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
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.7</td>
<td>Rulers</td>
<td>77</td>
</tr>
<tr>
<td>9.8</td>
<td>Statusbars</td>
<td>79</td>
</tr>
<tr>
<td>9.9</td>
<td>Text Entries</td>
<td>81</td>
</tr>
<tr>
<td>9.10</td>
<td>Spin Buttons</td>
<td>83</td>
</tr>
<tr>
<td>9.11</td>
<td>Combo Widget</td>
<td>89</td>
</tr>
<tr>
<td>9.12</td>
<td>Calendar</td>
<td>90</td>
</tr>
<tr>
<td>9.13</td>
<td>Color Selection</td>
<td>96</td>
</tr>
<tr>
<td>9.14</td>
<td>File Selection</td>
<td>100</td>
</tr>
<tr>
<td>9.15</td>
<td>Font Selection Dialog</td>
<td>102</td>
</tr>
<tr>
<td>10</td>
<td>Container Widgets</td>
<td>105</td>
</tr>
<tr>
<td>10.1</td>
<td>The EventBox</td>
<td>105</td>
</tr>
<tr>
<td>10.2</td>
<td>The Alignment widget</td>
<td>106</td>
</tr>
<tr>
<td>10.3</td>
<td>Fixed Container</td>
<td>107</td>
</tr>
<tr>
<td>10.4</td>
<td>Layout Container</td>
<td>108</td>
</tr>
<tr>
<td>10.5</td>
<td>Frames</td>
<td>111</td>
</tr>
<tr>
<td>10.6</td>
<td>Aspect Frames</td>
<td>113</td>
</tr>
<tr>
<td>10.7</td>
<td>Paned Window Widgets</td>
<td>115</td>
</tr>
<tr>
<td>10.8</td>
<td>Viewports</td>
<td>118</td>
</tr>
<tr>
<td>10.9</td>
<td>Scrolled Windows</td>
<td>118</td>
</tr>
<tr>
<td>10.10</td>
<td>Button Boxes</td>
<td>120</td>
</tr>
<tr>
<td>10.11</td>
<td>Toolbar</td>
<td>124</td>
</tr>
<tr>
<td>10.12</td>
<td>Notebooks</td>
<td>128</td>
</tr>
<tr>
<td>10.13</td>
<td>Plugs and Sockets</td>
<td>133</td>
</tr>
<tr>
<td>10.13.1</td>
<td>Plugs</td>
<td>133</td>
</tr>
<tr>
<td>10.13.2</td>
<td>Sockets</td>
<td>134</td>
</tr>
<tr>
<td>11</td>
<td>Menu Widget</td>
<td>137</td>
</tr>
<tr>
<td>11.1</td>
<td>Manual Menu Creation</td>
<td>137</td>
</tr>
<tr>
<td>11.2</td>
<td>Manual Menu Example</td>
<td>139</td>
</tr>
<tr>
<td>11.3</td>
<td>Using ItemFactory</td>
<td>141</td>
</tr>
<tr>
<td>11.4</td>
<td>Item Factory Example</td>
<td>141</td>
</tr>
<tr>
<td>12</td>
<td>Drawing Area</td>
<td>145</td>
</tr>
<tr>
<td>12.1</td>
<td>Graphics Context</td>
<td>145</td>
</tr>
<tr>
<td>12.2</td>
<td>Drawing Methods</td>
<td>149</td>
</tr>
<tr>
<td>13</td>
<td>TextView Widget</td>
<td>155</td>
</tr>
<tr>
<td>13.1</td>
<td>TextView Overview</td>
<td>155</td>
</tr>
<tr>
<td>13.2</td>
<td>TextViews</td>
<td>155</td>
</tr>
<tr>
<td>13.3</td>
<td>Text Buffers</td>
<td>160</td>
</tr>
<tr>
<td>13.3.1</td>
<td>TextBuffer Status Information</td>
<td>161</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Creating TextIters</td>
<td>161</td>
</tr>
<tr>
<td>13.3.3</td>
<td>Text Insertion, Retrieval and Deletion</td>
<td>162</td>
</tr>
<tr>
<td>13.3.4</td>
<td>TextMarks</td>
<td>163</td>
</tr>
<tr>
<td>13.3.5</td>
<td>Creating and Applying TextTags</td>
<td>163</td>
</tr>
<tr>
<td>13.3.6</td>
<td>Inserting Images and Widgets</td>
<td>164</td>
</tr>
<tr>
<td>13.4</td>
<td>TextIters</td>
<td>165</td>
</tr>
<tr>
<td>13.4.1</td>
<td>TextIter Attributes</td>
<td>165</td>
</tr>
<tr>
<td>13.4.2</td>
<td>Text Attributes at a TextIter</td>
<td>165</td>
</tr>
<tr>
<td>13.4.3</td>
<td>Copying a TextIter</td>
<td>166</td>
</tr>
<tr>
<td>13.4.4</td>
<td>Retrieving Text and Objects</td>
<td>166</td>
</tr>
<tr>
<td>13.4.5</td>
<td>Checking Conditions at a TextIter</td>
<td>166</td>
</tr>
<tr>
<td>13.4.6</td>
<td>Checking Location in Text</td>
<td>167</td>
</tr>
<tr>
<td>13.4.7</td>
<td>Moving Through Text</td>
<td>168</td>
</tr>
<tr>
<td>13.4.8</td>
<td>Moving to a Specific Location</td>
<td>168</td>
</tr>
<tr>
<td>13.4.9</td>
<td>Searching in Text</td>
<td>169</td>
</tr>
<tr>
<td>13.5</td>
<td>Text Marks</td>
<td>169</td>
</tr>
<tr>
<td>13.6</td>
<td>Text Tags and Tag Tables</td>
<td>170</td>
</tr>
</tbody>
</table>
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 TreeIterers .............................................. 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
CONTENTS

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 GenericCellRenderer ................................................ 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
## CONTENTS

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
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
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.7</td>
<td>Scrolled Window Example</td>
<td>119</td>
</tr>
<tr>
<td>10.8</td>
<td>Toolbar Example</td>
<td>128</td>
</tr>
<tr>
<td>10.9</td>
<td>Notebook Example</td>
<td>130</td>
</tr>
<tr>
<td>11.1</td>
<td>Menu Example</td>
<td>139</td>
</tr>
<tr>
<td>11.2</td>
<td>Item Factory Example</td>
<td>142</td>
</tr>
<tr>
<td>12.1</td>
<td>Drawing Area Example</td>
<td>151</td>
</tr>
<tr>
<td>13.1</td>
<td>Basic TextView Example</td>
<td>157</td>
</tr>
<tr>
<td>13.2</td>
<td>TextView Example</td>
<td>173</td>
</tr>
<tr>
<td>14.1</td>
<td>Basic TreeView Example</td>
<td>178</td>
</tr>
<tr>
<td>14.2</td>
<td>TreeViewColumns with CellRenderers</td>
<td>191</td>
</tr>
<tr>
<td>14.3</td>
<td>CellRenderer Data Function</td>
<td>194</td>
</tr>
<tr>
<td>14.4</td>
<td>File Listing Example Using Cell Data Functions</td>
<td>196</td>
</tr>
<tr>
<td>14.5</td>
<td>CellRendererText Markup</td>
<td>197</td>
</tr>
<tr>
<td>14.6</td>
<td>Editable and Activatable Cells</td>
<td>200</td>
</tr>
<tr>
<td>14.7</td>
<td>Expander Arrow in Second Column</td>
<td>202</td>
</tr>
<tr>
<td>14.8</td>
<td>TreeView Drag and Drop Example</td>
<td>209</td>
</tr>
<tr>
<td>14.9</td>
<td>TreeModelSort Example</td>
<td>211</td>
</tr>
<tr>
<td>14.10</td>
<td>TreeModelFilter Visibility Example</td>
<td>213</td>
</tr>
<tr>
<td>14.11</td>
<td>Generic TreeModel Example Program</td>
<td>218</td>
</tr>
<tr>
<td>15.1</td>
<td>Clipboard Example</td>
<td>226</td>
</tr>
<tr>
<td>16.1</td>
<td>Simple Action Example</td>
<td>230</td>
</tr>
<tr>
<td>16.2</td>
<td>Basic Action Example</td>
<td>232</td>
</tr>
<tr>
<td>16.3</td>
<td>Actions Example</td>
<td>235</td>
</tr>
<tr>
<td>16.4</td>
<td>ActionGroup Example</td>
<td>237</td>
</tr>
<tr>
<td>16.5</td>
<td>Basic ComboBox</td>
<td>239</td>
</tr>
<tr>
<td>16.6</td>
<td>ComboBox with Wrapped Layout</td>
<td>241</td>
</tr>
<tr>
<td>16.7</td>
<td>Basic ComboBoxEntry</td>
<td>243</td>
</tr>
<tr>
<td>16.8</td>
<td>ColorButton Example</td>
<td>245</td>
</tr>
<tr>
<td>16.9</td>
<td>FontButton Example</td>
<td>247</td>
</tr>
<tr>
<td>16.10</td>
<td>EntryCompletion</td>
<td>248</td>
</tr>
<tr>
<td>16.11</td>
<td>Expander Widget</td>
<td>249</td>
</tr>
<tr>
<td>16.12</td>
<td>File Selection Example</td>
<td>251</td>
</tr>
<tr>
<td>16.13</td>
<td>Simple.UIManager Example</td>
<td>257</td>
</tr>
<tr>
<td>16.14</td>
<td>UIManagerExample</td>
<td>259</td>
</tr>
<tr>
<td>21.1</td>
<td>Get Selection Example</td>
<td>275</td>
</tr>
<tr>
<td>21.2</td>
<td>Set Selection Example</td>
<td>278</td>
</tr>
</tbody>
</table>
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 jmcd@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 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:

```python
>>> 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!

>>> 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
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     def __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.
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__`
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).

```python
#!/usr/bin/env python
# example helloworld.py

import pygtk
pygtk.require('2.0')
import gtk

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

    def delete_event(self, widget, event, data=None):
        # If you return FALSE in the "delete_event" signal handler,
        # GTK will emit the "destroy" signal. Returning TRUE means
        # you don't want the window to be destroyed.
        # This is useful for popping up 'are you sure you want to quit?'
        # type dialogs.
        print "delete event occurred"
        return False

    def destroy(self, widget, data=None):
        gtk.main_quit()

    def __init__(self):
        # create a new window
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        # When the window is given the "delete_event" signal (this is given
        # by the window manager, usually by the "close" option, or on the
        # titlebar), we ask it to call the delete_event() function
        # as defined above. The data passed to the callback
        # function is NULL and is ignored in the callback function.
        self.window.connect("delete_event", self.delete_event)

        # Here we connect the "destroy" event to a signal handler.
        # This event occurs when we call gtk_widget_destroy() on the window,
        # or if we return FALSE in the "delete_event" callback.
        self.window.connect("destroy", self.destroy)

        # Sets the border width of the window.
```

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.
self.window.set_border_width(10)

# Creates a new button with the label "Hello World".
self.button = gtk.Button("Hello World")

# When the button receives the "clicked" signal, it will call the
# function hello() passing it None as its argument. The hello()
# function is defined above.
self.button.connect("clicked", self.hello, None)

# This will cause the window to be destroyed by calling
# gtk_widget_destroy(window) when "clicked". Again, the destroy
# signal could come from here, or the window manager.
self.button.connect_object("clicked", gtk.Widget.destroy, self.
←

# This packs the button into the window (a GTK container).
self.window.add(self.button)

# The final step is to display this newly created widget.
self.button.show()

# and the window
self.window.show()

def main(self):
    # All PyGTK applications must have a gtk.main(). Control ends here
    # and waits for an event to occur (like a key press or mouse event).
    gtk.main()

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

Figure 2.2 shows the window created by helloworld.py.

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

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.
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:

<table>
<thead>
<tr>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>button_press_event</td>
</tr>
<tr>
<td>button_release_event</td>
</tr>
<tr>
<td>scroll_event</td>
</tr>
<tr>
<td>motion_notify_event</td>
</tr>
<tr>
<td>delete_event</td>
</tr>
<tr>
<td>destroy_event</td>
</tr>
<tr>
<td>expose_event</td>
</tr>
<tr>
<td>key_press_event</td>
</tr>
<tr>
<td>key_release_event</td>
</tr>
<tr>
<td>enter_notify_event</td>
</tr>
<tr>
<td>leave_notify_event</td>
</tr>
<tr>
<td>configure_event</td>
</tr>
<tr>
<td>focus_in_event</td>
</tr>
<tr>
<td>focus_out_event</td>
</tr>
<tr>
<td>map_event</td>
</tr>
<tr>
<td>unmap_event</td>
</tr>
<tr>
<td>property_notify_event</td>
</tr>
<tr>
<td>selection_clear_event</td>
</tr>
<tr>
<td>selection_request_event</td>
</tr>
<tr>
<td>selection_notify_event</td>
</tr>
<tr>
<td>proximity_in_event</td>
</tr>
<tr>
<td>proximity_out_event</td>
</tr>
<tr>
<td>visibility_notify_event</td>
</tr>
<tr>
<td>client_event</td>
</tr>
<tr>
<td>no_expose_event</td>
</tr>
<tr>
<td>window_state_event</td>
</tr>
</tbody>
</table>

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:

```python
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
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 the button is clicked 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.

![Code snippet](image)

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.

![Code snippet](image)

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.

![Code snippet](image)

Widgets also have a `hide()` that is the opposite of `show()`. It doesn't actually destroy the widget, but it removes the widget rendering from your display. This can be reversed with another `show()` call. Lines 73-75 define the `main()` method which calls the `gtk.main()` function.

![Code snippet](image)

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.

![Code snippet](image)

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

Let's take another look at the `connect()` call.

```python
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:

```python
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.

```python
object.handler_block(handler_id)
object.handler_unblock(handler_id)
```

3.2 An Upgraded Hello World

```python
#!/usr/bin/env python
#
# example helloworld2.py
#
import pygtk
pygtk.require('2.0')
import gtk

class HelloWorld2:
    # Our new improved callback. The data passed to this method
    # is printed to stdout.
    def callback(self, widget, data):
        print "Hello again - %s was pressed" % data
    
    # another callback
    def delete_event(self, widget, event, data=None):
        gtk.main_quit()
        return False
    
    def __init__(self):
        # Create a new window
```

```python
self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
# This is a new call, which just sets the title of our
# new window to "Hello Buttons!"
self.window.set_title("Hello Buttons!")

# Here we just set a handler for delete_event that immediately
# exits GTK.
self.window.connect("delete_event", self.delete_event)

# Sets the border width of the window.
self.window.set_border_width(10)

# We create a box to pack widgets into. This is described in detail
# in the "packing" section. The box is not really visible, it
# is just used as a tool to arrange widgets.
selbox1 = gtk.HBox(False, 0)

# Put the box into the main window.
selbox1.pack_start(self.button1, True, True, 0)

# Creates a new button with the label "Button 1".
self.button1 = gtk.Button("Button 1")

# Now when the button is clicked, we call the "callback" method
# with a pointer to "button 1" as its argument
self.button1.connect("clicked", self.callback, "button 1")

# Instead of add(), we pack this button into the invisible
# box, which has been packed into the window.
self.box1.pack_start(self.button1, True, True, 0)

# Always remember this step, this tells GTK that our preparation
# for this button is complete, and it can now be displayed.
selbox1.show()

self.button2 = gtk.Button("Button 2")

# Call the same callback method with a different argument,
# passing a pointer to "button 2" instead.
self.button2.connect("clicked", self.callback, "button 2")

# The order in which we show the buttons is not really important,
# but I recommend showing the window last, so it all pops up at once.
selbox1.show()

def main():
    gtk.main()
if __name__ == "__main__":
    hello = HelloWorld2()
    main()
```

Running helloworld2.py produces the window illustrated in Figure 3.1.
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](image)

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.
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:
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=True, fill=True, padding=0)
```

*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)
```

*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

![Image](Image)

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()

![Image](Image)

### 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.

```python
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. ⤐
11 # for the variables we’re interested are passed in to this function. We do
```
# not show the box, but do show everything inside.

def make_box(homogeneous, spacing, expand, fill, padding):
    
    # Create a new hbox with the appropriate homogeneous
    # and spacing settings
    box = gtk.HBox(homogeneous, spacing)

    # Create a series of buttons with the appropriate settings
    button = gtk.Button("box.pack")
    box.pack_start(button, expand, fill, padding)
    button.show()

    button = gtk.Button("(button,")
    box.pack_start(button, expand, fill, padding)
    button.show()

    # Create a button with the label depending on the value of
    # expand.
    if expand == True:
        button = gtk.Button("True,")
    else:
        button = gtk.Button("False,")
    box.pack_start(button, expand, fill, padding)
    button.show()

    # This is the same as the button creation for "expand"
    # above, but uses the shorthand form.
    button = gtk.Button(("False","True,")[fill==True])
    box.pack_start(button, expand, fill, padding)
    button.show()

    padstr = "%d)" % padding

    button = gtk.Button(padstr)
    box.pack_start(button, expand, fill, padding)
    button.show()

    return box

class PackBox1:
    def delete_event(self, widget, event, data=None):
        gtk.main_quit()
        return False

    def __init__(self, which):

        # Create our window
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        # You should always remember to connect the delete_event signal
        # to the main window. This is very important for proper intuitive
        # behavior
        self.window.connect("delete_event", self.delete_event)
        self.window.set_border_width(10)

        # We create a vertical box (vbox) to pack the horizontal boxes into
        # This allows us to stack the horizontal boxes filled with buttons
        # on top of the other in this vbox.
        vbox = gtk.VBox(False, 0)

        # which example to show. These correspond to the pictures above.
if which == 1:
    # create a new label.
    label = gtk.Label("HBox(False, 0)")
    
    # Align the label to the left side. We’ll discuss this method
    # and others in the section on Widget Attributes.
    label.set_alignment(0, 0)
    
    # Pack the label into the vertical box (vbox box1). Remember other in
    # widgets added to a vbox will be packed one on top of the other.
    box1.pack_start(label, False, False, 0)
    label.show()

    # Call our make box function - homogeneous = False, spacing = 0,
    # expand = False, fill = False, padding = 0
    box2 = make_box(False, 0, False, False, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Call our make box function - homogeneous = False, spacing = 0,
    # expand = True, fill = False, padding = 0
    box2 = make_box(False, 0, True, False, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(False, 0, True, True, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Creates a separator, we’ll learn more about these later,
    # but they are quite simple.
    separator = gtk.HSeparator()

    # Pack the separator into the vbox. Remember each of these
    # widgets is being packed into a vbox, so they’ll be stacked vertically.
    box1.pack_start(separator, False, True, 5)
    separator.show()

    # Create another new label, and show it.
    label = gtk.Label("HBox(True, 0)")
    label.set_alignment(0, 0)
    box1.pack_start(label, False, False, 0)
    label.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(True, 0, True, False, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(True, 0, True, True, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Another new separator.
CHAPTER 4. PACKING WIDGETS

4.3. PACKING DEMONSTRATION PROGRAM

```python
separator = gtk.HSeparator()
# The last 3 arguments to pack_start are:
# expand, fill, padding.
box1.pack_start(separator, False, True, 5)
separator.show()

elif which == 2:
    # Create a new label, remember box1 is a vbox as created
    # near the beginning of __init__()
    label = gtk.Label("HBox(False, 10)")
    label.set_alignment(0, 0)
    box1.pack_start(label, False, False, 0)
    label.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(False, 10, True, False, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(False, 10, True, True, 0)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    separator = gtk.HSeparator()
    # The last 3 arguments to pack_start are:
    # expand, fill, padding.
    box1.pack_start(separator, False, True, 5)
    separator.show()

    label = gtk.Label("HBox(False, 0)")
    label.set_alignment(0, 0)
    box1.pack_start(label, False, False, 0)
    label.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(False, 0, True, False, 10)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    # Args are: homogeneous, spacing, expand, fill, padding
    box2 = make_box(False, 0, True, True, 10)
    box1.pack_start(box2, False, False, 0)
    box2.show()

    separator = gtk.HSeparator()
    # The last 3 arguments to pack_start are:
    # expand, fill, padding.
    box1.pack_start(separator, False, True, 5)
    separator.show()

    elif which == 3:
        # This demonstrates the ability to use pack_end() to
        # right justify widgets. First, we create a new box as before.
        box2 = make_box(False, 0, False, False, 0)

        # Create the label that will be put at the end.
        label = gtk.Label("end")
        # Pack it using pack_end(), so it is put on the right
        # side of the hbox created in the make_box() call.
        box2.pack_end(label, False, False, 0)
        # Show the label.
        label.show()
```

25
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 ←
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 ←
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()
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:

```python
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:

```python
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:

<table>
<thead>
<tr>
<th>FILL</th>
<th>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.</th>
</tr>
</thead>
</table>
SHRINK  If the table widget was allocated less space than 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.

