are of the form:

```
from [filename of your class file] import [class name]
```

Here is the class definition change:

```
class gui(io, undo, destroy):
```

The names of the base classes go between the parenthesis in the class definition. Now the gui class has become a derived class, and is able to use all the attributes and methods defined in its base classes. Also, the gui class is inheriting from multiple classes(two or more), this is known as multiple inheritance.

Now change the gui.py file to the updated version and run the program again. You will notice it works exactly the same, except now all the callback methods are in separate classes, being inherited by the gui class.

There is just one other matter of interest to take note of. As long as your gui.py program and your base class files are all in the same directory, everything will work just fine. But if you want to create another directory inside there called classes, in which to organize your files of base classes, then you will need to add two more lines of code with the rest of your import statements in gui.py, like this:

```
import sys
sys.path.append("classes")
```

where classes is the name of the directory you store your classes in. This lets Python know where to look for your classes. Try it out. Just make a directory called classes in the directory where you have the gui.py program. Then put your three base class files into this classes directory. Now add the two lines of code show above to the top of the gui.py file. And thats it!

One final note for those that use py2exe for compiling python programs. Put your base classes in your Python directory, then py2exe will included them in the compiled version of your program just like any other Python module.

Chapter 26

Contributing

This document, like so much other great software out there, was created for free by volunteers. If you are at all knowledgeable about any aspect of PyGTK that does not already have documentation, please consider contributing to this document.

If you do decide to contribute, please mail your text to John Finlay (finlay@moeraki.com). Also, be aware that the entirety of this document is free, and any addition by you provide must also be free. That is, people may use any portion of your examples in their programs, and copies of this document may be distributed at will, etc.

Thank you.

Chapter 27

Credits

27.1 Original GTK+ Credits

The following credits are from the original GTK+ 1.2 and GTK+ 2.0 Tutorials (from which this tutorial has mostly copied verbatim):

- Bawer Dagdeviren, chamele0n@geocities.com for the menus tutorial.
- Raph Levien, raph@acm.org for hello world ala GTK, widget packing, and general all around wisdom. He's also generously donated a home for this tutorial.
- Peter Mattis, petm@xcf.berkeley.edu for the simplest GTK program.. and the ability to make it :)
- Werner Kochwerner.koch@guug.de for converting the original plain text to SGML, and the widget class hierarchy.
- Mark Crichton crichton@expert.cc.purdue.edu for the menu factory code, and the table packing tutorial.
- Owen Taylor owt1@cornell.edu for the EventBox widget section (and the patch to the distro). He's also responsible for the selections code and tutorial, as well as the sections on writing your own GTK widgets, and the example application. Thanks a lot Owen for all you help!
- Mark VanderBoommvboom42@calvin.edu for his wonderful work on the Notebook, Progress Bar, Dialogs, and File selection widgets. Thanks a lot Mark! You've been a great help.
- Tim Janik timj@gtk.org for his great job on the Lists Widget. His excellent work on automatically extracting the widget tree and signal information from GTK. Thanks Tim :)
- Rajat Datta rajat@ix.netcom.com for the excellent job on the Pixmap tutorial.
- Michael K. Johnson johnsonm@redhat.com for info and code for popup menus.
- David Huggins-Daines bn711@freenet.carleton.ca for the Range Widgets and Tree Widget sections.
- Stefan Mars mars@lysator.liu.se for the CList section.
- David A. Wheeler dwheeler@ida.org for portions of the text on GLib and various tutorial fixups and improvements. The GLib text was in turn based on material developed by Damon Chaplin DAChaplin@msn.com
- David King for style checking the entire document.

And to all of you who commented on and helped refine this document.
Thanks.

27.2 PyGTK Tutorial Credits

This tutorial was originally adapted from the GTK+ documentation by John Finlay.

Thanks to:

- Nathan Hurst for the `Plugs` and `Sockets` section.
- Alex Roitman for the `FileChooser` section.
- Steve George for the example program illustrating editable `CellRendererText` and activatable `CellRendererToggle`.
- Charles Wilson for the "How to Separate Callback Methods From Signal Handlers" section in the "Tips For Writing PyGTK Applications" chapter.