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

and

```python
   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:

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

and,

```python
   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:

```python
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
```
# This callback quits the program
def delete_event(self, widget, event, data=None):
    gtk.main_quit()
    return False

def __init__(self):
    # Create a new window
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    # Set the window title
    self.window.set_title("Table")
    # Set a handler for delete_event that immediately
    # exits GTK.
    self.window.connect("delete_event", self.delete_event)
    # Sets the border width of the window.
    self.window.set_border_width(20)
    # Create a 2x2 table
    table = gtk.Table(2, 2, True)
    # Put the table in the main window
    self.window.add(table)
    # Create first button
    button = gtk.Button("button 1")
    # When the button is clicked, we call the "callback" method
    # with a pointer to "button 1" as its argument
    button.connect("clicked", self.callback, "button 1")
    # Insert button 1 into the upper left quadrant of the table
    table.attach(button, 0, 1, 0, 1)
    button.show()

    # Create second button
    button = gtk.Button("button 2")
    # When the button is clicked, we call the "callback" method
    # with a pointer to "button 2" as its argument
    button.connect("clicked", self.callback, "button 2")
    # Insert button 2 into the upper right quadrant of the table
    table.attach(button, 1, 2, 0, 1)
    button.show()

    # Create "Quit" button
    button = gtk.Button("Quit")
    # When the button is clicked, we call the main_quit function
    # and the program exits
    button.connect("clicked", lambda w: gtk.main_quit())
    # Insert the quit button into the both lower quadrants of the table
    table.attach(button, 0, 2, 1, 2)
    button.show()

    table.show()
self.window.show()
def main():
    gtk.main()
    return 0

if __name__ == '__main__':
    Table()
    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
```
CHAPTER 5. WIDGET OVERVIEW

5.1. WIDGET HIERARCHY

```
   +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.SeparatorToolItem
   |   |   +gtk.ScrolledWindow
   |   |   + 'gtk.Viewport
   |   |   +gtk.Box
   |   |   + gtk.ButtonBox
   |   |   + 'gtk.ButtonBox
   |   |   + 'gtk.VButtonBox
   |   |   + gtk.VBox
   |   |   + 'gtk.ColorSelection
   |   |   + 'gtk.FontSelection
   |   |   + 'gtk.GammaCurve
   |   |   + 'gtk.HBox
   |   |   + gtk.Combo
   |   |   + 'gtk.Statusbar
   |   |   +gtk.Fixed
   |   |   + gtk.Paned
   |   |   + 'gtk.Paned
   |   |   + '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
```

32
CHAPTER 5. WIDGET OVERVIEW

5.1. WIDGET HIERARCHY

|     | +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:

```python
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:

```python
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
```
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:

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

# example-start buttons buttons.py

import pygtk
pygtk.require('2.0')
import gtk

# Create a new hbox with an image and a label packed into it
# and return the box.
```
def xpm_label_box(parent, xpm_filename, label_text):
    # Create box for xpm and label
    box1 = gtk.HBox(False, 0)
    box1.set_border_width(2)

    # Now on to the image stuff
    image = gtk.Image()
    image.set_from_file(xpm_filename)

    # Create a label for the button
    label = gtk.Label(label_text)

    # Pack the pixmap and label into the box
    box1.pack_start(image, False, False, 3)
    box1.pack_start(label, False, False, 3)

    image.show()
    label.show()
    return box1

class Buttons:
    # Our usual callback method
    def callback(self, widget, data=None):
        print "Hello again - %s was pressed" % data

    def __init__(self):
        # Create a new window
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Image'd Buttons!")

        # It's a good idea to do this for all windows.
        self.window.connect("destroy", lambda wid: gtk.main_quit())
        self.window.connect("delete_event", lambda a1, a2: gtk.main_quit())

        # Sets the border width of the window.
        self.window.set_border_width(10)

        # Create a new button
        button = gtk.Button()

        # Connect the "clicked" signal of the button to our callback
        button.connect("clicked", self.callback, "cool button")

        # This calls our box creating function
        box1 = xpm_label_box(self.window, "info.xpm", "cool button")

        # Pack and show all our widgets
        button.add(box1)

        box1.show()
        button.show()
        self.window.add(button)
        self.window.show()

    def main():
        gtk.main()
        return 0

    if __name__ == "__main__":
        Buttons()
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:

```python
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:

```python
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:

```python
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.

```python
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:

```python
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
```
CHAPTER 6. THE BUTTON WIDGET

6.3 Check Buttons

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 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.
CHAPTER 6. THE BUTTON WIDGET

6.3. CHECK BUTTONS

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")
```
6.4 Radio Buttons

Radio buttons are similar to check buttons except they are grouped so that only one may be selected/pressed 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:
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](image)

The source code for the example program is:

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

# example radiobuttons.py

import pygtk
pygtk.require('2.0')
import gtk

class RadioButtons:
    def callback(self, widget, data=None):
        print "%s was toggled %s" % (data, ("OFF", "ON") [widget.get_active ()])

    def close_application(self, widget, event, data=None):
        gtk.main_quit()
        return False

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)

        self.window.connect("delete_event", self.close_application)

        self.window.set_title("radio buttons")
        self.window.set_border_width(0)

        box1 = gtk.VBox(False, 10)
        box1.set_border_width(10)
        box1.pack_start(box2, True, True, 0)
        box2.show()

        box2 = gtk.VBox(False, 0)
        self.window.add(box1)
        box1.show()
CHAPTER 6. THE BUTTON WIDGET  

6.4. RADIO BUTTONS

The code is fairly straightforward. 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 Adjustments 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:

```python
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:

```python
# 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:

```python
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:

```python
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:

```python
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:

```python
def cb_rotate_picture(adj, picture):
    set_picture_rotation(picture, adj.value)
    ...
```
and connect it to the scale widget’s adjustment like this:

```python
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:

```python
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:

```python
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:

```python
hscrollbar = gtk.HScrollerbar(adjustment=None)
vscrollbar = gtk.VScrollerbar(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:

```python
hscale = gtk.HScale(adjustment=None)
vscale = gtk.VScale(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:

```python
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:

```python
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:

```python
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.
CHAPTER 8. RANGE WIDGETS

8.4 KEY AND MOUSE BINDINGS

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

```python
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:

```python
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:

```python
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:
The rangewidgets.py source code is:

```python
#!/usr/bin/env python
# example rangewidgets.py
import pygtk
pygtk.require('2.0')
import gtk

# Convenience functions

def make_menu_item(name, callback, data=None):
    item = gtk.MenuItem(name)
    item.connect("activate", callback, data)
    item.show()
    return item

def scale_set_default_values(scale):
    scale.set_update_policy(gtk.UPDATE_CONTINUOUS)
    scale.set_digits(1)
    scale.set_value_pos(gtk.POS_TOP)
    scale.set_draw_value(True)

class RangeWidgets:
    def cb_pos_menu_select(self, item, pos):
```

Figure 8.1 Range Widgets Example
# Set the value position on both scale widgets
self.hscale.set_value_pos(pos)
self.vscale.set_value_pos(pos)

def cb_update_menu_select(self, item, policy):
    # Set the update policy for both scale widgets
    self.hscale.set_update_policy(policy)
    self.vscale.set_update_policy(policy)

def cb_digits_scale(self, adj):
    # Set the number of decimal places to which adj->value is rounded
    self.hscale.set_digits(adj.value)
    self.vscale.set_digits(adj.value)

def cb_page_size(self, get, set):
    # Set the page size and page increment size of the sample
    # adjustment to the value specified by the "Page Size" scale
    set.page_size = get.value
    set.page_incr = get.value
    # Now emit the "changed" signal to reconfigure all the widgets that
    # are attached to this adjustment
    set.emit("changed")

def cb_draw_value(self, button):
    # Turn the value display on the scale widgets off or on depending
    # on the state of the checkbutton
    self.hscale.set_draw_value(button.get_active())
    self.vscale.set_draw_value(button.get_active())

# makes the sample window

def __init__(self):
    # Standard window-creating stuff
    self.window = gtk.Window (gtk.WINDOW_TOPLEVEL)
    self.window.connect("destroy", lambda w: gtk.main_quit())
    self.window.set_title("range controls")

    box1 = gtk.VBox(False, 0)
    self.window.add(box1)
    box1.show()

    box2 = gtk.HBox(False, 10)
    box2.set_border_width(10)
    box1.pack_start(box2, True, True, 0)
    box2.show()

    # value, lower, upper, step_increment, page_increment, page_size
    # Note that the page_size value only makes a difference for
    # scrollbar widgets, and the highest value you'll get is actually
    # (upper - page_size).
    adj1 = gtk.Adjustment(0.0, 0.0, 101.0, 0.1, 1.0, 1.0)

    self.vscale = gtk.VScale(adj1)
    scale_set_default_values(self.vscale)
    box2.pack_start(self.vscale, True, True, 0)
    self.vscale.show()

    box3 = gtk.VBox(False, 10)
    box2.pack_start(box3, True, True, 0)
    box3.show()

    # Reuse the same adjustment
    self.hscale = gtk.HScale(adj1)
    self.hscale.set_size_request(200, 30)
```python
scale_set_default_values(self.hscale)
box3.pack_start(self.hscale, True, True, 0)
self.hscale.show()

# Reuse the same adjustment again
scrollbar = gtk.HScrollbar(adj1)
# Notice how this causes the scales to always be updated
# continuously when the scrollbar is moved
scrollbar.set_update_policy(gtk.UPDATE_CONTINUOUS)
box3.pack_start(scrollbar, True, True, 0)
scrollbar.show()

box2 = gtk.HBox(False, 10)
box2.set_border_width(10)
box1.pack_start(box2, True, True, 0)
box2.show()

# A checkbutton to control whether the value is displayed or not
button = gtk.CheckButton("Display value on scale widgets")
button.set_active(True)
button.connect("toggled", self.cb_draw_value)
box2.pack_start(button, True, True, 0)
button.show()

box2 = gtk.HBox(False, 10)
box2.set_border_width(10)

# An option menu to change the position of the value
label = gtk.Label("Scale Value Position:"
box2.pack_start(label, False, False, 0)
label.show()

opt = gtk.OptionMenu()
menu = gtk.Menu()

item = make_menu_item ("Top", self.cb_pos_menu_select, gtk.POS_TOP)
menu.append(item)

item = make_menu_item ("Bottom", self.cb_pos_menu_select, gtk.POS_BOTTOM)
menu.append(item)

item = make_menu_item ("Left", self.cb_pos_menu_select, gtk.POS_LEFT)
menu.append(item)

item = make_menu_item ("Right", self.cb_pos_menu_select, gtk.POS_RIGHT)
menu.append(item)

opt.set_menu(menu)
box2.pack_start(opt, True, True, 0)
box2.show()

box1.pack_start(box2, True, True, 0)
box2.show()

# Yet another option menu, this time for the update policy of the
# scale widgets
label = gtk.Label("Scale Update Policy:"
box2.pack_start(label, False, False, 0)
```
label.show()

opt = gtk.OptionMenu()
menu = gtk.Menu()

item = make_menu_item("Continuous", self.cb_update_menu_select,
    gtk.UPDATE_CONTINUOUS)
menu.append(item)

item = make_menu_item("Discontinuous", self.cb_update_menu_select,
    gtk.UPDATE_DISCONTINUOUS)
menu.append(item)

item = make_menu_item("Delayed", self.cb_update_menu_select,
    gtk.UPDATE_DELAYED)
menu.append(item)

opt.set_menu(menu)
box2.pack_start(opt, True, True, 0)
opt.show()

box1.pack_start(box2, True, True, 0)
box2.show()

box2 = gtk.HBox(False, 10)
box2.set_border_width(10)

# An HScale widget for adjusting the number of digits on the
# sample scales.
label = gtk.Label("Scale Digits:")
box2.pack_start(label, False, False, 0)
label.show()

adj2 = gtk.Adjustment(1.0, 0.0, 5.0, 1.0, 1.0, 0.0)
adj2.connect("value_changed", self.cb_digits_scale)
scale = gtk.HScale(adj2)
scale.set_digits(0)
box2.pack_start(scale, True, True, 0)
scale.show()

box1.pack_start(box2, True, True, 0)
box2.show()

box2 = gtk.HBox(False, 10)
box2.set_border_width(10)

# And, one last HScale widget for adjusting the page size of the
# scrollbar.
label = gtk.Label("Scrollbar Page Size:")
box2.pack_start(label, False, False, 0)
label.show()

adj2 = gtk.Adjustment(1.0, 1.0, 101.0, 1.0, 1.0, 0.0)
adj2.connect("value_changed", self.cb_page_size, adj1)
scale = gtk.HScale(adj2)
scale.set_digits(0)
box2.pack_start(scale, True, True, 0)
scale.show()

box1.pack_start(box2, True, True, 0)
box2.show()

separator = gtk.HSeparator()
box1.pack_start(separator, False, True, 0)
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:

```python
#!/usr/bin/env python
# example label.py

import pygtk
pygtk.require('2.0')
import gtk

class Labels:
    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.connect("destroy", lambda w: gtk.main_quit())

        self.window.set_title("Label")
        vbox = gtk.VBox(False, 5)
        hbox = gtk.HBox(False, 5)
        self.window.add(hbox)
        hbox.pack_start(vbox, False, False, 0)

        frame = gtk.Frame("Normal Label")
        label = gtk.Label("This is a Normal label")
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

        frame = gtk.Frame("Multi-line Label")
        label = gtk.Label("This is a Multi-line label.
Second line
Third line")
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

        frame = gtk.Frame("Left Justified Label")
        label = gtk.Label("This is a Left-Justified Multi-line label.
Third line")
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

        frame = gtk.Frame("Right Justified Label")
        label = gtk.Label("This is a Right-Justified Multi-line label.
Fourth line, (j/k)
")
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

        frame = gtk.Frame("Line wrapped label")
        label = gtk.Label("This is an example of a line-wrapped label. It should not be taking up the entire width allocated to it, but automatically wraps the words to fit. The time has come, for all good men, to come to the aid of their party. The sixth sheik's six sheep's sick. It supports multiple paragraphs correctly, and correctly adds many extra spaces."
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

        frame = gtk.Frame("Filled, wrapped label")
        label = gtk.Label("This is an example of a line-wrapped, filled label. It should be taking up the entire width allocated to it. Here is a sentence to prove my point. Here is another sentence. Here comes the sun, do de do de do. This is a new paragraph.
This is another newer, longer, better paragraph. It is coming to an end, unfortunately."
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

        frame = gtk.Frame("Underlined label")
        label = gtk.Label("This label is underlined!
This one is underlined in quite a funky fashion")
        frame.add(label)
        vbox.pack_start(frame, False, False, 0)

self.window.set_border_width(5)

self.window.show_all()

def __init__(self):
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.connect("destroy", lambda w: gtk.main_quit())
    self.window.set_title("Label")
    vbox = gtk.VBox(False, 5)
    hbox = gtk.HBox(False, 5)
    self.window.add(hbox)
    hbox.pack_start(vbox, False, False, 0)

    frame = gtk.Frame("Normal Label")
    label = gtk.Label("This is a Normal label")
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

    frame = gtk.Frame("Multi-line Label")
    label = gtk.Label("This is a Multi-line label.
Second line
Third line")
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

    frame = gtk.Frame("Left Justified Label")
    label = gtk.Label("This is a Left-Justified Multi-line label.
Third line")
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

    frame = gtk.Frame("Right Justified Label")
    label = gtk.Label("This is a Right-Justified Multi-line label.
Fourth line, (j/k)
")
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

    frame = gtk.Frame("Line wrapped label")
    label = gtk.Label("This is an example of a line-wrapped label. It should not be taking up the entire width allocated to it, but automatically wraps the words to fit. The time has come, for all good men, to come to the aid of their party. The sixth sheik's six sheep's sick. It supports multiple paragraphs correctly, and correctly adds many extra spaces."
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

    frame = gtk.Frame("Filled, wrapped label")
    label = gtk.Label("This is an example of a line-wrapped, filled label. It should be taking up the entire width allocated to it. Here is a sentence to prove my point. Here is another sentence. Here comes the sun, do de do de do. This is a new paragraph.
This is another newer, longer, better paragraph. It is coming to an end, unfortunately."
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

    frame = gtk.Frame("Underlined label")
    label = gtk.Label("This label is underlined!
This one is underlined in quite a funky fashion")
    frame.add(label)
    vbox.pack_start(frame, False, False, 0)

self.window.set_border_width(5)

self.window.show_all()
```
frame.add(label)
vbox.pack_start(frame, False, False, 0)

frame = gtk.Frame("Left Justified Label")
label = gtk.Label("This is a Left-Justified Multi-line label.\nThird line")
label.set_justify(gtk.JUSTIFY_LEFT)
frame.add(label)
vbox.pack_start(frame, False, False, 0)

frame = gtk.Frame("Right Justified Label")
label = gtk.Label("This is a Right-Justified\nMulti-line label.\nFourth line, (j/k)")
label.set_justify(gtk.JUSTIFY_RIGHT)
frame.add(label)
vbox.pack_start(frame, False, False, 0)

vbox = gtk.VBox(False, 5)
hbox.pack_start(vbox, False, False, 0)
frame = gtk.Frame("Line wrapped label")
label = gtk.Label("This is an example of a line-wrapped label. It should not be taking up the entire width allocated to it, but automatically wraps the words to fit. The time has come, for all good men, to come to the aid of their party. The sixth sheik’s six sheep’s sick. It supports multiple paragraphs correctly, and correctly adds many extra spaces.")
label.set_line_wrap(True)
frame.add(label)
vbox.pack_start(frame, False, False, 0)

frame = gtk.Frame("Filled, wrapped label")
label = gtk.Label("This is an example of a line-wrapped, filled label. It should be taking up the entire width allocated to it."
"Here is a sentence to prove "my point. Here is another sentence. "Here comes the sun, do de do do do."
"This is a new paragraph."
"This is another newer, longer, better paragraph. It is coming to an end,"
"unfortunately."")
label.set_justify(gtk.JUSTIFY_FILL)
label.set_line_wrap(True)
frame.add(label)
vbox.pack_start(frame, False, False, 0)

frame = gtk.Frame("Underlined label")
label = gtk.Label("This label is underlined!\nThis one is underlined in quite a funky fashion")
label.set_justify(gtk.JUSTIFY_LEFT)
label.set_pattern("_________________________ _ _________ _ ______ __ _______ ___")
CHAPTER 9. MISCELLANEOUS WIDGETS

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:

```python
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
- SHADOWETCHED_IN
- SHADOWETCHED_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:

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

# example arrow.py

import pygtk
pygtk.require('2.0')
import gtk
```

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.

```python
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:

```python
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:

---

The source code for `tooltip.py` is:

```python
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         window.set_title("Tooltips")
24
25         # It’s a good idea to do this for all windows.
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 box to hold the arrows/buttons
32         box = gtk.HBox(False, 0)
33         box.set_border_width(2)
34```
CHAPTER 9. MISCELLANEOUS WIDGETS

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 let's 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:

```python
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.

```python
progressbar.pulse()
```

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

```python
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:

```python
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:

```python
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:
The source code for `progressbar.py` is:

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

# example progressbar.py

import pygtk
pygtk.require('2.0')
import gtk, gobject

# Update the value of the progress bar so that we get
# some movement
def progress_timeout(pbobj):
    if pbobj.activity_check.get_active():
        pbobj.pbar.pulse()
    else:
        # Calculate the value of the progress bar using the
        # value range set in the adjustment object
        new_val = pbobj.pbar.get_fraction() + 0.01
        if new_val > 1.0:
            new_val = 0.0
        # Set the new value
        pbobj.pbar.set_fraction(new_val)
    # As this is a timeout function, return TRUE so that it
    # continues to get called
    return True

class ProgressBar:
    # Callback that toggles the text display within the progress
    # bar trough
    def toggle_show_text(self, widget, data=None):
        if widget.get_active():
            self.pbar.set_text("some text")
        else:
            self.pbar.set_text("")

    # Callback that toggles the activity mode of the progress
    # bar
    def toggle_activity_mode(self, widget, data=None):
        if widget.get_active():
            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, 2,
104                  gtk.EXPAND | gtk.FILL, gtk.EXPAND | gtk.FILL,
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:
CHAPTER 9. MISCELLANEOUS WIDGETS

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:

```python
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:

```python
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.

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

Here’s an example program, message.py

```python
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:

```python
message.show()
gtk.main()
```
image = gtk.Image()

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

```python
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**

![Example Images in Buttons](images.png)

The source code is:

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

# example images.py

import pygtk
pygtk.require('2.0')
import gtk

class ImagesExample:
    def close_application(self, widget, event, data=None):
        gtk.main_quit()
        return False

    def button_clicked(self, widget, data=None):
        # is invoked when the button is clicked. It just prints a message.
```

69
print "button %s clicked" % data

def __init__(self):
    # create the main window, and attach delete_event signal to terminating
    # the application
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.connect("delete_event", self.close_application)
    window.set_border_width(10)
    window.show()

    # a horizontal box to hold the buttons
    hbox = gtk.HBox()
    hbox.show()
    window.add(hbox)

    pixbufanim = gtk.gdk.PixbufAnimation("goalie.gif")
    image = gtk.Image()
    image.set_from_animation(pixbufanim)
    image.show()
    # a button to contain the image widget
    button = gtk.Button()
    button.add(image)
    button.show()
    hbox.pack_start(button)
    button.connect("clicked", self.button_clicked, "1")

    # create several images with data from files and load images into buttons
    image = gtk.Image()
    image.set_from_file("apple-red.png")
    image.show()
    # a button to contain the image widget
    button = gtk.Button()
    button.add(image)
    button.show()
    hbox.pack_start(button)
    button.connect("clicked", self.button_clicked, "2")

    image = gtk.Image()
    image.set_from_file("chaos.jpg")
    image.show()
    # a button to contain the image widget
    button = gtk.Button()
    button.add(image)
    button.show()
    hbox.pack_start(button)
    button.connect("clicked", self.button_clicked, "3")

    image = gtk.Image()
    image.set_from_file("important.tif")
    image.show()
    # a button to contain the image widget
    button = gtk.Button()
    button.add(image)
    button.show()
    hbox.pack_start(button)
    button.connect("clicked", self.button_clicked, "4")

    image = gtk.Image()
    image.set_from_file("soccerball.gif")
    image.show()
    # a button to contain the image widget
    button = gtk.Button()
CHAPTER 9. MISCELLANEOUS WIDGETS 9.6. IMAGES

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

```python
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:

```python
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:

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

```python
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:

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

# example pixmap.py

import pygtk
pygtk.require('2.0')
import gtk

# XPM data of Open-File icon
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. ",
    " .XXXXXXX. ",
    " ",
    " ",
    " 
]
```

The source code is:

```python
#!
```
class PixmapExample:
    # when invoked (via signal delete_event), terminates the application.
    def close_application(self, widget, event, data=None):
        gtk.main_quit()
        return False

    # is invoked when the button is clicked. It just prints a message.
    def button_clicked(self, widget, data=None):
        print "button clicked"

    def __init__(self):
        # create the main window, and attach delete_event signal to terminating
        # the application
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.connect("delete_event", self.close_application)
        window.set_border_width(10)
        window.show()

        # now for the pixmap from XPM data
        pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(window.window, None, xpm_data)

        # an image widget to contain the pixmap
        image = gtk.Image()
        image.set_from_pixmap(pixmap, mask)
        image.show()

        # a button to contain the image widget
        button = gtk.Button()
        button.add(image)
        window.add(button)
        button.show()

        button.connect("clicked", self.button_clicked)

    def main():
        gtk.main()
        return 0

    if __name__ == "__main__":
        PixmapExample()
        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:

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

# example wheelbarrow.py

import pygtk
pygtk.require('2.0')
import gtk

# XPM
WheelbarrowFull_xpm = [
    "48 48 64 1",
    " c None",
    ". c #DF7DCF3CC71B",
    "X c #965875D669A6",
    "+ c #965892489658",
    "@ c #8E38410330C2",
    "; c #C71B30C230C2",
    ": c #C71B9A699658",
    "> c #618561856185",
    ", c #20811C712081",
    "1 c #861720812081",
    "2 c #DF7D4D34103",
    "3 c #79E769A671C6",
    "4 c #861782078617",
    "5 c #41033CF34103",
    "6 c #000000000000",
    "7 c #49241C711040",
    "8 c #492445144924",
    "9 c #082008200820",
    "0 c #69A618611861",
    "q c #B6DA71C65144",
    "w c #410330C238E3",
    "e c #CF3CBAEAB6DA",
    "r c #71C6451430C2",
]
```
class WheelbarrowExample:
    # When invoked (via signal delete_event), terminates the application
    def close_application(self, widget, event, data=None):
        gtk.main_quit()
        return False
    
    def __init__(self):
        # Create the main window, and attach delete_event signal to terminate
        # the application. Note that the main window will not have a titlebar.
        # since we’re making it a popup.
        window = gtk.Window(gtk.WINDOW_POPUP)
        window.connect("delete_event", self.close_application)
        window.set_events(window.get_events() | gtk.gdk.BUTTON_PRESS_MASK)
        window.connect("button_press_event", self.close_application)
        window.show()

        # Now for the pixmap and the image widget
        pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(
            window.window, None, WheelbarrowFull_xpm)
        image = gtk.Image()
        image.set_from_pixmap(pixmap, mask)
        image.show()

        # To display the image, we use a fixed widget to place the image
        fixed = gtk.Fixed()
        fixed.set_size_request(200, 200)
        fixed.put(image, 0, 0)
        window.add(fixed)
        fixed.show()

        # This masks out everything except for the image itself
        window.shape_combine_mask(mask, 0, 0)

        # show the window
        window.set_position(gtk.WIN_POS_CENTER_ALWAYS)
        window.show()

    def main():
        gtk.main()
        return 0

    if __name__ == "__main__":
        WheelbarrowExample()
        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:

```python
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:

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

The default measure is PIXELS.

```python
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:

```python
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:

```python
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:

```python
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":

```python
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:

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

# example rulers.py

import pygtk
pygtk.require('2.0')
import gtk

class RulersExample:
    XSIZE = 400
    YSIZE = 400

    # This routine gets control when the close button is clicked
    def close_application(self, widget, event, data=None):
        gtk.main_quit()
        return False
```

78
```python
def __init__(self):
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.connect("delete_event", self.close_application)
    window.set_border_width(10)

    # Create a table for placing the ruler and the drawing area
    table = gtk.Table(3, 2, False)
    window.add(table)

    area = gtk.DrawingArea()
    area.set_size_request(self.XSIZE, self.YSIZE)
    table.attach(area, 1, 2, 1, 2,
                  gtk.EXPAND|gtk.FILL, gtk.FILL, 0, 0)
    area.set_events(gtk.gdk.POINTER_MOTION_MASK |
                    gtk.gdk.POINTER_MOTION_HINT_MASK)

    # The horizontal ruler goes on top. As the mouse moves across the
    # drawing area, a motion_notify_event is passed to the
    # appropriate event handler for the ruler.
    hrule = gtk.HRuler()
    hrule.set_metric(gtk.PIXELS)
    hrule.set_range(7, 13, 0, 20)
    def motion_notify(ruler, event):
        return ruler.emit("motion_notify_event", event)
    area.connect_object("motion_notify_event", motion_notify, hrule)
    table.attach(hrule, 1, 2, 0, 1,
                 gtk.EXPAND|gtk.SHRINK|gtk.FILL, gtk.FILL, 0, 0)

    # The vertical ruler goes on the left. As the mouse moves across
    # the drawing area, a motion_notify_event is passed to the
    # appropriate event handler for the ruler.
    vrule = gtk.VRuler()
    vrule.set_metric(gtk.PIXELS)
    vrule.set_range(0, self.YSIZE, 10, self.YSIZE)
    area.connect_object("motion_notify_event", motion_notify, vrule)
    table.attach(vrule, 0, 1, 1, 2,
                 gtk.FILL, gtk.EXPAND|gtk.SHRINK|gtk.FILL, 0, 0)

    # Now show everything
    area.show()
    hrule.show()
    vrule.show()
    table.show()
    window.show()

def main():
    gtk.main()
    return 0

if __name__ == "__main__":
    RulersExample()
    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:

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

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

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

There are three additional methods that operate on statusbars:

```python
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**

![Statusbar Example](image)

The `statusbar.py` source code is:

```python
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.statusbar.push(data, buff)
14         return
15
16     def pop_item(self, widget, data):
17         self.statusbar.pop(data)
18         return
19
20     def __init__(self):
21         self.count = 1
22         # create a new window
```
CHAPTER 9. MISCELLANEOUS WIDGETS

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:

```python
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:

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

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

```python
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:

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

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

```python`
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.

```python
    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.

```python
    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.

```python
    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.

```python
    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:

```python
    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):
```
CHAPTER 9. MISCELLANEOUS WIDGETS

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:

```python
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>initial value for the Spin Button</td>
</tr>
<tr>
<td>lower</td>
<td>lower range value</td>
</tr>
<tr>
<td>upper</td>
<td>upper range value</td>
</tr>
<tr>
<td>step_increment</td>
<td>value to increment/decrement when pressing mouse button-1 on a button</td>
</tr>
<tr>
<td>page_increment</td>
<td>value to increment/decrement when pressing mouse button-2 on a button</td>
</tr>
<tr>
<td>page_size</td>
<td>unused</td>
</tr>
</tbody>
</table>

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

```python
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:

```python
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:

```python
spin_button.set_adjustment(adjustment)
adjustment = spin_button.get_adjustment()
```

The number of decimal places can also be altered using:

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

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

```python
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:

```python
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:

```python
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:
CHAPTER 9. MISCELLANEOUS WIDGETS

9.10. SPIN BUTTONS

Figure 9.11 Spin Button Example

The spinbutton.py source code is:

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

# example spinbutton.py

import pygtk
pygtk.require('2.0')
import gtk

class SpinButtonExample:
    def toggle_snap(self, widget, spin):
        spin.set_snap_to_ticks(widget.get_active())
    
    def toggle_numeric(self, widget, spin):
        spin.set_numeric(widget.get_active())
    
    def change_digits(self, widget, spin, spin1):
        spin1.set_digits(spin.get_value_as_int())
    
    def get_value(self, widget, data, spin, spin2, label):
        if data == 1:
            buf = "%d" % spin.get_value_as_int()
        else:
            buf = "%0.*f" % (spin2.get_value_as_int(),
                              spin.get_value())
        label.set_text(buf)
    
    def __init__(self):
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.connect("destroy", lambda w: gtk.main_quit())
        window.set_title("Spin Button")
```

86
main_vbox = gtk.VBox(False, 5)
main_vbox.set_border_width(10)
window.add(main_vbox)
frame = gtk.Frame("Not accelerated")
main_vbox.pack_start(frame, True, True, 0)
vbox = gtk.VBox(False, 0)
vbox.set_border_width(5)
frame.add(vbox)

# Day, month, year spinners
hbox = gtk.HBox(False, 0)
vbox.pack_start(hbox, True, True, 5)

vbox2 = gtk.VBox(False, 0)
hbox.pack_start(vbox2, True, True, 5)
label = gtk.Label("Day :")
label.set_alignment(0, 0.5)
vbox2.pack_start(label, False, True, 0)
adj = gtk.Adjustment(1.0, 1.0, 31.0, 1.0, 5.0, 0.0)
spinner = gtk.SpinButton(adj, 0, 0)
spinner.set_wrap(True)
vbox2.pack_start(spinner, False, True, 0)

vbox2 = gtk.VBox(False, 0)
hbox.pack_start(vbox2, True, True, 5)
label = gtk.Label("Month :")
label.set_alignment(0, 0.5)
vbox2.pack_start(label, False, True, 0)
adj = gtk.Adjustment(1.0, 1.0, 12.0, 1.0, 5.0, 0.0)
spinner = gtk.SpinButton(adj, 0, 0)
spinner.set_wrap(True)
vbox2.pack_start(spinner, False, True, 0)

vbox2 = gtk.VBox(False, 0)
hbox.pack_start(vbox2, True, True, 5)
label = gtk.Label("Year :")
label.set_alignment(0, 0.5)
vbox2.pack_start(label, False, True, 0)
adj = gtk.Adjustment(1998.0, 0.0, 2100.0, 1.0, 100.0, 0.0)
spinner = gtk.SpinButton(adj, 0, 0)
spinner.set_wrap(False)
spinner.set_size_request(55, -1)
vbox2.pack_start(spinner, False, True, 0)

frame = gtk.Frame("Accelerated")
main_vbox.pack_start(frame, True, True, 0)
vbox = gtk.VBox(False, 0)
vbox.set_border_width(5)
frame.add(vbox)

hbox = gtk.HBox(False, 0)
vbox.pack_start(hbox, True, True, 5)
vbox2 = gtk.VBox(False, 0)
hbox.pack_start(vbox2, True, True, 5)
CHAPTER 9. MISCELLANEOUS WIDGETS

9.10. SPIN BUTTONS

```python
label = gtk.Label("Value :")
lable.set_alignment(0, 0.5)
vbox2.pack_start(label, False, True, 0)
adj = gtk.Adjustment(0.0, -10000.0, 10000.0, 0.5, 100.0, 0.0)
spinner1 = gtk.SpinButton(adj, 1.0, 2)
spinner1.set_wrap(True)
spinner1.set_size_request(100, -1)
vbox2.pack_start(spinner1, False, True, 0)

vbox2 = gtk.VBox(False, 0)
hbox.pack_start(vbox2, True, True, 5)
label = gtk.Label("Digits :")
lable.set_alignment(0, 0.5)
vbox2.pack_start(label, False, True, 0)
adj = gtk.Adjustment(2, 1, 5, 1, 1, 0)
spinner2 = gtk.SpinButton(adj, 0.0, 0)
spinner2.set_wrap(True)
adj.connect("value_changed", self.change_digits, spinner2, spinner1)
vbox2.pack_start(spinner2, False, True, 0)
hbox = gtk.HBox(False, 0)
vbox.pack_start(hbox, False, True, 5)
button = gtk.CheckButton("Snap to 0.5-ticks")
button.connect("clicked", self.toggle_snap, spinner1)
button.set_active(True)

button = gtk.CheckButton("Numeric only input mode")
button.connect("clicked", self.toggle_numeric, spinner1)
button.set_active(True)
val_label = gtk.Label("")

hbox = gtk.HBox(False, 0)
vbox.pack_start(hbox, False, True, 5)
button = gtk.Button("Value as Int")
button.connect("clicked", self.get_value, 1, spinner1, spinner2, val_label)

button = gtk.Button("Value as Float")
button.connect("clicked", self.get_value, 2, spinner1, spinner2, val_label)
vbox.pack_start(val_label, True, True, 0)
val_label.set_text("0")

hbox = gtk.HBox(False, 0)
mained_vbox.pack_start(hbox, False, True, 0)
button = gtk.Button("Close")
button.connect("clicked", lambda w: gtk.main_quit())
hbox.pack_start(button, True, True, 5)
window.show_all()

def main():
```

88
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:

```python
combo.entry
combo.list
```

First off, to create a Combo, use:

```python
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:

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

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

```python
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:

```python
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:

```python
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:
```python
combo.entry.connect("activate", my_callback, my_data)
```
Getting the text at any arbitrary time is accomplished by simply using the entry method:
```python
string = combo.entry.get_text()
```
That's about all there is to it. There is a method:
```python
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:
```python
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.
```python
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:
```python
calendar.display_options(flags)
```
The `flags` argument can be formed by combining either of the following five options using the
logical bitwise OR (|) operation:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALENDAR_SHOW_HEADING</td>
<td>this option specifies that the month and year should be shown when drawing the calendar.</td>
</tr>
<tr>
<td>CALENDAR_SHOW_DAY_NAMES</td>
<td>this option specifies that the three letter descriptions should be displayed for each day (e.g. Mon, Tue, etc.).</td>
</tr>
</tbody>
</table>
The following methods are used to set the currently displayed date:

```python
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:

```python
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.