Much of Section 4.1 was adapted from the PyGTK FAQ item 12.2 *How does packing work (or how do I get my widget to stay the size I want)* as it existed on 21 October 2008, And had been originally written by Christian Reis.

Chapter 28

Tutorial Copyright and Permissions Notice

The PyGTK Tutorial is Copyright (C) 2001-2005 John Finlay.

The GTK Tutorial is Copyright (C) 1997 Ian Main.

Copyright (C) 1998-1999 Tony Gale.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this document under the conditions for verbatim copying, provided that this copyright notice is included exactly as in the original, and that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this document into another language, under the above conditions for modified versions.

If you are intending to incorporate this document into a published work, please contact the maintainer, and we will make an effort to ensure that you have the most up to date information available.

There is no guarantee that this document lives up to its intended purpose. This is simply provided as a free resource. As such, the authors and maintainers of the information provided within can not make any guarantee that the information is even accurate.

Appendix A

GTK Signals

As PyGTK is an object oriented widget set, it has a hierarchy of inheritance. This inheritance mechanism applies for signals. Therefore, you should refer to the widget hierarchy tree when using the signals listed in this section.

A.1 GtkObject

```
destroy(object, data)
```

A.2 GtkWidget

```
show(GtkWidget, data)

hide(widget, data)

map(widget, data)

unmap(widget, data)

realize(widget, data)

unrealize(widget, data)

draw(widget, area, data)

draw-focus(widget, data)

draw-default(widget, data)

size-request(widget, requisition, data)

size-allocate(widget, allocation, data)

state-changed(widget, state, data)

parent-set(widget, object, data)

style-set(widget, style, data)

add-accelerator(widget, accel_signal_id, accel_group, accel_key, accel_mods,
                accel_flags, data)

remove-accelerator(widget, accel_group, accel_key, accel_mods, data)
```

```
bool = event(widget, event, data)

bool = button-press-event(widget, event, data)

bool = button-release-event(widget, event, data)

bool = motion-notify-event(widget, event, data)

bool = delete-event(widget, event, data)

bool = destroy-event(widget, event, data)

bool = expose-event(widget, event, data)

bool = key-press-event(widget, event, data)

bool = key-release-event(widget, event, data)

bool = enter-notify-event(widget, event, data)

bool = leave-notify-event(widget, event, data)

bool = configure-event(widget, event, data)

bool = focus-in-event(widget, event, data)

bool = focus-out-event(widget, event, data)

bool = map-event(widget, event, data)

bool = unmap-event(widget, event, data)

bool = property-notify-event(widget, event, data)

bool = selection-clear-event(widget, event, data)

bool = selection-request-event(widget, event, data)

bool = selection-notify-event(widget, event, data)

selection-get(widget, selection_data, info, time, data)

selection-received(widget, selection_data, time, data)

bool = proximity-in-event(widget, event, data)

bool = proximity-out-event(widget, event, data)

drag-begin(widget, context, data)

drag-end(widget, context, data)

drag-data-delete(widget, context, data)

drag-leave(widget, context, time, data)

bool = drag-motion(widget, context, x, y, time, data)

bool = drag-drop(widget, context, x, y, time, data)

drag-data-get(widget, context, selection_data, info, time, data)

drag-data-received(widget, context, info, time, selection_data,
                   info, time, data)
```

```
bool = client-event(widget, event, data)

bool = no-expose-event(widget, event, data)

bool = visibility-notify-event(widget, event, data)

debug-msg(widget, string, data)
```

A.3 GtkData

```
  disconnect(data_obj, data)
```

A.4 GtkContainer

```
add(container, widget, data)

remove(container, widget, data)

check-resize(container, data)

direction = focus(container, direction, data)

set-focus-child(container, widget, data)
```

A.5 GtkCalendar

```
month-changed(calendar, data)

day-selected(calendar, data)

day-selected-double-click(calendar, data)

prev-month(calendar, data)

next-month(calendar, data)

prev-year(calendar, data)

next-year(calendar, data)
```