```python
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:
The calendar.py source code is:

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

# example calendar.py
#
# Copyright (C) 1998 Cesar Miquel, Shawn T. Amundson, Mattias Gronlund
# Copyright (C) 2000 Tony Gale
# Copyright (C) 2001-2004 John Finlay
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

import pygtk
pygtk.require('2.0')
import gtk, pango
import time
```
class CalendarExample:
    DEF_PAD = 10
    DEF_PAD_SMALL = 5
    TM_YEAR_BASE = 1900

    calendar_show_header = 0
    calendar_show_days = 1
    calendar_month_change = 2
    calendar_show_week = 3

    def calendar_date_to_string(self):
        year, month, day = self.window.get_date()
        mytime = time.mktime((year, month+1, day, 0, 0, 0, 0, 0, -1))
        return time.strftime("%x", time.localtime(mytime))

    def calendar_set_signal_strings(self, sig_str):
        prev_sig = self.prev_sig.get()
        self.prev2_sig.set_text(prev_sig)
        prev_sig = self.last_sig.get()
        self.prev_sig.set_text(prev_sig)
        self.last_sig.set_text(sig_str)

    def calendar_month_changed(self, widget):
        buffer = "month_changed: %s" % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

    def calendar_day_selected(self, widget):
        buffer = "day_selected: %s" % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

    def calendar_day_selected_double_click(self, widget):
        buffer = "day_selected_double_click: %s"
        buffer = buffer % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

        year, month, day = self.window.get_date()

        if self.marked_date[day-1] == 0:
            self.window.mark_day(day)
            self.marked_date[day-1] = 1
        else:
            self.window.unmark_day(day)
            self.marked_date[day-1] = 0

    def calendar_prev_month(self, widget):
        buffer = "prev_month: %s" % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

    def calendar_next_month(self, widget):
        buffer = "next_month: %s" % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

    def calendar_prev_year(self, widget):
        buffer = "prev_year: %s" % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

    def calendar_next_year(self, widget):
        buffer = "next_year: %s" % self.calendar_date_to_string()
        self.calendar_set_signal_strings(buffer)

    def calendar_set_flags(self):
        options = 0
        for i in range(5):
if self.settings[i]:
    options = options + (1<<i)
if self.window:
    self.window.display_options(options)

def calendar_toggle_flag(self, toggle):
    j = 0
    for i in range(5):
        if self.flag_checkboxes[i] == toggle:
            j = i
    self.settings[j] = not self.settings[j]
    self.calendar_set_flags()

def calendar_font_selection_ok(self, button):
    self.font = self.font_dialog.get_font_name()
    if self.window:
        font_desc = pango.FontDescription(self.font)
        if font_desc:
            self.window.modify_font(font_desc)

def calendar_select_font(self, button):
    if not self.font_dialog:
        window = gtk.FontSelectionDialog("Font Selection Dialog")
        self.font_dialog = window
    window.set_position(gtk.WIN_POS_MOUSE)
    window.connect("destroy", self.font_dialog_destroyed)
    window.ok_button.connect("clicked",
                              self.calendar_font_selection_ok)
    window.cancel_button.connect_object("clicked",
                                        lambda wid: wid.destroy(),
                                        self.font_dialog)
    window = self.font_dialog
    if not (window.flags() & gtk.VISIBLE):
        window.show()
    else:
        window.destroy()
        self.font_dialog = None

def font_dialog_destroyed(self, data=None):
    self.font_dialog = None

__init__ (self):
flags = [
    "Show Heading",
    "Show Day Names",
    "No Month Change",
    "Show Week Numbers",
]
sel.window = None
self.font = None
self.font_dialog = None
self.flag_checkboxes = 5*[None]
sel.settings = 5*[0]
sel.marked_date = 31*[0]
window = gtk.Window(gtk.WINDOW_Toplevel)
window.set_title("Calendar Example")
window.set_border_width(5)
window.connect("destroy", lambda x: gtk.main_quit())
window.set_resizable(False)

vbox = gtk.VBox(False, self.DEF_PAD)
window.add(vbox)

# The top part of the window, Calendar, flags and fontsel.
hbox = gtk.HBox(False, self.DEF_PAD)
vbox.pack_start(hbox, True, True, self.DEF_PAD)
hbbox = gtk.HButtonBox()
hbbox.pack_start(hbbox, False, False, self.DEF_PAD)
hbbox.set_layout(gtk.BUTTONBOX_SPREAD)
hbbox.set_spacing(5)

# Calendar widget
frame = gtk.Frame("Calendar")
hbbox.pack_start(frame, False, True, self.DEF_PAD)
calendar = gtk.Calendar()
self.window = calendar
self.calendar_set_flags()
calendar.mark_day(19)
self.marked_date[19-1] = 1
frame.add(calendar)
calendar.connect("month_changed", self.calendar_month_changed)
calendar.connect("day_selected", self.calendar_day_selected)
calendar.connect("day_selected_double_click",
                 self.calendar_day_selected_double_click)
calendar.connect("prev_month", self.calendar_prev_month)
calendar.connect("next_month", self.calendar_next_month)
calendar.connect("prev_year", self.calendar_prev_year)
calendar.connect("next_year", self.calendar_next_year)

separator = gtk.VSeparator()
hbbox.pack_start(separator, False, True, 0)

vbox2 = gtk.VBox(False, self.DEF_PAD)
hbbox.pack_start(vbox2, False, False, self.DEF_PAD)

# Build the Right frame with the flags in
frame = gtk.Frame("Flags")
vbox2.pack_start(frame, True, True, self.DEF_PAD)
vbox3 = gtk.VBox(True, self.DEF_PAD_SMALL)
frame.add(vbox3)

for i in range(len(flags)):
toggle = gtk.CheckButton(flags[i])
toggle.connect("toggled", self.calendar_toggle_flag)
vbox3.pack_start(toggle, True, True, 0)
self.flag_checkboxes[i] = toggle

# Build the right font-button
button = gtk.Button("Font...")
button.connect("clicked", self.calendar_select_font)
vbox2.pack_start(button, False, False, 0)

# Build the Signal-event part.
frame = gtk.Frame("Signal events")
vbox.pack_start(frame, True, True, self.DEF_PAD)
vbox2 = gtk.VBox(True, self.DEF_PAD_SMALL)
frame.add(vbox2)

hbbox = gtk.HBox (False, 3)
vbox2.pack_start(hbbox, False, True, 0)
label = gtk.Label("Signal:"")
CHAPTER 9. MISCELLANEOUS WIDGETS

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 when-
ever the current color in the widget changes, either when the user changes it or if it’s set explicitly
through the set_color() method.

Let’s have a look at what the color selection widget has to offer us. The widget comes in two flavors:

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.

colorseeldig = 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:

```python
colorsel = colorseldlg.colorsel
```

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

```python
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.

```python
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 `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 65536 (fully opaque).

```python
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:
The source code for `colorsel.py` is:

```python
#!/usr/bin/env python
# example colorsel.py
import pygtk
pygtk.require('2.0')
import gtk

class ColorSelectionExample:
    # Color changed handler
    def color_changed_cb(self, widget):
        # Get drawingarea colormap
        colormap = self.drawingarea.get_colormap()

        # Get current color
        color = self.colorseldlg.colorsel.get_current_color()

        # Set window background color
        self.drawingarea.modify_bg(gtk.STATE_NORMAL, color)

    # Drawingarea event handler
    def area_event(self, widget, event):
        handled = False

        # Check if we’ve received a button pressed event
```
if event.type == gtk.gdk.BUTTON_PRESS:
    handled = True

    # Create color selection dialog
    if self.colorseldlg == None:
        self.colorseldlg = gtk.ColorSelectionDialog(
            "Select background color")

    # Get the ColorSelection widget
    colorsel = self.colorseldlg.colorsel

    colorsel.set_previous_color(self.color)
    colorsel.set_current_color(self.color)
    colorsel.set_has_palette(True)

    # Connect to the "color_changed" signal
    colorsel.connect("color_changed", self.color_changed_cb)

    # Show the dialog
    response = self.colorseldlg.run()

    if response == gtk.RESPONSE_OK:
        self.color = colorsel.get_current_color()
    else:
        self.drawingarea.modify_bg(gtk.STATE_NORMAL, self.color)

    self.colorseldlg.hide()

return handled

# Close down and exit handler
def destroy_window(self, widget, event):
    gtk.main_quit()
    return True

def __init__(self):
    self.colorseldlg = None

    # Create toplevel window, set title and policies
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.set_title("Color selection test")
    window.set_resizable(True)

    # Attach to the "delete" and "destroy" events so we can exit
    window.connect("delete_event", self.destroy_window)

    # Create drawingarea, set size and catch button events
    self.drawingarea = gtk.DrawingArea()

    self.color = self.drawingarea.get_colormap().alloc_color(0, 65535, 0)

    self.drawingarea.set_size_request(200, 200)
    self.drawingarea.set_events(gtk.gdk.BUTTON_PRESS_MASK)
    self.drawingarea.connect("event", self.area_event)

    # Add drawingarea to window, then show them both
    window.add(self.drawingarea)
    self.drawingarea.show()
    window.show()

def main():
    gtk.main()
    return 0

if __name__ == "__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:

```python
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:

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

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

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

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

```python
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:
The source code for filesel.py is:

```python
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.filew.ok_sel)
24
25         # Connect the cancel_button to destroy the widget
```
CHAPTER 9. MISCELLANEOUS WIDGETS

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:

```python
defontseldlg = 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. The `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:

```python
defontseldlg.set_font_name(fontname)
```

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

```python
defontseldlg.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:

```python
defont_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:

```python
defontseldlg.set_preview_text(text)
```

The preview text can be retrieved with the method:

```python
defont_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:

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

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

```python
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**

![Event Box Example](image)

The source code for `eventbox.py` is:

```python
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")
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:

```python
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`:

```python
#!/usr/bin/env python
import pygtk

import fixed.py
```
class FixedExample:
    # This callback method moves the button to a new position
    # in the Fixed container.
    def move_button(self, widget):
        self.x = (self.x+30)%300
        self.y = (self.y+50)%300
        self.fixed.move(widget, self.x, self.y)

    def __init__(self):
        self.x = 50
        self.y = 50

        # Create a new window
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_title("Fixed Container")

        # Here we connect the "destroy" event to a signal handler
        window.connect("destroy", lambda w: gtk.main_quit())

        # Sets the border width of the window.
        window.set_border_width(10)

        # Create a Fixed Container
        self.fixed = gtk.Fixed()
        window.add(self.fixed)
        self.fixed.show()

        for i in range(1, 4):
            # Creates a new button with the label "Press me"
            button = gtk.Button("Press me")

            # When the button receives the "clicked" signal, it will call the
            # method move_button().
            button.connect("clicked", self.move_button)

            # This packs the button into the fixed containers window.
            self.fixed.put(button, i*50, i*50)

        # The final step is to display this newly created widget.
        button.show()

        # Display the window
        window.show()

    def main():
        # Enter the event loop
        gtk.main()
        return 0

    if __name__ == "__main__":
        FixedExample()
        main()
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:

```python
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:

```python
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:

```python
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:

```python
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:

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

# example layout.py

import pygtk
pygtk.require('2.0')
import gtk
import random

class LayoutExample:
    def WindowDeleteEvent(self, widget, event):
        # return false so that window will be destroyed
        return False

    def WindowDestroy(self, widget, *data):
        # exit main loop
        gtk.main_quit()

    def ButtonClicked(self, button):
        # move the button
        self.layout.move(button, random.randint(0,500),
                         random.randint(0,500))

    def __init__(self):
        # create the top level window
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_title("Layout Example")
        window.set_default_size(300, 300)
        window.connect("delete-event", self.WindowDeleteEvent)
```

110
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.

```python
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:

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

# example frame.py

import pygtk
pygtk.require('2.0')
import gtk

class FrameExample:
    def __init__(self):
        # Create a new window
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_title("Frame Example")

        # Here we connect the "destroy" event to a signal handler
```

112
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:
The source code for `aspectframe.py` is:

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

# example aspectframe.py

import pygtk
pygtk.require('2.0')
import gtk

class AspectFrameExample:
    def __init__(self):
        window = gtk.Window(gtk.WINDOW_TOPLEVEL);
        window.set_title("Aspect Frame")
        window.connect("destroy", lambda x: gtk.main_quit())
        window.set_border_width(10)

        # Create an aspect_frame and add it to our toplevel window
        aspect_frame = gtk.AspectFrame("2x1", # label
                                      0.5, # center x
                                      0.5, # center y
                                      2, # xsize/ysize = 2
                                      False) # ignore child's aspect

        window.add(aspect_frame)
        aspect_frame.show()

        # Now add a child widget to the aspect frame
        drawing_area = gtk.DrawingArea()

        # Ask for a 200x200 window, but the AspectFrame will give us a 200 x 100
        # window since we are forcing a 2x1 aspect ratio
        drawing_area.set_size_request(200, 200)
        aspect_frame.add(drawing_area)
        drawing_area.show()
        window.show()
```

Figure 10.5 Aspect Frame Example
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:

```python
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:

```python
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:

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

# example paned.py

import pygtk
pygtk.require('2.0')
import gtk, gobject

class PanedExample:
    # Create the list of "messages"
    def create_list(self):
        # Create a new scrolled window, with scrollbars only if needed
        scrolled_window = gtk.ScrolledWindow()
        scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)

        model = gtk.ListStore(gobject.TYPE_STRING)
        tree_view = gtk.TreeView(model)
        scrolled_window.add_withViewport(tree_view)

        tree_view.show()
```

# Add some messages to the window
for i in range(10):
    msg = "Message #%d" % i
    iter = model.append()
    model.set(iter, 0, msg)

cell = gtk.CellRendererText()
column = gtk.TreeViewColumn("Messages", cell, text=0)
tree_view.append_column(column)

return scrolled_window

# Add some text to our text widget - this is a callback that is invoked
# when our window is realized. We could also force our window to be
# realized with gtk.Widget.realize(), but it would have to be part of a
# hierarchy first
def insert_text(self, buffer):
    iter = buffer.get_iter_at_offset(0)
    buffer.insert(iter,
                  "From: pathfinder@nasa.gov\n"
                  "To: mom@nasa.gov\n"
                  "Subject: Made it!\n"
                  "\n"
                  "We just got in this morning. The weather has been\n"
                  "great - clear but cold, and there are lots of fun sights.\n"
                  "Sojourner says hi. See you soon.\"
                  " -Path\n")

# Create a scrolled text area that displays a "message"
def create_text(self):
    view = gtk.TextView()
    buffer = view.get_buffer()
    scrolled_window = gtk.ScrolledWindow()
    scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
    scrolled_window.add(view)
    self.insert_text(buffer)
    scrolled_window.show_all()
    return scrolled_window

def __init__(self):
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.set_title("Paned Windows")
    window.connect("destroy", lambda w: gtk.main_quit())
    window.set_border_width(10)
    window.set_size_request(450, 400)

    # create a vpaned widget and add it to our toplevel window
    vpaned = gtk.VPaned()
    window.add(vpaned)
    vpaned.show()

    # Now create the contents of the two halves of the window
    list = self.create_list()
    vpaned.add1(list)
    list.show()

    text = self.create_text()
    vpaned.add2(text)
    text.show()

    window.show()

def main():

117
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:

```python
group = 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:

```python
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:

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

Possible values for the type parameter are:

- SHADOW_NONE
- SHADOW_IN
- SHADOW_OUT
- SHADOWETCHED_IN
- SHADOWETCHED_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.

```python
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.

```python
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.
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:

![Scrolled Window Example](image)

The source code for the `scrolledwin.py` program is:

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

# example scrolledwin.py

import pygtk
pygtk.require('2.0')
import gtk

class ScrolledWindowExample:
    def destroy(self, widget):
        gtk.main_quit()
    
    def __init__(self):
        # Create a new dialog window for the scrolled window to be
        # packed into.
        window = gtk.Dialog()
        window.connect("destroy", self.destroy)
        window.set_title("ScrolledWindow example")
        window.set_border_width(0)
        window.set_size_request(300, 300)

        # create a new scrolled window.
        scrolled_window = gtk.ScrolledWindow()
```

...
scrolled_window.set_border_width(10)

# the policy is 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. The first one is the horizontal scrollbar, the second, the vertical.
scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_ALWAYS)

# The dialog window is created with a vbox packed into it.
window.vbox.pack_start(scrolled_window, True, True, 0)
scrolled_window.show()

# create a table of 10 by 10 squares.
table = gtk.Table(10, 10, False)

# set the spacing to 10 on x and 10 on y
table.set_row_spacings(10)
table.set_col_spacings(10)

# pack the table into the scrolled window
scrolled_window.add_with_viewport(table)
table.show()

# this simply creates a grid of toggle buttons on the table
# to demonstrate the scrolled window.
for i in range(10):
    for j in range(10):
        buffer = "button (%d,%d)" % (i, j)
        button = gtk.ToggleButton(buffer)
        table.attach(button, i, i+1, j, j+1)
        button.show()

# Add a "close" button to the bottom of the dialog
button = gtk.Button("close")
button.connect_object("clicked", self.destroy, window)

# this makes it so the button is the default.
button.set_flags(gtk.CAN_DEFAULT)
window.action_area.pack_start(button, True, True, 0)

# This grabs this button to be the default button. Simply hitting the "Enter" key will cause this button to activate.
button.grab_default()
button.show()
window.show()

def main():
    gtk.main()
    return 0

if __name__ == "__main__":
    ScrolledWindowExample()
    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:

```python
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:

```python
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:

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

Buttons are added to a `ButtonBox` using the usual `Container` method:

```python
button_box.add(widget)
```

The `buttonbox.py` example program illustrates all the different layout settings for `ButtonBoxes`. The resulting display is:
CHAPTER 10. CONTAINER WIDGETS

10.10. BUTTON BOXES

The source code for the buttonbox.py program is:

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

# example buttonbox.py

import pygtk
pygtk.require('2.0')
import gtk

class ButtonBoxExample:
    # Create a Button Box with the specified parameters
    def create_bbox(self, horizontal, title, spacing, layout):
        frame = gtk.Frame(title)
```

The source code for the buttonbox.py program is:

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

# example buttonbox.py

import pygtk
pygtk.require('2.0')
import gtk

class ButtonBoxExample:
    # Create a Button Box with the specified parameters
    def create_bbox(self, horizontal, title, spacing, layout):
        frame = gtk.Frame(title)
```

```python
# Create a Button Box with the specified parameters
    def create_bbox(self, horizontal, title, spacing, layout):
        frame = gtk.Frame(title)
```
```
if horizontal:
    bbox = gtk.HButtonBox()
else:
    bbox = gtk.VButtonBox()

bbox.set_border_width(5)
frame.add(bbox)

# Set the appearance of the Button Box
bbox.set_layout(layout)
bbox.set_spacing(spacing)

button = gtk.Button(stock=gtk.STOCK_OK)
bbox.add(button)

button = gtk.Button(stock=gtk.STOCK_CANCEL)
bbox.add(button)

button = gtk.Button(stock=gtk.STOCK_HELP)
bbox.add(button)

return frame

def __init__(self):
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.set_title("Button Boxes")
    window.connect("destroy", lambda x: gtk.main_quit())
    window.set_border_width(10)

    main_vbox = gtk.VBox(False, 0)
    window.add(main_vbox)

    frame_horz = gtk.Frame("Horizontal Button Boxes")
    main_vbox.pack_start(frame_horz, True, True, 10)

    vbox = gtk.VBox(False, 0)
    vbox.set_border_width(10)
    frame_horz.add(vbox)

    vbox.pack_start(self.create_bbox(True, "Spread (spacing 40)",
                          40, gtk.BUTTONBOX_SPREAD),
                      True, True, 0)

    vbox.pack_start(self.create_bbox(True, "Edge (spacing 30)",
                          30, gtk.BUTTONBOX_EDGE),
                      True, True, 5)

    vbox.pack_start(self.create_bbox(True, "Start (spacing 20)",
                          20, gtk.BUTTONBOX_START),
                      True, True, 5)

    vbox.pack_start(self.create_bbox(True, "End (spacing 10)",
                          10, gtk.BUTTONBOX_END),
                      True, True, 5)

    frame_vert = gtk.Frame("Vertical Button Boxes")
    main_vbox.pack_start(frame_vert, True, True, 10)

    hbox = gtk.HBox(False, 0)
    hbox.set_border_width(10)
    frame_vert.add(hbox)
```
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:
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 # example toolbar.py
3
4 import pygtk
5 pygtk.require('2.0')
6 import gtk
7
8 class ToolbarExample:
9     # This method is connected to the Close button or
10     # closing the window from the WM
11     def delete_event(self, widget, event=None):
12         gtk.main_quit()
13         return False
14
15     # that’s easy... when one of the buttons is toggled, we just
16     # check which one is active and set the style of the toolbar
17     # accordingly
18     def radio_event(self, widget, toolbar):
19         if self.text_button.get_active():
20             toolbar.set_style(gtk.TOOLBAR_TEXT)
21         elif self.icon_button.get_active():
22             toolbar.set_style(gtk.TOOLBAR_ICONS)
23         elif self.both_button.get_active():
24             toolbar.set_style(gtk.TOOLBAR_BOTH)
25
26     # even easier, just check given toggle button and enable/disable
27     # tooltips
28     def toggle_event(self, widget, toolbar):
29         toolbar.set_tooltips(widget.get_active())
30
31     def __init__(self):
32         # Here is our main window (a dialog) and a handle for the ←
33         # Ok, we need a toolbar, an icon with a mask (one for all of
34         # the buttons) and an icon widget to put this icon in (but
35         # we’ll create a separate widget for each button)
36         # create a new window with a given title, and nice size
37         dialog = gtk.Dialog()
38         dialog.set_title("GTKToolbar Tutorial")
39         dialog.set_size_request(450, 250)
```

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     def __init__(self):
17         # Here is our main window (a dialog) and a handle for the ←
18         # Ok, we need a toolbar, an icon with a mask (one for all of
19         # the buttons) and an icon widget to put this icon in (but
20         # we’ll create a separate widget for each button)
21         # create a new window with a given title, and nice size
22         dialog = gtk.Dialog()
23         dialog.set_title("GTKToolbar Tutorial")
24         dialog.set_size_request(450, 250)
```
41 dialog.set_resizable(True)
42 # typically we quit if someone tries to close us
43 dialog.connect("delete_event", self.delete_event)
44 # to make it nice we’ll put the toolbar into the handle box,
45 # so that it can be detached from the main window
46 handlebox = gtk.HandleBox()
47 dialog.vbox.pack_start(handlebox, False, False, 5)

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)

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(  # button label
       "Close",
       "Closes this app", # this button’s tooltip
       "Private", # tooltip private info
       iconw, # icon widget
       self.delete_event) # a signal
64 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(  # type of element
6      gtk.TOOLBAR_CHILD_RADIOBUTTON,
6      None,
6      "Icon",
6      "Only icons in toolbar",
6      "Private",
6      iconw, # icon
66 self.radio_event, # signal
68 toolbar) # data for signal
69 toolbar.append_space()
70 self.icon_button = icon_button
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_S-PACE 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.

```python
iconw = gtk.Image() # icon widget
text_button = toolbar.append_element(  
    gtk.TOOlBAR_CHILD_RADIOBUTTON,  
    icon_button,  
    "Text",  
    "Only texts in toolbar",  
    "Private",  
    iconw,  
    self.radio_event,  
    toolbar)  
toolbar.append_space()  
self.text_button = text_button

iconw = gtk.Image() # icon widget
iconw.set_from_file("gtk.xpm")
both_button = toolbar.append_element(  
    gtk.TOOlBAR_CHILD_RADIOBUTTON,  
    text_button,  
    "Both",  
    "Icons and text in toolbar",  
    "Private",  
    iconw,  
    self.radio_event,  
    toolbar)  
toolbar.append_space()  
self.both_button = both_button  
both_button.set_active(True)
```

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).

```python
# here we have just a simple toggle button
iconw = gtk.Image() # icon widget
iconw.set_from_file("gtk.xpm")
tooltips_button = toolbar.append_element(  
    gtk.TOOlBAR_CHILD_TOGGLEBUTTON,  
    None,  
    "Tooltips",  
    "Toolbar with or without tips",  
    "Private",  
    iconw,  
    self.toggle_event,  
    toolbar)  
toolbar.append_space()  
tooltips_button.set_active(True)
```

A toggle button can be created in the obvious way (if one knows how to create radio buttons already).

```python
# to pack a widget into toolbar, we only have to
# create it and append it with an appropriate tooltip
entry = gtk.Entry()
toolbar.append_widget(entry, "This is just an entry", "Private")
```
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).

```python
135 # well, it isn't created within the toolbar, so we must still show it
136 show it
137 entry.show()
138
139 # that's it! let's show everything.
140 toolbar.show()
141 handlebox.show()
142 dialog.show()
143
def main():
144    # rest in gtk_main and wait for the fun to begin!
145    gtk.main()
146    return 0
147
148 if __name__ == "__main__":
149    ToolbarExample()
150    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](image)

### 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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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:

```python
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.

```python
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 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](image)

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
```
class NotebookExample:
    # This method rotates the position of the tabs
    def rotate_book(self, button, notebook):
        notebook.set_tab_pos((notebook.get_tab_pos()+1) % 4)

    # Add/Remove the page tabs and the borders
    def tabsborder_book(self, button, notebook):
        tval = False
        bval = False
        if self.show_tabs == False:
            tval = True
        if self.show_border == False:
            bval = True

        notebook.set_show_tabs(tval)
        self.show_tabs = tval
        notebook.set_show_border(bval)
        self.show_border = bval

    # Remove a page from the notebook
    def remove_book(self, button, notebook):
        page = notebook.get_current_page()
        notebook.remove_page(page)
        # Need to refresh the widget --
        # This forces the widget to redraw itself.
        notebook.queue_draw_area(0,0,-1,-1)

    def delete(self, widget, event=None):
        gtk.main_quit()
        return False

    def __init__(self):
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.connect("delete_event", self.delete)
        window.set_border_width(10)

        table = gtk.Table(3,6,False)
        window.add(table)

        # Create a new notebook, place the position of the tabs
        notebook = gtk.Notebook()
        notebook.set_tab_pos(gtk.POS_TOP)
        table.attach(notebook, 0,6,0,1)
        notebook.show()
        self.show_tabs = True
        self.show_border = True

        # Let's append a bunch of pages to the notebook
        for i in range(5):
            bufferf = "Append Frame %d" % (i+1)
            bufferl = "Page %d" % (i+1)

            frame = gtk.Frame(bufferf)
            frame.set_size_request(100, 75)
            frame.show()

            label = gtk.Label(bufferf)
            frame.add(label)
            label.show()

            label = gtk.Label(bufferl)
            notebook.append_page(frame, label)
# Now let's add a page to a specific spot
checkbutton = gtk.CheckButton("Check me please!")
checkbutton.set_size_request(100, 75)
checkbutton.show()

label = gtk.Label("Add page")
notebook.insert_page(checkbutton, label, 2)

# Now finally let's prepend pages to the notebook
for i in range(5):
  bufferf = "Prepend Frame %d" % (i+1)
  bufferl = "PPage %d" % (i+1)

  frame = gtk.Frame(bufferf)
  frame.set_border_width(10)
  frame.set_size_request(100, 75)
  frame.show()

  label = gtk.Label(bufferf)
  frame.add(label)
  label.show()

  label = gtk.Label(bufferl)
  notebook.prepend_page(frame, label)

# Set what page to start at (page 4)
notebook.set_current_page(3)

# Create a bunch of buttons
button = gtk.Button("close")
button.connect("clicked", self.delete)
table.attach(button, 0,1,1,2)
button.show()

button = gtk.Button("next page")
button.connect("clicked", lambda w: notebook.next_page())
table.attach(button, 1,2,1,2)
button.show()

button = gtk.Button("prev page")
button.connect("clicked", lambda w: notebook.prev_page())
table.attach(button, 2,3,1,2)
button.show()

button = gtk.Button("tab position")
button.connect("clicked", self.rotate_book, notebook)
table.attach(button, 3,4,1,2)
button.show()

button = gtk.Button("tabs/border on/off")
button.connect("clicked", self.tabsborder_book, notebook)
table.attach(button, 4,5,1,2)
button.show()

button = gtk.Button("remove page")
button.connect("clicked", self.remove_book, notebook)
table.attach(button, 5,6,1,2)
button.show()

table.show()
window.show()
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:

```python
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:

```python
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:

```python
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
```

The program is invoked like:

```python
gtk.mainloop()
```

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`:

```python
socket = gtk.Socket()
```

The `Socket` must be contained in a toplevel window before you invoke the `add_id()` method:

```python
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:

```python
plug = gtk.Plug(0L)
```

and then pass the number returned by the `Plug` `get_id()` method to the `Socket` `add_id()` method:

```python
socket.add_id(plug)
```

or you can invoke the `Socket` `get_id()` method:

```python
window_id = socket.get_id()
```

to get the window ID for the socket, and then create the plug with:

```python
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`:

```python
#!/usr/bin/python
import string

```
```python
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:

```python
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.

```python
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):

```python
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:

```python
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")
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:

```python
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:

```python
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:

```python
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.

```python
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:

  ```python
  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:

  ```python
  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](image)

The `menu.py` program source code is:

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

# example menu.py

import pygtk
pygtk.require('2.0')

import gtk

def handler(widget, event):

  # code to handle the event

widget.connect_object("event", handler, menu)
```

1 #!/usr/bin/env python
2 # example menu.py
3 import pygtk
4 pygtk.require('2.0')
import gtk

class MenuExample:
    def __init__(self):
        # create a new window
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_size_request(200, 100)
        window.set_title("GTK Menu Test")
        window.connect("delete_event", lambda w,e: gtk.main_quit())

        # Init the menu-widget, and remember -- never
        # show() the menu widget!!
        # This is the menu that holds the menu items, the one that
        # will pop up when you click on the "Root Menu" in the app
        menu = gtk.Menu()

        # Next we make a little loop that makes three menu-entries for
        # "test-menu". Notice the call to gtk_menu_append. Here we are
        # adding a list of menu items to our menu. Normally, we'd also
        # catch the "clicked" signal on each of the menu items and setup a
        # callback for it, but it's omitted here to save space.
        for i in range(3):
            # Copy the names to the buf.
            buf = "Test-undermenu - %d" % i

            # Create a new menu-item with a name...
            menu_items = gtk.MenuItem(buf)

            # ...and add it to the menu.
            menu.append(menu_items)

            # Do something interesting when the menuitem is selected
            menu_items.connect("activate", self.menuitem_response, buf)

        # Show the widget
        menu_items.show()

        # This is the root menu, and will be the label
        # displayed on the menu bar. There won't be a signal handler attached,
        # as it only pops up the rest of the menu when pressed.
        root_menu = gtk.MenuItem("Root Menu")

        root_menu.show()

        # Now we specify that we want our newly created "menu" to be the
        # menu for the "root menu"
        root_menu.set_submenu(menu)

        # A vbox to put a menu and a button in:
        vbox = gtk.VBox(False, 0)
        window.add(vbox)
        vbox.show()

        # Create a menu-bar to hold the menus and add it to our main window
        menu_bar = gtk.MenuBar()
        vbox.pack_start(menu_bar, False, False, 2)
        menu_bar.show()

        # Create a button to which to attach menu as a popup
        button = gtk.Button("press me")
        button.connect_object("event", self.button_press, menu)
        vbox.pack_end(button, True, True, 2)
        button.show()
# And finally we append the menu-item to the menu-bar -- this is the "root" menu-item I have been raving about =)
menu_bar.append (root_menu)

# always display the window as the last step so it all splashes on the screen at once.
window.show()

# Respond to a button-press by posting a menu passed in as widget.
# Note that the "widget" argument is the menu being posted, NOT the button that was pressed.
def button_press(self, widget, event):
    if event.type == gtk.gdk.BUTTON_PRESS:
        widget.popup(None, None, None, event.button, event.time)
        # Tell calling code that we have handled this event the buck stops here.
        return True
    # Tell calling code that we have not handled this event pass it on.
    return False

# Print a string when a menu item is selected
def menuitem_response(self, widget, string):
    print "%s" % string

def main():
    gtk.main()
    return 0

if __name__ == '__main__':
    MenuExample()
    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:
The source code for `itemfactory.py` is:

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

# example itemfactory.py

import pygtk
pygtk.require('2.0')
import gtk

class ItemFactoryExample:
    # Obligatory basic callback
    def print_hello(self, w, data):
        print "Hello, World!"

    # This is the ItemFactoryEntry structure used to generate new menus.
    # Item 1: The menu path. The letter after the underscore indicates an
    # accelerator key once the menu is open.
    # Item 2: The accelerator key for the entry
    # Item 3: The callback.
    # Item 4: The callback action. This changes the parameters with
    #        which the callback is called. The default is 0.
    # Item 5: The item type, used to define what kind of an item it is.
    #        Here are the possible values:
    #        NULL -> "<Item>
    #        "    -> "<Item>
    #        "Title>" -> create a title item
    #        "<Item>" -> create a simple item
    #        "<CheckItem>" -> create a check item
    #        "<ToggleItem>" -> create a toggle item
    #        "<RadioItem>" -> create a radio item
    #        <path> -> path of a radio item to link against
    #        "<Separator>" -> create a separator
    #        "<Branch>" -> create an item to hold sub items (←
    #                  optional)
    #        "<LastBranch>" -> create a right justified branch

    def get_main_menu(self, window):
        accel_group = gtk.AccelGroup()
```

Figure 11.2 Item Factory Example
# This function initializes the item factory.
# Param 1: The type of menu - can be MenuBar, Menu, 
# or OptionMenu.
# Param 2: The path of the menu.
# Param 3: A reference to an AccelGroup. The item factory sets up
# the accelerator table while generating menus.
item_factory = gtk.ItemFactory(gtk.MenuBar, "<main>", accel_group)

# This method generates the menu items. Pass to the item factory
# the list of menu items
item_factory.create_items(self.menu_items)

# Attach the new accelerator group to the window.
window.add_accel_group(accel_group)

# need to keep a reference to item_factory to prevent its ↓
# destruction
self.item_factory = item_factory
# Finally, return the actual menu bar created by the item factory.
return item_factory.get_widget("<main>")

def __init__(self):
    self.menu_items = (
        ("/_File", None, None, 0, "<Branch>"),
        ("/_File/New", "<control>N", self.print_hello, 0, None),
        ("/_File/Open", "<control>O", self.print_hello, 0, None),
        ("/_File/Save", "<control>S", self.print_hello, 0, None),
        ("/_File/Save_As", None, None, 0, None),
        ("/_File/Quit", "<control>Q", gtk.main_quit, 0, None),
        ("/_Options", None, None, 0, "<Branch>"),
        ("/_Options/Test", None, None, 0, None),
        ("/_Help", None, None, 0, "<LastBranch>"),
        ("/_Help/About", None, None, 0, None),
    )
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.connect("destroy", lambda w: gtk.main_quit(), "WM destroy")
    window.set_title("Item Factory")
    window.set_size_request(300, 200)
    main_vbox = gtk.VBox(False, 1)
    main_vbox.set_border_width(1)
    window.add(main_vbox)
    main_vbox.show()
    menubar = self.get_main_menu(window)
    menubar.pack_start(menubar, False, True, 0)
    window.add(menubar)
    window.show()
    def main():
        gtk.main()
        return 0
    if __name__ == "__main__":
        ItemFactoryExample()
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:

```python
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:

```python
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:

```python
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
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:

```python
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:

```python
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:

```python
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:

```python
navajowhite = colormap.alloc('navajo white')
cyans = colormap.alloc(0, 65535, 65535)
red = colormap.alloc_color('#FF0000', True, True)
```

The colormap associated with a widget can be retrieved using the method:

```python
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:

<table>
<thead>
<tr>
<th>Cap Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAP_NOT_LAST</td>
<td>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.</td>
</tr>
<tr>
<td>CAP_BUTT</td>
<td>The ends of the lines are drawn squared off and extending to the coordinates of the end point.</td>
</tr>
<tr>
<td>CAP_ROUND</td>
<td>The ends of the lines are drawn as semicircles with the diameter equal to the line width and centered at the end point.</td>
</tr>
<tr>
<td>CAP_PROJECTING</td>
<td>The ends of the lines are drawn squared off and extending half the width of the line beyond the end point.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOLID</td>
<td>draw with the foreground color.</td>
</tr>
<tr>
<td>TILED</td>
<td>draw with a tiled pixmap.</td>
</tr>
<tr>
<td>STIPPLED</td>
<td>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.</td>
</tr>
<tr>
<td>OPAQUE_STIPPLED</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td></td>
</tr>
<tr>
<td>INVERT</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>AND_REVERSE</td>
<td></td>
</tr>
<tr>
<td>AND_INVERT</td>
<td></td>
</tr>
<tr>
<td>NOOP</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>EQUIV</td>
<td></td>
</tr>
<tr>
<td>OR_REVERSE</td>
<td></td>
</tr>
<tr>
<td>COPY_INVERT</td>
<td></td>
</tr>
<tr>
<td>OR_INVERT</td>
<td></td>
</tr>
<tr>
<td>NAND</td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOIN_MITER</td>
<td>the sides of each line are extended to meet at an angle.</td>
</tr>
<tr>
<td>JOIN_ROUND</td>
<td>the sides of the two lines are joined by a circular arc.</td>
</tr>
<tr>
<td>JOIN_BEVEL</td>
<td>the sides of the two lines are joined by a straight line which makes an equal angle with each line.</td>
</tr>
</tbody>
</table>

line_style specifies the style that a line will be drawn with. The available styles are:

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE_SOLID</td>
<td>lines are drawn as continuous segments.</td>
</tr>
<tr>
<td>LINE_ON_OFF_DASH</td>
<td>even segments are drawn; odd segments are not drawn.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIP_BY_CHILDREN</td>
<td>only draw onto the window itself but not its child windows</td>
</tr>
<tr>
<td>INCLUDE_INFERIORS</td>
<td>draw onto the window and its child windows.</td>
</tr>
</tbody>
</table>

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:

```python
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:

```python
gc.cap_style = CAP_BUTT
gc.line_width = 10
gc.fill = SOLID
gc.foreground = mycolor
```

or by using the following methods:

```python
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:

```python
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:

```python
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:

```python
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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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, xsr, ysr, xdest, ydest, width, height)

gc is the Graphics Context.
src is the source drawable.
xsr and ysr 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, ysr, xdest, ydest, width, height)

gc is the Graphics Context.
image is the source image.
xsr and ysr 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:

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

# example drawingarea.py

import pygtk
pygtk.require('2.0')
import gtk
import operator
import time
import string
```
12 class DrawingAreaExample:
    def __init__(self):
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_title("Drawing Area Example")
        self.area = gtk.DrawingArea()
        self.area.set_size_request(400, 300)
        self.pangolayout = self.area.create_pango_layout("")
        self.sw = gtk.ScrolledWindow()
        self.sw.add_with_viewport(self.area)
        self.table = gtk.Table(2, 2)
        self.hruler = gtk.HRuler()
        self.vruler = gtk.VRuler()
        self.hruler.set_range(0, 400, 0, 400)
        self.vruler.set_range(0, 300, 0, 300)
        self.hruler.set_range(0, 400, 0, 400)
        self.vruler.set_range(0, 300, 0, 300)
        self.table.attach(self.hruler, 1, 2, 0, 1, yoptions=0)
        self.table.attach(self.vruler, 0, 1, 1, 2, xoptions=0)
        self.table.attach(self.sw, 1, 2, 1, 2)
        window.add(self.table)
        self.area.set_events(gtk.gdk.POINTER_MOTION_MASK |
                             gtk.gdk.POINTER_MOTION_HINT_MASK)
        self.area.connect("expose-event", self.area_expose_cb)
        self.area.connect_object("motion_notify_event", motion_notify,
                                 self.hruler)
        self.area.connect_object("motion_notify_event", motion_notify,
                                 self.vruler)
        self.hadj = self.sw.get_hadjustment()
        self.vadj = self.sw.get_vadjustment()
        def val_cb(adj, ruler, horiz):
            if horiz:
                span = self.sw.get_allocation()[3]
            else:
                span = self.sw.get_allocation()[2]
            l, u, p, m = ruler.get_range()
            v = adj.value
            ruler.set_range(v, v+span, p, m)
            while gtk.events_pending():
                gtk.main_iteration()
            self.hadj.connect('value-changed', val_cb, self.hruler, True)
            self.vadj.connect('value-changed', val_cb, self.vruler, False)
        def size_allocate_cb(wid, allocation):
            x, y, w, h = allocation
            l, u, p, m = self.hruler.get_range()
            m = max(m, w)
            l, u, p, m = self.vruler.get_range()
            m = max(m, h)
            self.sw.set_range(l, l+w, p, m)
            self.sw.connect('size-allocate', size_allocate_cb)
            self.area.show()
        def area_expose_cb(self, area, event):
            self.style = self.area.get_style()
            self.gc = self.style.fg_gc[gtk.STATE_NORMAL]
            self.draw_point(10, 10)
            self.draw_points(110, 10)
            self.draw_line(210, 10)
def self.draw_lines(310, 10)
self.draw_segments(10, 100)
self.draw_rectangles(110, 100)
self.draw_arcs(210, 100)
self.drawPixmap(310, 100)
self.draw_polygon(10, 200)
self.draw_rgb_image(110, 200)
return True

def self.draw_point(x, y):
self.area.window.draw_point(self.gc, x+30, y+30)
self.pangolayout.set_text("Point")
self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout)
return

def self.draw_points(x, y):
points = [(x+10, y+10), (x+10, y), (x+40, y+30),
(x+30, y+10), (x+50, y+10)]
self.area.window.draw_points(self.gc, points)
self.pangolayout.set_text("Points")
self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout)
return

def self.draw_line(x, y):
self.area.window.draw_line(self.gc, x+10, y+10, x+20, y+30)
self.pangolayout.set_text("Line")
self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout)
return

def self.draw_lines(x, y):
points = [(x+10, y+10), (x+10, y), (x+40, y+30),
(x+30, y+10), (x+50, y+10)]
self.area.window.draw_lines(self.gc, points)
self.pangolayout.set_text("Lines")
self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout)
return

def self.draw_segments(x, y):
segments = ((x+20, y+10, x+20, y+70), (x+60, y+10, x+60, y+70),
(x+10, y+30, x+70, y+30), (x+10, y+50, x+70, y+50))
self.area.window.draw_segments(self.gc, segments)
self.pangolayout.set_text("Segments")
self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout)
return

def self.draw_rectangles(x, y):
self.area.window.draw_rectangle(self.gc, False, x, y, 80, 70)
self.area.window.draw_rectangle(self.gc, True, x+10, y+10, 20)
self.area.window.draw_rectangle(self.gc, True, x+50, y+10, 20)
self.pangolayout.set_text("Rectangles")
self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout)
return

def self.draw_arcs(x, y):

131 self.area.window.draw_arc(self.gc, False, x+10, y, 70, 70, 0, 360*64)
132 self.area.window.draw_arc(self.gc, True, x+30, y+20, 10, 10, 0, 360*64)
133 self.area.window.draw_arc(self.gc, True, x+50, y+20, 10, 10, 0, 360*64)
134 self.area.window.draw_arc(self.gc, True, x+30, y+10, 30, 50, 210*64, 120*64)
135 self.pangolayout.set_text("Arcs")
136 self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout)
137 return
138
def draw_pixmap(self, x, y):
139 pixmap, mask = gtk.gdk.pixmap_create_from_xpm(
self.area.window, self.style.bg[gtk.STATE_NORMAL], "gtk.xpm")
140 self.area.window.draw_drawable(self.gc, pixmap, 0, 0, x+15, y+25, -1, -1)
141 self.pangolayout.set_text("Pixmap")
142 self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout)
143 return
144
def draw_polygon(self, x, y):
145 points = [(x+10,y+60), (x+10,y+20), (x+40,y+70),
(x+30,y+30), (x+50,y+40)]
146 self.area.window.draw_polygon(self.gc, True, points)
147 self.pangolayout.set_text("Polygon")
148 self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout)
149 return
150
def draw_rgb_image(self, x, y):
151 b = 80*3*80*[0]
152 for i in range(80):
153 for j in range(80):
154 b[3*80*i+3*j] = chr(255-3*i)
155 b[3*80*i+3*j+1] = chr(255-3*abs(i-j))
156 b[3*80*i+3*j+2] = chr(255-3*j)
157 buff = string.join(b, '')
158 self.area.window.draw_rgb_image(self.gc, x, y, 80, 80,
gtk.gdk.RGB_DITHER_NONE, buff, 80*3)
159 self.pangolayout.set_text("RGB Image")
160 self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout)
161 return
162
def main():
163 gtk.main()
164 return 0
165
166 if __name__ == "__main__":
167 DrawingAreaExample()
168 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.

```python
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:

```python
TextView.set_buffer(buffer)
```

Use the following method to retrieve a reference to the TextBuffer from a TextView:

```python
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, use the following method:

```python
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:

```python
setting = textView.get_editable()
```

When the TextView is not editable, you probably should hide the cursor using the method:

```python
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:

```python
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:

```python
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:

```python
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:

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

textview.set_pixels_below_lines(pixels_below_line)
```

```python
textview.set_pixels_inside_wrap(pixels_inside_wrap)
```

```python
textview.set_tabs(tabs)
```

```python
tabs = textview.get_tabs()
```

```python
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:

```python
1 #!/usr/bin/env python
2
3 # example textview-basic.py
4
5 import pygtk
```
class TextViewExample:
    def toggle_editable(self, checkbutton, textview):
        textview.set_editable(checkbutton.get_active())
    
    def toggle_cursor_visible(self, checkbutton, textview):
        textview.set_cursor_visible(checkbutton.get_active())
    
    def toggle_left_margin(self, checkbutton, textview):
        if checkbutton.get_active():
            textview.set_left_margin(50)
        else:
            textview.set_left_margin(0)
    
    def toggle_right_margin(self, checkbutton, textview):
        if checkbutton.get_active():
            textview.set_right_margin(50)
        else:
            textview.set_right_margin(0)
    
    def new_wrap_mode(self, radiobutton, textview, val):
        if radiobutton.get_active():
            textview.set_wrap_mode(val)
    
    def new_justification(self, radiobutton, textview, val):
        if radiobutton.get_active():
            textview.set_justification(val)
    
    def close_application(self, widget):
        gtk.main_quit()
    
    def __init__(self):
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_resizable(True)
        window.connect("destroy", self.close_application)
        window.set_title("TextView Widget Basic Example")
        window.set_border_width(0)

        box1 = gtk.VBox(False, 0)
        window.add(box1)
        box1.show()

        box2 = gtk.VBox(False, 10)
        box2.set_border_width(10)
        box1.pack_start(box2, True, True, 0)
        box2.show()

        sw = gtk.ScrolledWindow()
        sw.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
        textview = gtk.TextView()
        textbuffer = textview.get_buffer()
        sw.add(textview)
        sw.show()
        textview.show()

        box2.pack_start(sw)
        # Load the file textview-basic.py into the text window
        infile = open("textview-basic.py", "r")
        if infile:
            string = infile.read()
        infile.close()
textbuffer.set_text(string)

hbox = gtk.HButtonBox()
box2.pack_start(hbox, False, False, 0)
hbox.show()

vbox = gtk.VBox()
vbox.show()
hbox.pack_start(vbox, False, False, 0)
# check button to toggle editable mode
check = gtk.CheckButton("Editable")
vbox.pack_start(check, False, False, 0)
check.connect("toggled", self.toggle_editable, textview)
check.set_active(True)
check.show()

# check button to toggle cursor visibility
check = gtk.CheckButton("Cursor Visible")
vbox.pack_start(check, False, False, 0)
check.connect("toggled", self.toggle_cursor_visible, textview)
check.set_active(True)
check.show()

# check button to toggle left margin
check = gtk.CheckButton("Left Margin")
vbox.pack_start(check, False, False, 0)
check.connect("toggled", self.toggle_left_margin, textview)
check.set_active(False)
check.show()

# check button to toggle right margin
check = gtk.CheckButton("Right Margin")
vbox.pack_start(check, False, False, 0)
check.connect("toggled", self.toggle_right_margin, textview)
check.set_active(False)
check.show()

# radio buttons to specify wrap mode
vbox = gtk.VBox()
vbox.show()
hbox.pack_start(vbox, False, False, 0)
radio = gtk.RadioButton(None, "WRAP__NONE")
vbox.pack_start(radio, False, True, 0)
radio.connect("toggled", self.new_wrap_mode, textview, gtk.WRAP_NONE)
radio.set_active(True)
radio.show()
radio = gtk.RadioButton(radio, "WRAP__CHAR")
vbox.pack_start(radio, False, True, 0)
radio.connect("toggled", self.new_wrap_mode, textview, gtk.WRAP_CHAR)
radio.show()
radio = gtk.RadioButton(radio, "WRAP__WORD")
vbox.pack_start(radio, False, True, 0)
radio.connect("toggled", self.new_wrap_mode, textview, gtk.WRAP_WORD)
radio.show()

# radio buttons to specify justification
vbox = gtk.VBox()
vbox.show()
hbox.pack_start(vbox, False, False, 0)
radio = gtk.RadioButton(None, "JUSTIFY__LEFT")
vbox.pack_start(radio, False, True, 0)
radio.connect("toggled", self.new_justification, textview, gtk.JUSTIFY_LEFT)
radio.set_active(True)
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:

```python
line_count = textbuffer.get_line_count()
```

Likewise you can get the number of characters in the textbuffer using:

```python
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:

```python
modified = textbuffer.get_modified()
```

If the program saves the contents of the textbuffer the following method can be used to reset the modified flag:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
textbuffer.delete_mark(mark)
textbuffer.delete_mark_by_name(name)
```

A mark can be retrieved by name using the method:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
text_view.add_child_at_anchor(child, anchor)
```

The list of widgets at a particular buffer anchor can be retrieved using the method:

```python
widget_list = anchor.get_widgets()
```

A widget can also be added to a TextView using the method:

```python
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:
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):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bg_color</td>
<td>background color</td>
</tr>
<tr>
<td>fg_color</td>
<td>foreground color</td>
</tr>
<tr>
<td>bg_stipple</td>
<td>background stipple bitmap</td>
</tr>
<tr>
<td>fg_stipple</td>
<td>foreground stipple bitmap</td>
</tr>
<tr>
<td>rise</td>
<td>offset of text above baseline</td>
</tr>
<tr>
<td>underline</td>
<td>style of underline</td>
</tr>
<tr>
<td>strikethrough</td>
<td>whether text is strikethrough</td>
</tr>
<tr>
<td>draw_bg</td>
<td>TRUE if some tags affect the drawing of the background</td>
</tr>
<tr>
<td>justification</td>
<td>style of justification</td>
</tr>
<tr>
<td>direction</td>
<td>which direction the text runs</td>
</tr>
<tr>
<td>font</td>
<td>PangoFontDescription in use</td>
</tr>
<tr>
<td>font_scale</td>
<td>scale of the font in use</td>
</tr>
<tr>
<td>left_margin</td>
<td>location of left margin</td>
</tr>
<tr>
<td>right_margin</td>
<td>location of right margin</td>
</tr>
<tr>
<td>pixels_above_lines</td>
<td>pixels spacing above a line</td>
</tr>
<tr>
<td>pixels_below_lines</td>
<td>pixel spacing below a line</td>
</tr>
<tr>
<td>pixels_inside_wrap</td>
<td>pixel spacing between wrapped lines</td>
</tr>
<tr>
<td>tabs</td>
<td>PangoTabArray in use</td>
</tr>
<tr>
<td>wrap_mode</td>
<td>mode of wrap in use</td>
</tr>
</tbody>
</table>
### 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.
The equivalence of two TextIter can be determined with the method:

```python
are_equal = lhs.equal(rhs)
```

Two TextIter can be compared with the method:

```python
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 TextIter use the method:

```python
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:

```python
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:

```python
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:

```python
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:

```python
result = iter.is_cursor_position()
```
13.4.7 Moving Through Text

TextIter 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:

```python
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:

```python
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:

```python
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:

```python
eresult = iter.forward_to_tag_toggle(tag)
eresult = iter.backward_to_tag_toggle(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:

```python
match_start, match_end = iter.forward_search(str, flags, limit=None)
mach_start, match_end = iter.backward_search(str, flags, limit=None)
```

The `return` value is a tuple containing `TextIter`s 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:

```python
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:

```python
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:

```python
setting = textmark.getVisible()
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:

```python
buffer = textmark.get_buffer()
```

You can determine whether a `TextMark` has been deleted using the method:

```python
setting = textmark.get_deleted()
```

The left gravity of a `TextMark` can be retrieved using the method:

```python
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

Text Tags specify attributes that can be applied to a range of text in a TextBuffer. Each TextBuffer has a TextTagTable that contains the TextTags that can be applied within the TextBuffer. TextTagTables can be used with more than one TextBuffer to provide consistent text styles.

13.6.1 Text Tags

TextTags can be named or anonymous. A TextTag is created using the function:

```python
tag = gtk.TextTag(name=None)
```

If `name` is not specified or is `None` the `tag` will be anonymous. TextTags 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Read / Write</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Read / Write</td>
<td>Name of the text tag. None if anonymous.</td>
</tr>
<tr>
<td>background</td>
<td>Write</td>
<td>Background color as a string</td>
</tr>
<tr>
<td>foreground</td>
<td>Write</td>
<td>Foreground color as a string</td>
</tr>
<tr>
<td>background-gdk</td>
<td>Read / Write</td>
<td>Background color as a GdkColor</td>
</tr>
<tr>
<td>foreground-gdk</td>
<td>Read / Write</td>
<td>Foreground color as a GdkColor</td>
</tr>
<tr>
<td>background-stipple</td>
<td>Read / Write</td>
<td>Bitmap to use as a mask when drawing the text background</td>
</tr>
<tr>
<td>foreground-stipple</td>
<td>Read / Write</td>
<td>Bitmap to use as a mask when drawing the text foreground</td>
</tr>
<tr>
<td>font</td>
<td>Read / Write</td>
<td>Font description as a string, e.g. &quot;Sans Italic 12&quot;</td>
</tr>
<tr>
<td>font-desc</td>
<td>Read / Write</td>
<td>Font description as a PangoFontDescription</td>
</tr>
<tr>
<td>family</td>
<td>Read / Write</td>
<td>Name of the font family, e.g. Sans, Helvetica, Times, Monospace</td>
</tr>
<tr>
<td>style</td>
<td>Read / Write</td>
<td>Font style as a PangoStyle, e.g. pango.STYLE_ITALIC.</td>
</tr>
<tr>
<td>variant</td>
<td>Read / Write</td>
<td>Font variant as a PangoVariant, e.g. pango.VARIANT_SMALL_CAPS.</td>
</tr>
<tr>
<td>weight</td>
<td>Read / Write</td>
<td>Font weight as an integer, see predefined values in PangoWeight; for example, pango.WEIGHT_BOLD.</td>
</tr>
<tr>
<td>stretch</td>
<td>Read / Write</td>
<td>Font stretch as a PangoStretch, e.g. pango.STRETCH_CONDENSED.</td>
</tr>
<tr>
<td>size</td>
<td>Read / Write</td>
<td>Font size in Pango units.</td>
</tr>
<tr>
<td>size-points</td>
<td>Read / Write</td>
<td>Font size in points</td>
</tr>
<tr>
<td>scale</td>
<td>Read / Write</td>
<td>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.</td>
</tr>
<tr>
<td>pixels-above-lines</td>
<td>Read / Write</td>
<td>Pixels of blank space above paragraphs</td>
</tr>
<tr>
<td>pixels-below-lines</td>
<td>Read / Write</td>
<td>Pixels of blank space below paragraphs</td>
</tr>
<tr>
<td>pixels-inside-wrap</td>
<td>Read / Write</td>
<td>Pixels of blank space between wrapped lines in a paragraph</td>
</tr>
<tr>
<td>editable</td>
<td>Read / Write</td>
<td>Whether the text can be modified by the user</td>
</tr>
<tr>
<td>wrap-mode</td>
<td>Read / Write</td>
<td>Whether to wrap lines never, at word boundaries, or at character boundaries</td>
</tr>
<tr>
<td>justification</td>
<td>Read / Write</td>
<td>Left, right, or center justification</td>
</tr>
<tr>
<td>direction</td>
<td>Read / Write</td>
<td>Text direction, e.g. right-to-left or left-to-right</td>
</tr>
<tr>
<td>left-margin</td>
<td>Read / Write</td>
<td>Width of the left margin in pixels</td>
</tr>
<tr>
<td>indent</td>
<td>Read / Write</td>
<td>Amount to indent the paragraph, in pixels</td>
</tr>
<tr>
<td>strikethrough</td>
<td>Read / Write</td>
<td>Whether to strike through the text</td>
</tr>
<tr>
<td>right-margin</td>
<td>Read / Write</td>
<td>Width of the right margin in pixels</td>
</tr>
<tr>
<td>underline</td>
<td>Read / Write</td>
<td>Style of underline for this text</td>
</tr>
<tr>
<td>rise</td>
<td>Read / Write</td>
<td>Offset of text above the baseline (below the baseline if rise is negative) in pixels</td>
</tr>
<tr>
<td>background-full-height</td>
<td>Read / Write</td>
<td>Whether the background color fills the entire line height or only the height of the tagged characters</td>
</tr>
<tr>
<td>language</td>
<td>Read / Write</td>
<td>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</td>
</tr>
<tr>
<td>tabs</td>
<td>Read / Write</td>
<td>Custom tabs for this text</td>
</tr>
</tbody>
</table>
The attributes can be set by using the method:

\[ \text{tag.set_property(name, value)} \]

Where \( \text{name} \) is a string containing the name of the property and \( \text{value} \) is what the property should be set to.

Likewise the attribute value can be retrieved with the method:

\[ \text{value = tag.get_property(name)} \]

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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Read / Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>background-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>foreground-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>background-stipple-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>foreground-stipple-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>family-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>style-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>variant-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>weight-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>stretch-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>size-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>scale-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>pixels-above-lines-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>pixels-below-lines-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>pixels-inside-wrap-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>editable-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>wrap-mode-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>justification-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>direction-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>left-margin-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>indent-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>strikethrough-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>right-margin-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>underline-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>rise-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>background-full-height-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>language-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>tabs-set</td>
<td>Read / Write</td>
</tr>
<tr>
<td>invisible-set</td>
<td>Read / Write</td>
</tr>
</tbody>
</table>

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:

\[ \text{if tag.get_property("justification-set")}: \]
\[ \text{justification = tag.get_property("justification")} \]

The priority of a tag is by default the order in which they are added to the \texttt{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:

\[ \text{priority = tag.get_priority()} \]
\[ \text{tag.set_priority(priority)} \]

The priority of a tag must be between 0 and one less than the \texttt{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:

```python
table = TextTagTable()
```

A TextTag can be added to a TextTagTable using the method:

```python
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:

```python
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:

```python
table.remove(tag)
```

The size of the TextTagTable can be obtained with the method:

```python
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, TextIter, TextMark, TextTag, TextTagTable. Figure 13.2 illustrates its operation:
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 TextTags 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.Labels 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 main-
tained 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 TreeViewColumns 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 TreeViewColumns are created and inserted in the TreeView. Each of these will
present a single display column.

• For each TreeViewColumn one or more CellRenderers are created and added to the TreeVi-
ewColumn.

• 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:

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

# example basictreeview.py

import pygdk
pygdk.require('2.0')
import gtk

class BasicTreeViewExample:
    # close the window and quit
```
def delete_event(self, widget, event, data=None):
    gtk.main_quit()
    return False

def __init__(self):
    # Create a new window
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.set_title("Basic TreeView Example")
    self.window.set_size_request(200, 200)
    self.window.connect("delete_event", self.delete_event)

    # create a TreeStore with one string column to use as the model
    self.treestore = gtk.TreeStore(str)

    # we'll add some data now - 4 rows with 3 child rows each
    for parent in range(4):
        piter = self.treestore.append(None, ['parent %i' % parent])
        for child in range(3):
            self.treestore.append(piter, ['child %i of parent %i' % (child, parent)])

    # create the TreeView using treestore
    self.treeview = gtk.TreeView(self.treestore)

    # create the TreeViewColumn to display the data
    self.tvcolumn = gtk.TreeViewColumn('Column 0')

    # add tvcolumn to treeview
    self.treeview.append_column(self.tvcolumn)

    # create a CellRendererText to render the data
    self.cell = gtk.CellRendererText()

    # add the cell to the tvcolumn and allow it to expand
    self.tvcolumn.pack_start(self.cell, True)

    # set the cell "text" attribute to column 0 - retrieve text
    # from that column in treestore
    self.tvcolumn.add_attribute(self.cell, 'text', 0)

    # make it searchable
    self.treeview.set_search_column(0)

    # Allow sorting on the column
    self.tvcolumn.set_sort_column_id(0)

    # Allow drag and drop reordering of rows
    self.treeview.set_reorderable(True)

    self.window.add(self.treeview)
    self.window.show_all()

def main():
    gtk.main()

if __name__ == '__main__':
    tvexample = BasicTreeViewExample()
    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:

```python
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 "5" 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:

```python
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 TreeIters can become
invalid. A TreeModel that supports persistent TreeIters should set the gtk.TREE_MODEL_ITERS_P-
ERSIST 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:

```python
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.

```python
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.

```python
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:

```python
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.

```python
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.

```python
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:

```python
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 TreeRowRefer-
ence will point at the same data row as long as it exists.
You can create a `TreeRowReference` using its constructor:

```python
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:

```python
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:

```python
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:

```python
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:

```python
... 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:

```python
treiter = liststore.remove(iter)
```

where *iter* is a `TreeIter` pointing at the row to remove. The returned `TreeIter (treiter)` 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`:

```python
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:

```python
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 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 TreeIter *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:

```python
'two'
'three'
'one'
'four'
```

The method call:

```python
liststore.reorder([2, 1, 3, 0])
```

would produce the resulting order:

```python
'three'
'two'
'four'
'one'
```

**Note**

These methods will only rearrange unsorted ListStores.

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:

```python
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:

```python
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:

```python
new_order[newpos] = oldpos
```

For example, if `treestore` contained four rows:

```python
'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 TreeStores.

### 14.2.6.4 Managing Multiple Rows

One of the trickier aspects of dealing with ListStores and TreeStores 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 ListStores and TreeStores 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 TreeIters.

Assuming that TreeIters 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):
```

185
where \texttt{model} is the TreeModel data store, \texttt{path} is the tree path of a row in \texttt{model}, \texttt{iter} is a TreeIter pointing at \texttt{path} and \texttt{user_data} is the passed in data. If \texttt{func} returns TRUE the \texttt{foreach()} method will cease iterating and return.

The problem with that is that changing the contents of the store while the \texttt{foreach()} method is iterating over it may have unpredictable results. Using the \texttt{foreach()} method to create and save TreeRowReferences to the rows to be removed and then removing them after the \texttt{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 \texttt{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:

```python
# 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:

```python
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 \texttt{search()} function iterates recursively over the row (specified by \texttt{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 \texttt{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 \texttt{for} statement or a list comprehension. For example:

```python
liststore = gtk.ListStore(str, str)
```
CHAPTER 14. TREE VIEW WIDGET

14.2. THE TREEMODEL INTERFACE AND...

... # 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:

```python
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:

```python
... 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:

```python
... 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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 TreeIter `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:

```python
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:
You can check if a store has a default sort function using the method:

```python
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:

```python
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:

```python
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:

```python
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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Read-Write</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;enable-search&quot;</td>
<td>Read-Write</td>
<td>If <code>TRUE</code>, the user can search through columns interactively. Default is <code>TRUE</code></td>
</tr>
<tr>
<td>&quot;expand-column&quot;</td>
<td>Read-Write</td>
<td>The column for the expander. Default is 0</td>
</tr>
<tr>
<td>&quot;fixed-height-mode&quot;</td>
<td>Read-Write</td>
<td>If <code>TRUE</code>, assume all rows have the same height thereby speeding up display. Available in GTK+ 2.4 and above. Default is <code>FALSE</code></td>
</tr>
<tr>
<td>&quot;hadjustment&quot;</td>
<td>Read-Write</td>
<td>The horizontal <code>Adjustment</code> for the widget. New one created by default.</td>
</tr>
<tr>
<td>&quot;headers-clickable&quot;</td>
<td>Write</td>
<td>If <code>TRUE</code>, the column headers respond to click events. Default is <code>FALSE</code></td>
</tr>
<tr>
<td>&quot;headers-visible&quot;</td>
<td>Read-Write</td>
<td>If <code>TRUE</code>, show the column header buttons. Default is <code>TRUE</code></td>
</tr>
<tr>
<td>&quot;model&quot;</td>
<td>Read-Write</td>
<td>The model for the tree view. Default is <code>None</code></td>
</tr>
<tr>
<td>&quot;reorderable&quot;</td>
<td>Read-Write</td>
<td>If <code>TRUE</code>, the view is reorderable. Default is <code>FALSE</code></td>
</tr>
<tr>
<td>&quot;rules-hint&quot;</td>
<td>Read-Write</td>
<td>If <code>TRUE</code>, hint to the theme engine to draw rows in alternating colors. Default is <code>FALSE</code></td>
</tr>
<tr>
<td>&quot;search-column&quot;</td>
<td>Read-Write</td>
<td>The model column to search when searching through code. Default is -1.</td>
</tr>
<tr>
<td>&quot;vadjustment&quot;</td>
<td>Read-Write</td>
<td>The vertical <code>Adjustment</code> for the widget. New one created by default.</td>
</tr>
</tbody>
</table>

The corresponding methods are:

```python
column = treeview.get_expander_column() treeview.set_expander_column(column)
```

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

![TreeViewColumns with CellRenderers](image)

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:
### 14.4. CELL RENDERERS

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mode&quot;</td>
<td>Read-Write</td>
<td>The editable mode of the CellRenderer. One of: <code>gtk.CELL_RENDERER_MODE_INERT</code>, <code>gtk.CELL_RENDERER_MODE_ACTIVATABLE</code> or <code>gtk.CELL_RENDERER_MODE_EDITABLE</code></td>
</tr>
<tr>
<td>&quot;visible&quot;</td>
<td>Read-Write</td>
<td>If TRUE the cell is displayed</td>
</tr>
<tr>
<td>&quot;xalign&quot;</td>
<td>Read-Write</td>
<td>The fraction of free space to the left of the cell in the range 0.0 to 1.0.</td>
</tr>
<tr>
<td>&quot;yalign&quot;</td>
<td>Read-Write</td>
<td>The fraction of free space above the cell in the range 0.0 to 1.0.</td>
</tr>
<tr>
<td>&quot;xpad&quot;</td>
<td>Read-Write</td>
<td>The amount of padding to the left and right of the cell.</td>
</tr>
<tr>
<td>&quot;ypad&quot;</td>
<td>Read-Write</td>
<td>The amount of padding above and below cell.</td>
</tr>
<tr>
<td>&quot;width&quot;</td>
<td>Read-Write</td>
<td>The fixed width of the cell.</td>
</tr>
<tr>
<td>&quot;height&quot;</td>
<td>Read-Write</td>
<td>The fixed height of the cell.</td>
</tr>
<tr>
<td>&quot;is-expander&quot;</td>
<td>Read-Write</td>
<td>If TRUE the row has children</td>
</tr>
<tr>
<td>&quot;is-expanded&quot;</td>
<td>Read-Write</td>
<td>If TRUE the row has children and it is expanded to show the children.</td>
</tr>
<tr>
<td>&quot;cell-background&quot;</td>
<td>Write</td>
<td>The background color of the cell as a string.</td>
</tr>
<tr>
<td>&quot;cell-background-gdk&quot;</td>
<td>Read-Write</td>
<td>The background color of the cell as a <code>gtk.gdk.Color</code>.</td>
</tr>
<tr>
<td>&quot;cell-background-set&quot;</td>
<td>Read-Write</td>
<td>If TRUE the cell background color is set by this cellrenderer.</td>
</tr>
<tr>
<td>&quot;pixbuf&quot;</td>
<td>Read-Write</td>
<td>The pixbuf to render - overridden by &quot;stock-id&quot;</td>
</tr>
<tr>
<td>&quot;pixbuf-expander-open&quot;</td>
<td>Read-Write</td>
<td>Pixbuf for open expander.</td>
</tr>
<tr>
<td>&quot;pixbuf-expander-closed&quot;</td>
<td>Read-Write</td>
<td>Pixbuf for closed expander.</td>
</tr>
<tr>
<td>&quot;stock-id&quot;</td>
<td>Read-Write</td>
<td>The stock ID of the stock icon to render</td>
</tr>
<tr>
<td>&quot;stock-size&quot;</td>
<td>Read-Write</td>
<td>The size of the rendered icon</td>
</tr>
<tr>
<td>&quot;stock-detail&quot;</td>
<td>Read-Write</td>
<td>Render detail to pass to the theme engine</td>
</tr>
<tr>
<td>&quot;text&quot;</td>
<td>Read-Write</td>
<td>Text to render</td>
</tr>
<tr>
<td>&quot;markup&quot;</td>
<td>Read-Write</td>
<td>Marked up text to render.</td>
</tr>
<tr>
<td>&quot;attributes&quot;</td>
<td>Read-Write</td>
<td>A list of style attributes to apply to the text of the renderer.</td>
</tr>
<tr>
<td>&quot;background&quot;</td>
<td>Write</td>
<td>Background color as a string</td>
</tr>
<tr>
<td>&quot;foreground&quot;</td>
<td>Write</td>
<td>Foreground color as a string</td>
</tr>
<tr>
<td>&quot;background-gdk&quot;</td>
<td>Read-Write</td>
<td>Background color as a <code>gtk.gdk.Color</code>.</td>
</tr>
<tr>
<td>&quot;foreground-gdk&quot;</td>
<td>Read-Write</td>
<td>Foreground color as a <code>gtk.gdk.Color</code>.</td>
</tr>
<tr>
<td>&quot;font&quot;</td>
<td>Read-Write</td>
<td>Font description as a string</td>
</tr>
<tr>
<td>&quot;font-desc&quot;</td>
<td>Read-Write</td>
<td>Font description as a <code>pango.FontDescription</code>.</td>
</tr>
<tr>
<td>&quot;family&quot;</td>
<td>Read-Write</td>
<td>Name of the font family, e.g. Sans, Helvetica, Times, Monospace</td>
</tr>
<tr>
<td>&quot;style&quot;</td>
<td>Read-Write</td>
<td>Font style</td>
</tr>
<tr>
<td>&quot;variant&quot;</td>
<td>Read-Write</td>
<td>Font variant</td>
</tr>
<tr>
<td>&quot;weight&quot;</td>
<td>Read-Write</td>
<td>Font weight</td>
</tr>
<tr>
<td>&quot;stretch&quot;</td>
<td>Read-Write</td>
<td>Font stretch</td>
</tr>
<tr>
<td>&quot;size&quot;</td>
<td>Read-Write</td>
<td>Font size</td>
</tr>
<tr>
<td>&quot;size-points&quot;</td>
<td>Read-Write</td>
<td>Font size in points</td>
</tr>
<tr>
<td>&quot;scale&quot;</td>
<td>Read-Write</td>
<td>Font scaling factor</td>
</tr>
<tr>
<td>&quot;editable&quot;</td>
<td>Read-Write</td>
<td>If TRUE the text can be modified by the user</td>
</tr>
<tr>
<td>&quot; strikethrough&quot;</td>
<td>Read-Write</td>
<td>If TRUE strike through the text</td>
</tr>
<tr>
<td>&quot;underline&quot;</td>
<td>Read-Write</td>
<td>Style of underline for this text</td>
</tr>
<tr>
<td>&quot;rise&quot;</td>
<td>Read-Write</td>
<td>Offset of text above the baseline (below the baseline if rise is negative)</td>
</tr>
<tr>
<td>&quot;language&quot;</td>
<td>Read-Write</td>
<td>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.</td>
</tr>
</tbody>
</table>

The above properties are available for all `CellRenderer` subclasses. The individual `CellRenderer` types also have their own properties.

The `CellRendererPixbuf` has these properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;pixbuf&quot;</td>
<td>Read-Write</td>
<td>The pixbuf to render - overridden by &quot;stock-id&quot;</td>
</tr>
<tr>
<td>&quot;pixbuf-expander-open&quot;</td>
<td>Read-Write</td>
<td>Pixbuf for open expander.</td>
</tr>
<tr>
<td>&quot;pixbuf-expander-closed&quot;</td>
<td>Read-Write</td>
<td>Pixbuf for closed expander.</td>
</tr>
<tr>
<td>&quot;stock-id&quot;</td>
<td>Read-Write</td>
<td>The stock ID of the stock icon to render</td>
</tr>
<tr>
<td>&quot;stock-size&quot;</td>
<td>Read-Write</td>
<td>The size of the rendered icon</td>
</tr>
<tr>
<td>&quot;stock-detail&quot;</td>
<td>Read-Write</td>
<td>Render detail to pass to the theme engine</td>
</tr>
</tbody>
</table>

The `CellRendererText` has a large number of properties mostly dealing with style specification:
### CHAPTER 14. TREE VIEW WIDGET

#### 14.4. CELLRENDERERS

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;single-paragraph-mode&quot;</td>
<td>Read-Write</td>
<td>If TRUE, keep all text in a single paragraph. GTK+ 2.4 and above.</td>
</tr>
</tbody>
</table>

- **"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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;activatable&quot;</td>
<td>Read-Write</td>
<td>If TRUE, the toggle button can be activated</td>
</tr>
<tr>
<td>&quot;active&quot;</td>
<td>Read-Write</td>
<td>If TRUE, the button is active.</td>
</tr>
<tr>
<td>&quot;radio&quot;</td>
<td>Read-Write</td>
<td>If TRUE, draw the toggle button as a radio button</td>
</tr>
<tr>
<td>&quot;inconsistent&quot;</td>
<td>Read-Write</td>
<td>If TRUE, the button is in an inconsistent state. GTK+ 2.2 and above.</td>
</tr>
</tbody>
</table>

The properties can be set for all rows by using the gobject.set_property() method. See the treeview-column.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)
it = treestore.append(None, [win])
it = treestore.append(it, [treeview])
it = treestore.append(it, [tvcolumn])
it = treestore.append(it, [cell])
it = treestore.append(None, [treestore])
...
```

The resulting display should be something like Figure 14.3:

---

Figure 14.3 CellRenderer Data Function

![Figure 14.3 CellRenderer Data Function](image)

---

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:
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> markup can'
' change\n<i>style</i> <big>size</big>, <u>underline,'\n'<s>strikethrough</s></u>,
'and <span font_family="URW Chancery L"><big>font family'
'e.g. URW Chancery L</big></span><</span>
'<span background="cyan">cyan background</span>'])
...
```

produces a display similar to Figure 14.5:
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:

```python
cellrenderertext.set_property('editable', True)
```

Individual cells can be set editable by adding an attribute to the TreeViewColumn using the CellRendererText similar to:

```python
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:

```python
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 get `new_text` in the data store you probably want to pass the TreeModel as `user_data` in the connect() method:

```python
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:

```python
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:

```python
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.

```python
#!/usr/bin/env python
# vim: ts=4:sw=4:tw=78:nowrap
""
    Demonstration using editable and activatable CellRenderers ""
import pygtk
pygtk.require("2.0")
import gtk, gobject

tasks = {
    "Buy groceries": "Go to Asda after work",
    "Do some programming": "Remember to update your software",
    "Power up systems": "Turn on the client but leave the server",
    "Watch some tv": "Remember to catch ER"
}

class GUI_Controller:
    ""
    The GUI class is the controller for our application ""
    def __init__(self):
        # setup the main window
```
self.root = gtk.Window(type=gtk.WINDOW_TOplevel)
self.root.set_title("CellRenderer Example")
self.root.connect("destroy", self.destroy_cb)

# Get the model and attach it to the view
self.mdl = Store.get_model()
self.view = Display.make_view( self.mdl )
# Add our view into the main window
self.root.add(self.view)
self.root.show_all()
return

def destroy_cb(self, *kw):
    """ Destroy callback to shutdown the app """
gtk.main_quit()
return

def run(self):
    """ run is called to set off the GTK mainloop """
gtk.main()
return

class InfoModel:
    """ The model class holds the information we want to display """
def __init__(self):
    """ Sets up and populates our gtk.TreeStore """
self.tree_store = gtk.TreeStore( gobject.TYPE_STRING,
                                gobject.TYPE_BOOLEAN )
    # places the global people data into the list
    # we form a simple tree.
    for item in tasks.keys():
        parent = self.tree_store.append( None, (item, None) )
        self.tree_store.append( parent, (tasks[item],None) )
    return

def get_model(self):
    """ Returns the model """
    if self.tree_store:
        return self.tree_store
    else:
        return None

class DisplayModel:
    """ Displays the Info_Model model in a view """
def make_view( self, model ):
    """ Form a view for the Tree Model """
self.view = gtk.TreeView( model )
    # setup the text cell renderer and allows these
    # cells to be edited.
self.renderer = gtk.CellRendererText()
self.renderer.set_property( 'editable', True )
self.renderer.connect( 'edited', self.col0_edited_cb, model )

self.renderer1 = gtk.CellRendererToggle()
self.renderer1.set_property('activatable', True)
self.renderer1.connect( 'toggled', self.col1_toggled_cb, model )

# Connect column0 of the display with column 0 in our list model
# The renderer will then display whatever is in column 0 of
# our model.
self.column0 = gtk.TreeViewColumn("Name", self.renderer, text=0)
self.column0.set_active_column(0)
self.column0.set_cellrenderertext(True)
self.column0.set_cellrenderer(self.renderer)
self.column0.set_active_column(0)

# The columns active state is attached to the second column
# in the model. So when the model says True then the button
# will show as active e.g on.
self.column1 = gtk.TreeViewColumn("Complete", self.renderer1)
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.