A.6 GtkEditable

```
changed(editable, data)

insert-text(editable, new_text, text_length, position, data)

delete-text(editable, start_pos, end_pos, data)

activate(editable, data)

set-editable(editable, is_editable, data)

move-cursor(editable, x, y, data)
```

```
move-word(editable, num_words, data)

move-page(editable, x, y, data)

move-to-row(editable, row, data)

move-to-column(editable, column, data)

kill-char(editable, direction, data)

kill-word(editable, drirection, data)

kill-line(editable, direction, data)

cut-clipboard(editable, data)

copy-clipboard(editable, data)

paste-clipboard(editable, data)
```

A.7 GtkNotebook

```
  switch-page(noteboook, page, page_num, data)
```

A.8 GtkList

```
selection-changed(list, data)

select-child(list, widget, data)

unselect-child(list, widget, data)
```

A.9 GtkMenuShell

```
deactivate(menu_shell, data)

selection-done(menu_shell, data)

move-current(menu_shell, direction, data)

activate-current(menu_shell, force_hide, data)

cancel(menu_shell, data)
```

A.10 GtkToolbar

```
orientation-changed(toolbar, orientation, data)

style-changed(toolbar, toolbar_style, data)
```

A.11 GtkButton

```
pressed(button, data)

released(button, data)

clicked(button, data)

enter(button, data)

leave(button, data)
```

A.12 GtkItem

```
select(item, data)

deselect(item, data)

toggle(item, data)
```

A.13 GtkWindow

```
set-focus(window, widget, data)
```

A.14 GtkHandleBox

```
child-attached(handle_box, widget, data)

child-detached(handle_box, widget, data)
```

A.15 GtkToggleButton

```
toggled(toggle_button, data)
```

A.16 GtkMenuItem

```
activate(menu_item, data)

activate-item(menu_item, data)
```

A.17 GtkCheckMenuItem

```
toggled(check_menu_item, data)
```

A.18 GtkInputDialog

```
enable-device(input_dialog, deviceid, data)

disable-device(input_dialog, deviceid, data)
```

A.19 GtkColorSelection

```
color-changed(color_selection, data)
```

A.20 GtkStatusBar

```
text-pushed(statusbar, context_id, text, data)

text-popped(statusbar, context_id, text, data)
```

A.21 GtkCurve

```
curve-type-changed(curve, data)
```

A.22 GtkAdjustment

```
changed(adjustment, data)

value-changed(adjustment, data)
```

Appendix B

Code Examples

B.1 scribblesimple.py

```
 1   #!/usr/bin/env python
 2
 3   # example scribblesimple.py
 4
 5  # GTK - The GIMP Toolkit
 6  # Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh MacDonald
 7  # Copyright (C) 2001-2002 John Finlay
 8  #
 9  # This library is free software; you can redistribute it and/or
10  # modify it under the terms of the GNU Library General Public
11  # License as published by the Free Software Foundation; either
12  # version 2 of the License, or (at your option) any later version.
13  #
14  # This library is distributed in the hope that it will be useful,
15  # but WITHOUT ANY WARRANTY; without even the implied warranty of
16  # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
17  # Library General Public License for more details.
18  #
19  # You should have received a copy of the GNU Library General Public
20  # License along with this library; if not, write to the
21  # Free Software Foundation, Inc., 59 Temple Place - Suite 330,
22  # Boston, MA 02111-1307, USA.
23
24
25 import gtk
26
27 # Backing pixmap for drawing area
28 pixmap = None
29
30 # Create a new backing pixmap of the appropriate size
31 def configure_event(widget, event):
32     global pixmap
33
34     x, y, width, height = widget.get_allocation()
35     pixmap = gtk.gdk.Pixmap(widget.window, width, height)
36     pixmap.draw_rectangle(widget.get_style().white_gc,
37                           True, 0, 0, width, height)
38
39     return True
40
41 # Redraw the screen from the backing pixmap
42 def expose_event(widget, event):
43     x , y, width, height = event.area
44     widget.window.draw_drawable(widget.get_style().fg_gc[gtk.STATE_NORMAL],
45                                 pixmap, x, y, x, y, width, height)
```

```
 46     return False
 47 
 48 # Draw a rectangle on the screen
 49 def draw_brush(widget, x, y):
 50     rect = (x - 5, y - 5, 10, 10)
 51     pixmap.draw_rectangle(widget.get_style().black_gc, True,
 52                           rect[0], rect[1], rect[2], rect[3])
 53     widget.queue_draw_area(rect[0], rect[1], rect[2], rect[3])
 54 
 55 def button_press_event(widget, event):
 56     if event.button == 1 and pixmap != None:
 57         draw_brush(widget, event.x, event.y)
 58     return True
 59 
 60 def motion_notify_event(widget, event):
 61     if event.is_hint:
 62         x, y, state = event.window.get_pointer()
 63     else:
 64         x = event.x
 65         y = event.y
 66         state = event.state
 67 
 68     if state & gtk.gdk.BUTTON1_MASK and pixmap != None:
 69         draw_brush(widget, x, y)
 70 
 71     return True
 72 
 73 def main():
 74     window = gtk.Window(gtk.WINDOW_TOPLEVEL)
 75     window.set_name ("Test Input")
 76 
 77     vbox = gtk.VBox(False, 0)
 78     window.add(vbox)
 79     vbox.show()
 80 
 81     window.connect("destroy", gtk.mainquit)
 82 
 83     # Create the drawing area
 84     drawing_area = gtk.DrawingArea()
 85     drawing_area.set_size_request(200, 200)
 86     vbox.pack_start(drawing_area, True, True, 0)
 87 
 88     drawing_area.show()
 89 
 90     # Signals used to handle backing pixmap
 91     drawing_area.connect("expose_event", expose_event)
 92     drawing_area.connect("configure_event", configure_event)
 93 
 94     # Event signals
 95     drawing_area.connect("motion_notify_event", motion_notify_event)
 96     drawing_area.connect("button_press_event", button_press_event)
 97 
 98     drawing_area.set_events(gtk.gdk.EXPOSURE_MASK
 99                             | gtk.gdk.LEAVE_NOTIFY_MASK
100                             | gtk.gdk.BUTTON_PRESS_MASK
101                             | gtk.gdk.POINTER_MOTION_MASK
102                             | gtk.gdk.POINTER_MOTION_HINT_MASK)
103 
104     # .. And a quit button
105     button = gtk.Button("Quit")
106     vbox.pack_start(button, False, False, 0)
107 
108     button.connect_object("clicked", lambda w: w.destroy(), window)
109     button.show()
```

```
110
111     window.show()
112
113     gtk.main()
114
115     return 0
116
117 if __name__ == "__main__":
118     main()
```

Appendix C

ChangeLog

```
2011-03-14  Rafael Villar Burke  <pachi@rvburke.com>

         * MovingOn.xml: Fix renamed methods handler_block, handler_unblock

2009-02-24  John Finlay  <finlay@moeraki.com>

        * Credits.xml (url): Add credit to Charles Wilson. Rearrange the
        credits and consolidate PyGTK credits.

        * TipsForWritingPygtkApplications.xml: Add section in Tips from
        Charles Wilson "How to Separate Callback Methods From Signal
        Handlers"

2008-03-28  John Finlay  <finlay@moeraki.com>

        * CellRenderers.xml: Fix code fragment for set_cell_data_func
        example. (Yotam Medini)

2007-03-09  Rafael Villar Burke  <pachi@rvburke.com>

        * tut/ComboBoxAndComboboxEntry.xml: Add reference to new (2.6)
        get_active_text() method Fixes #364187.
        * tut-es/ComboBoxAndComboBoxEntry.xml: fix same problem

2006-03-02  John Finlay  <finlay@moeraki.com>

        * pygtk2-tut.xml: Bump revision number and date.

        * DragAndDrop.xml: Add drag finish to dragtargets.py example.

        ============== 2.4 ==================
2005-04-13  John Finlay  <finlay@moeraki.com>

        * pygtk2-tut.xml: Set version number and pubdate.

        * Copyright.xml: Update date.

        * Replace gtk.TRUE and gtk.FALSE. Fix misc. other deprecations.

2005-03-31  John Finlay  <finlay@moeraki.com>

        * ComboBoxAndComboBoxEntry.xml: Convenience function is
        gtk.combo_box_entry_new_text(). (brett@belizebotanic.org)
```

```
2005-02-28  John Finlay  <finlay@moeraki.com>

        * GettingStarted.xml (Stepping) add print statement to destroy
        handler for illustrative purposes. (Rodolfo Gouveia)

        ============== 2.3 ==================
2004-12-24  John Finlay  <finlay@moeraki.com>

        * pygtk2-tut.xml: Set version number and pubdate. Add revhistory.

        * UIManager.xml: Add.

2004-12-13  John Finlay  <finlay@moeraki.com>

        * MovingOn.xml: Remove reference to WINODW_DIALOG (Jens Knutson)

2004-12-08  John Finlay  <finlay@moeraki.com>

        * DragAndDrop.xml Patch from Rafael Villar Burke

2004-12-01  John Finlay  <finlay@moeraki.com>

        * Scribble.xml Patch by Rafael Villar Burke.

2004-11-29  John Finlay  <finlay@moeraki.com>

        * ComboBoxAndComboBoxEntry.xml Patch by Rafael Villar Burke.
        * TimeoutsIOAndIdleFunctions.xml Patch by Rafael Villar Burke.

        * AdvancedEventAndSignalHandling.xml Add parameter tags to function
        and method defs. Patch by Rafael Villar Burke.

2004-11-20  John Finlay  <finlay@moeraki.com>

        * ColorButtonAndFontButton.xml:
        * SettingWidgetAttributes.xml: Fix xml tags. (Rafael Villar Burke)

2004-10-31  John Finlay  <finlay@moeraki.com>

        * ExpanderWidget.xml
        * GenericTreeModel.xml
        * CellRenderers.xml Fixes by Rafael Villar Burke.

2004-10-28  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml Fixes by Rafael Villar Burke.

2004-10-24  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml Many fixes by Rafael Villar Burke.

        * MiscellaneaousWidgets.xml Many fixes by Rafael Villar Burke.

2004-10-13  John Finlay  <finlay@moeraki.com>

        * PackingWidgets.xml ButtonWidget.xml Fix typos per kraai
        Fixes #155318.

2004-09-20  John Finlay  <finlay@moeraki.com>
```

```
        * TextViewWidget.xml Minor fixes by Rafael Villar Burke.

        * ActionsAndActionGroups.xml Add.
        * NewInPyGTK2.4.xml Include ActionsAndActionGroups.xml

2004-09-12  John Finlay  <finlay@moeraki.com>

        * TreeModel.xml (sec-ManagingRowData) Minor fix. Patch by
        Rafael Villar Burke

2004-09-08  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml (sec-AspectFrames)
        (sec-Alignment) Fix link to Ref manual.

2004-08-31  John Finlay  <finlay@moeraki.com>

        * DrawingArea.xml Rewrite portions based on patch by Rafael Villar
        Burke.

        * DrawingArea.xml Add missing literal tags.
        Patch by Rafael Villar Burke.

2004-08-21  John Finlay  <finlay@moeraki.com>

        * ColorButtonAndFontButton.xml Add.

        * NewInPyGTK24.xml Include ColorButtonAndFontButton.xml.

2004-08-19  John Finlay  <finlay@moeraki.com>

        * Scribble.xml (sec-DrawingAreaWidgetAndDrawing)
        Update example description.

2004-08-16  John Finlay  <finlay@moeraki.com>

        * CellRenderers.xml Add cellrenderer.py example program section

        * Credits.xml Credit Steve George for cellrenderer.py example
        program.

2004-08-15  John Finlay  <finlay@moeraki.com>

        * CellRenderers.xml (Activatable Toggle Cells) Add info about
        setting the toggle from a column. (#150212) (Steve George)

2004-08-13  John Finlay  <finlay@moeraki.com>

        * TreeModel.xml Clean up Adding TreeStore rows section (Joey Tsai)
        Add missing text in Large Data Stores section.. (Joey Tsai)

2004-08-08  John Finlay  <finlay@moeraki.com>

        * TreeModel.xml
        * CellRenderers.xml
        * GenericTreeModel.xml
        * TreeViewWidget.xml
        * NewWidgetsAndObjects.xml Minor rewording and addition of tags.
```

```
2004-08-06  John Finlay  <finlay@moeraki.com>

        * ButtonWidget.xml Fix errors in examples.
        * DragAndDrop.xml Fix anchor.
        * MenuWidget.xml Fix typo.
        * PackingWidgets.xml Fix typo.
        * MiscellaneousWidgets.xml (Dialogs) (Images) (Pixmaps) (Rulers)
        (Progressbar) (Label)
        Fix faulty wording and errors (All thanks to Marc Verney)

2004-08-04  John Finlay  <finlay@moeraki.com>

        * DrawingArea.xml Update example to use rulers and scrolled
        window.

        * pygtk2-tut.xml Bump version number and pubdate.

        ============== 2.2 ==================
2004-08-03  John Finlay  <finlay@moeraki.com>

        * ComboBoxAndComboBoxEntry.xml Add.

        * EntryCompletion.xml Add.

        * ExpanderWidget.xml Add.

        * NewInPyGTK24.xml Add.

        * NewWidgetsAndObject.xml Rearrange and make as a chapter.

        * pygtk2-tut.xml Add NewInPyGTK24.xml. Update date.

2004-08-02  John Finlay  <finlay@moeraki.com>

        * MiscellaneousWidgets.xml Change Combo Box to Combo Widget to avoid
        confusion with new ComboBox widget. Add deprecation note.

2004-07-28  John Finlay  <finlay@moeraki.com>

        * Credits.xml Add PyGTK section with credit to Nathan Durst and
        Alex Roitman.

        * FileChooser.xml Create.
        * NewWidgetsAndObject.xml Add include for FileChooser.xml.

        * NewWidgetsAndObject.xml Create.

        * pygtk2-tut.xml Add NewWidgetsAndObject.xml file. Bump version
        number and set date.

2004-07-20  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (sec-ManagingCellRenderers) Fix title.
        More detail on set_sort_column_id().

2004-07-12  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (sec-CreatingTreeView) Fix faulty capitalization.
```

```
        (thanks to Doug Quale)

2004-07-08  John Finlay  <finlay@moeraki.com>

        * Adjustments.xml AdvancedEventAndSignalHandling.xml
        ButtonWidget.xml ChangeLog ContainerWidgets.xml
        DragAndDrop.xml DrawingArea.xml GettingStarted.xml
        ManagingSelections.xml MenuWidget.xml
        MiscellaneousWidgets.xml MovingOn.xml
        PackingWidgets.xml RangeWidgets.xml Scribble.xml
        SettingWidgetAttributes.xml TextViewWidget.xml
        TimeoutsIOAndIdleFunctions.xml WidgetOverview.xml
        Update files with example programs.

2004-07-06  John Finlay  <finlay@moeraki.com>

        * examples/*.py Update examples to eliminate deprecated methods
        and use import pygtk.

        ============== 2.1 ==================
2004-07-06  John Finlay  <finlay@moeraki.com>

        * pygtk2-tut.xml Bump version number to 2.1 and set
        pubdate.

        * TreeViewWidgets.xml Revise the treeviewdnd.py example to
        illustrate row reordering with external drag and drop and
        add explanation.

2004-07-03  John Finlay  <finlay@moeraki.com>

        * TimeoutsIOAndIdleFunctions.xml Update descriptions to use
        the gobject functions.

2004-06-30  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml Extract the CellRenderers section into
        CellRenderers.xml.

        * CellRenderers.xml Create and add section on editable
        CellRendererText.

        * TreeViewWidget.xml Extract the TreeModel section and put into
        new file TreeModel.xml. Add detail to the TreeViewColumn use of
        its sort column ID.

        * TreeModel.xml Create and add section on sorting TreeModel rows
        using the TreeSortable interface.

2004-06-27  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (Cell Data Function) Add filelisting example
        using cell data functions. Add XInclude header to include generic
        tree model and cell renderer subsections.
        Fix typos and errors in links. Fix bugs in example listings.
        Add section on TreeModel signals.

2004-06-22  John Finlay  <finlay@moeraki.com>
```

```
        * Introduction.xml Add note about pygtkconsole.py and gpython.py
        programs do not work on Windows. Thanks to vector180.

2004-06-14  John Finlay  <finlay@moeraki.com>

        * DragAndDrop.xml Fix signal lists for drag source and dest.
        Add detail to the overview drag cycle. Add detail about signal
        handler operation.

        * DragAndDrop.xml Add small example program dragtargets.py to
        print out drag targets.

2004-05-31  John Finlay  <finlay@moeraki.com>

        * GettingStarted.xml Change wording in helloworld.py example
        program - delete_event() comments confusing. Thanks to Ming Hua.

2004-05-28  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (TreeModelFilter) Replace 'file' with 'filter'.
        Thanks to Guilherme Salgado.

2004-05-27  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (AccessingDataValues) Fix store.set example
        column number wrong. Thanks to Rafael Villar Burke and Guilherme
        Salgado.
        (CellRendererAttributes) Fix error. Thanks to Doug Quale.
        (TreeModelIntroduction)
        (PythonProtocolSupport) Fix grammatical and spelling errors.
        Thanks to Thomas Mills Hinkle.

2004-05-25  John Finlay  <finlay@moeraki.com>

        * Introduction.xml Add reference links to www.pygtk.org website
        and describe some of its resources.

        ============ 2.0 ==============
2004-05-24  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml Add beginning of tutorial chapter.

        * Introduction.xml Add reference to gpython.py program.

        * pygtk2-tut.xml Bump release number to 2.0.

2004-03-31  John Finlay  <finlay@moeraki.com>

        * MiscellaneousWidgets.xml Fix bug in calendar.py example causing
        date string to be off by one day in some time zones. Fixes #138487.
        (thanks to Eduard Luhtonen)

2004-01-28  John Finlay  <finlay@moeraki.com>

        * DrawingArea.xml Modify description of DrawingArea to clarify that
        drawing is done on the wrapped gtk.gdk.Window. Modify GC description
        to clarify that new GCs created from drawables. (thanks to Antoon
        Pardon)
```

```
        * UndocumentedWidgets.xml Remove the section on Plugs and Sockets -
        now in ContainerWidgets.xml.

        * ContainerWidgets.xml Add section on Plugs and Sockets written
        by Nathan Hurst.

        * pygtk2-tut.xml Change date and version number.

2003-11-05  John Finlay  <finlay@moeraki.com>

        * Introduction.xml Add reference to the PyGTK 2.0 Reference Manual.

2003-11-04  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml
        * RangeWidgets.xml
        * WidgetOverview.xml Remove reference to testgtk.py since it doesn't
        exist in PyGTK 2.0 (thanks to Steve Chaplin)

2003-10-07  John Finlay  <finlay@moeraki.com>

        * TextViewWidget.xml Change PANGO_ to pango. (thanks to Stephane Klein)
        * pygtk2-tut.xml Change date and version number.

2003-10-06  John Finlay  <finlay@moeraki.com>

        * GettingStarted.xml Change third to second in description of signal
        handler arguments. (thanks to Kyle Smith)

2003-09-26  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml Fix text layout error in frame shadow
        description (thanks to Steve Chaplin)

2003-09-19  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml
        * layout.py Use random module instead of whrandom in
        layout.py example program (thanks to Steve Chaplin)

        * PackingWidgets.xml
        * packbox.py Use set_size_request() instead of set_usize() in
        packbox.py example (thanks to Steve Chaplin)

2003-07-11  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml Fix link references to class-gtkalignment to
        use a ulink instead of a link.

        * ChangeLog Add this change log file

        * pygtk2-tut.xml Change date and add a version number. Add ChangeLog
        as an appendix.
```