14.5 TreeViewColumns

14.5.1 Creating TreeViewColumns

A `TreeViewColumn` is created using the constructor:

```python
treedviewcolumn = 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:

```python
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:

```python
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:

```python
cell_list = treeviewcolumn.get_cell_renderers()
```

returns a list of all the CellRenderer in the column.

The `clear()` method removes all the CellRenderer attributes from the TreeViewColumn:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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

TreeView 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:


• 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.

```python
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:

```python
treeselection = treeview.get_selection()
```

You can retrieve the `TreeView` associated with a `TreeSelection` by calling the method:

```python
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:

```python
mode = treeselection.get_mode()
```

The mode can be set using:

```python
treeselection.set_mode(mode)
```

where `mode` is one of the above selection modes.
14.8.3 Retreiving 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:

```python
(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:

```python
(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:

```python
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:

```python
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:

```python
... 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:

```python
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:

```python
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` 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:

```python
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:

```python
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:

```python
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:

```python
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)
```

```python
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:

```python
def callback(widget, drag_context, selection_data, info, timestamp)
```

206
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.SELECT_ON_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:

```python
...
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:

```python
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.

```python
1 #!/usr/bin/env python
2 # example treeviewdnd.py
3 import pygtk
4 pygtk.require('2.0')
5 import gtk
6
7 class TreeViewDnDExample:
8     
9     TARGETS = [
10         ('MY_TREE_MODEL_ROW', gtk.TARGET_SAME_WIDGET, 0),
11         ('text/plain', 0, 1),
12         ('TEXT', 0, 2),
13         ('STRING', 0, 3),
14     ]
15     
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return False
19     
20     def clear_selected(self, button):
21         selection = self.treeview.get_selection()
22         model, iter = selection.get_selected()
23         if iter:
24             model.remove(iter)
25         return
26 ...
```
def __init__(self):
    # Create a new window
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.set_title("URL Cache")
    self.window.set_size_request(200, 200)
    self.window.connect("delete_event", self.delete_event)
    self.scrolledwindow = gtk.ScrolledWindow()
    self.vbox = gtk.VBox()
    self.hbox = gtk.HButtonBox()
    self.vbox.pack_start(self.scrolledwindow, True)
    self.vbox.pack_start(self.hbox, False)
    self.b0 = gtk.Button('Clear All')
    self.b1 = gtk.Button('Clear Selected')
    self.hbox.pack_start(self.b0)
    self.hbox.pack_start(self.b1)
    # create a liststore with one string column to use as the model
    self.liststore = gtk.ListStore(str)
    # create the TreeView using liststore
    self.treeview = gtk.TreeView(self.liststore)
    # create a CellRenderer to render the data
    self.cell = gtk.CellRendererText()
    # create the TreeViewColumns to display the data
    self.tvcolumn = gtk.TreeViewColumn('URL', self.cell, text=0)
    # add columns to treeview
    self.treeview.append_column(self.tvcolumn)
    self.b0.connect_object('clicked', gtk.ListStore.clear, self.
    self.b1.connect('clicked', self.clear_selected)
    # make treeview searchable
    self.treeview.set_search_column(0)
    # Allow sorting on the column
    self.tvcolumn.set_sort_column_id(0)
    # Allow enable drag and drop of rows including row move
    self.treeview.enable_model_drag_source( gtk.gdk.BUTTON1_MASK,
        gtk.gdk.ACTION_DEFAULT|gtk.gdk.ACTION_MOVE)
    self.treeview.enable_model_drag_dest(self.TARGETS,
        gtk.gdk.ACTION_DEFAULT)
    self.treeview.connect("drag_data_get", self.drag_data_get_data)
    self.treeview.connect("drag_data_received", self.
    self.scrolledwindow.add(self.treeview)
    self.window.add(self.vbox)
    self.window.show_all()

def drag_data_get_data(self, treeview, context, selection, target_id, etime):
    treeselection = treeview.get_selection()
    model, iter = treeselection.get_selected()
    data = model.get_value(iter, 0)
selection.set(selection.target, 8, data)

def drag_data_received_data(self, treeview, context, x, y, selection, info, etime):
    model = treeview.get_model()
    data = selection.data
    drop_info = treeview.get_dest_row_at_pos(x, y)
    if drop_info:
        path, position = drop_info
        iter = model.get_iter(path)
        if (position == gtk.TREE_VIEW_DROP_BEFORE
            or position == gtk.TREE_VIEW_DROP_INTO_OR_BEFORE):
            model.insert_before(iter, [data])
        else:
            model.insert_after(iter, [data])
    else:
        model.append([data])

def main():
    gtk.main()

if __name__ == '__main__':
    treeviewdndex = TreeViewDnDExample()
    main()

The result of running the example program treeviewdnd.py is illustrated in Figure 14.8:

![TreeView Drag and Drop Example](image)

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 TreeIter from the child model to the sorted model and back:

```
sorted_path = treemodelsort.convert_child_path_to_path(child_path)
cchild_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)
cchild_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)
cchild_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:
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:

```python
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:

```python
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:

```python
...  
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:

```python
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:

```python
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:

```python
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:

```python
...  
def match_type(model, iter, udata):
    value = model.get_value(iter, 2)
    return value in udata  
...  
show_vals = ["OPEN", "NEW", "RESC"]  
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 \texttt{model} is the \texttt{TreeModelFilter}, \texttt{iter} is a \texttt{TreeIter} that points to a row in model, \texttt{column} is the number of the column that a value is needed for and \texttt{user_data} is the parameter \texttt{data.func} must return a value matching the type for \texttt{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 \texttt{TreeViewColumn.set_cell_data_func()} method.

Usually within the modify function, you will have to convert the \texttt{TreeModelFilter TreeIter} to a \texttt{TreeIter} in the child model using:

\begin{verbatim}
child_iter = treemodelfilter.convert_iter_to_child_iter(filter_iter)
\end{verbatim}

Of course, you'll also need to retrieve the child model using:

\begin{verbatim}
child_model = treemodelfilter.get_model()
\end{verbatim}

These give you access to the child model row and its values for generating the value for the specified \texttt{TreeModelFilter row} and column. There's also a method to convert a child \texttt{TreeIter} to a filter model \texttt{TreeIter} and methods to convert filter model paths to and from child tree paths:

\begin{verbatim}
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)
\end{verbatim}

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 \texttt{TreeModel}.

## 14.11 The Generic TreeModel

When you find that the standard \texttt{TreeModel}s are not sufficiently powerful for your application needs, you can use the \texttt{GenericTreeModel} to build your own custom \texttt{TreeModel} in Python. Creating a \texttt{GenericTreeModel} may be useful when there are performance issues with the standard \texttt{TreeStore} and \texttt{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 \texttt{TreeStore} or \texttt{ListStore}.

### 14.11.1 \texttt{GenericTreeModel} Overview

With the \texttt{GenericTreeModel} you build and manage your data model and provide external access through the standard \texttt{TreeModel} interface by defining a set of class methods. \texttt{PyGTK} implements the \texttt{TreeModel} interface and arranges for your \texttt{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 \texttt{GenericTreeModel} are the row references that are wrapped by the external \texttt{TreeIter}s. And these references are not visible to the application.

Let’s examine in detail the \texttt{GenericTreeModel} interface that you have to provide.

### 14.11.2 The \texttt{GenericTreeModel} Interface

The \texttt{GenericTreeModel} interface consists of the following methods that must be implemented in your custom tree model:

\begin{verbatim}
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)
\end{verbatim}
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:

```python
flm = FileListModel()
print flm.on_get_column_type(1), flm.get_column_type(1)
```
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:

```python
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:

```python
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:

```python
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:

```python
def on_iter_next(self, rowref):
    try:
        i = self.files.index(rowref)+1
```

would print:

```python
<type 'str'> <GType gchararray (64)>
```
The index of the \texttt{rowref} file name is determined and the next file name is returned or \texttt{None} is returned if there is no next file.

The \texttt{on_iter_children()} method should return a row reference to the first child row of the row specified by \texttt{rowref}. If \texttt{rowref} is \texttt{None}, a reference to the first top level row is returned. If there is no child row \texttt{None} is returned. For our example:

\begin{verbatim}
def on_iter_children(self, rowref):
    if rowref:
        return None
    return self.files[0]
\end{verbatim}

Since the model is a list model only the top level (rowref=\texttt{None}) can have child rows. None is returned if \texttt{rowref} contains a file name.

The \texttt{on_iter_has_child()} method should return \texttt{TRUE} if the row specified by \texttt{rowref} has child rows; \texttt{FALSE} otherwise. Our example returns \texttt{FALSE} since no row can have a child:

\begin{verbatim}
def on_iter_has_child(self, rowref):
    return False
\end{verbatim}

The \texttt{on_iter_n_children()} method should return the number of child rows that the row specified by \texttt{rowref} has. If \texttt{rowref} is \texttt{None}, the number of top level rows is returned. Our example returns \texttt{0} if \texttt{rowref} is not \texttt{None}:

\begin{verbatim}
def on_iter_n_children(self, rowref):
    if rowref:
        return 0
    return len(self.files)
\end{verbatim}

The \texttt{on_iter_nth_child()} method should return a row reference to the \texttt{n}th child row of the row specified by \texttt{parent}. If \texttt{parent} is \texttt{None}, a reference to the \texttt{n}th top level row is returned. Our example returns the \texttt{n}th top level row reference if \texttt{parent} is \texttt{None}. Otherwise \texttt{None} is returned:

\begin{verbatim}
def on_iter_nth_child(self, rowref, n):
    if rowref:
        try:
            return self.files[n]
        except IndexError:
            return None
\end{verbatim}

The \texttt{on_iter_parent()} method should return a row reference to the parent row of the row specified by \texttt{rowref}. If \texttt{rowref} points to a top level row, \texttt{None} should be returned. Our example always returns \texttt{None} assuming that \texttt{rowref} must point to a top level row:

\begin{verbatim}
def on_iter_parent(child):
    return None
\end{verbatim}

This example is put together in the \texttt{filelisting-gtm.py} program. Figure 14.11 shows the result of running the program.
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:
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 `TreeViews` 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:

```python
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 `TreeViews` 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 `TreeViews` if the model has changed. The following methods are used to notify the `TreeViews` 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 TreeIter holds 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 TreeView-Column 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 TreeModels are thoroughly tested.
- managing the references of Python objects used by TreeIter 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 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:

```python
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:

```python
clipboard = gtk.clipboard_get(selection)
```

Finally, a Clipboard can also be created using the Widget method:

```python
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:
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. TextBuffers have similar methods though they also allow specifying the clipboard to use:

```python
editable.cut_clipboard()
editable.copy_clipboard()
editable.paste_clipboard()
```

```python
textbuffer.copy_clipboard(clipboard)
```

The selection text will be copied to the Clipboard specified by `clipboard`.

```python
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.

```python
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.

TextBuffers also have two methods to manage a set of Clipboards that are automatically set with the contents of the current selection:

```python
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:

```python
clipboard.set_with_data(targets, get_func, clear_func, user_data)
```

```python
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:

```python
def get_func(clipboard, selectiondata, info, data):
```

```python
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:
CHAPTER 15. NEW WIDGETS IN PYGTK 2.2

15.1. CLIPBOARDS

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
selectiondata = clipboard.wait_for_contents(target)
```

```python
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:
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 Actions and ActionGroups 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, MenuItems 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:

```python
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:

```python
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:

```python
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:

```python
#!/usr/bin/env python
import pygtk
pygtk.require('2.0')
import gtk
```

```python
1 pygtk.require('2.0')
2 import gtk
```
class SimpleAction:
    def __init__(self):
        # Create the toplevel window
        window = gtk.Window()
        window.set_size_request(70, 30)
        window.connect('destroy', lambda w: gtk.main_quit())

        # Create an accelerator group
        accelgroup = gtk.AccelGroup()
        # Add the accelerator group to the toplevel window
        window.add_accel_group(accelgroup)

        # Create an action for quitting the program using a stock item
        action = gtk.Action('Quit', None, None, gtk.STOCK_QUIT)
        # Connect a callback to the action
        action.connect('activate', self.quit_cb)

        # Create an ActionGroup named SimpleAction
        actiongroup = gtk.ActionGroup('SimpleAction')
        # Add the action to the actiongroup with an accelerator
        actiongroup.add_action_with_accel(action, None)

        # Have the action use accelgroup
        action.set_accel_group(accelgroup)

        # Connect the accelerator to the action
        action.connect_accelerator()

        # Create the button to use as the action proxy widget
        quitbutton = gtk.Button()
        # add it to the window
        window.add(quitbutton)

        # Connect the action to its proxy widget
        action.connect_proxy(quitbutton)

        return

    def quit_cb(self, b):
        print 'Quitting program'
        gtk.main_quit()

if __name__ == '__main__':
    sa = SimpleAction()
    gtk.main()
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:

```python
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:

```python
#!/usr/bin/env python
import pygtk
pygtk.require('2.0')
import gtk

class BasicAction:
    def __init__(self):
        # Create the toplevel window
        window = gtk.Window()
        window.connect('destroy', lambda w: gtk.main_quit())
        vbox = gtk.VBox()
        vbox.show()
        window.add(vbox)

        # Create an accelerator group
        accelgroup = gtk.AccelGroup()
        # Add the accelerator group to the toplevel window
        window.add_accel_group(accelgroup)

        # Create an action for quitting the program using a stock item
        action = gtk.Action('Quit', '_Quit me!', 'Quit the Program',
                            gtk.STOCK_QUIT)
        action.set_property('short-label', '_Quit')
        # Connect a callback to the action
        action.connect('activate', self.quit_cb)

        # Create an ActionGroup named BasicAction
        actiongroup = gtk.ActionGroup('BasicAction')
        # Add the action to the actiongroup with an accelerator
        actiongroup.add_action_with_accel(action, None)
        # Have the action use accelgroup
        action.set_accel_group(accelgroup)

        # Create a MenuBar
        menubar = gtk.MenuBar()
        menubar.show()
        vbox.pack_start(menubar, False)

        # Create the File Action and MenuItem
        file_action = gtk.Action('File', '_File', None, None)
```

---

**Figure 16.1** Simple Action Example

![Simple Action Example](image)
This example introduces an ActionGroup to hold the Actions used in the program. Section 16.1.2 will go into more detail on the use of ActionGroups.

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:

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:

```python
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:

```python
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](image)

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:

```python
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:

```python
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:

```python
actiongroup.add_action(action)
```

```python
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_with_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:

```python
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:

```python
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:

```python
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:

```python
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:
CHAPTER 16. NEW WIDGETS IN PYGTK 2.4

16.1. THE ACTION AND ACTIONGROUP

```python
def radiationlist = [('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(radiationlist, 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:

```python
action = actiongroup.get_action(action_name)
```

A list of all the Action objects contained in an ActionGroup can be retrieved using the method:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:
The active text can be retrieved using the method:

```python
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:

```python
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:

```python
active = combobox.get_active()
```

The active item can be set using the method:

```python
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:

```python
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:

```python
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:
CHAPTER 16. NEW WIDGETS IN PYGTK 2.4

16.2. COMBOBOX AND COMBOBOXENTRY

```
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:
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.

```python
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 CellRenderers.

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:

```python
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:

```python
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:

```python
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

![ComboBoxEntry Example](image)

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 Advanced ComboBoxEntry Use

The constructor for a ComboBoxEntry is:

```python
comboxboxentry = 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:

```python
liststore = gtk.ListStore(str)
comboxboxentry = 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:

```python
comboxboxentry.set_text_column(text_column)
text_column = comboxboxentry.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:

```python
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:

```python
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:

```python
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):

```python
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:

```python
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:

```python
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.
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.
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:

```python
completion = gtk.EntryCompletion()
```

You can use the `Entry` method `set_completion()` to associate an `EntryCompletion` with an `Entry`:

```python
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:

```python
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:
CHAPTER 16. NEW WIDGETS IN PYGTK 2.4

16.4. ENTRYCOMPLETION OBJECTS

This method is equivalent to the following:

```python
cell = CellRendererText()
cell.pack_start(cell)
cell.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:

```python
cell = CellRendererText()
cell.pack_start(cell)
cell.add_attribute(cell, 'text', column)
```

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:

```python
cell = CellRendererText()
cell.pack_start(cell)
cell.add_attribute(cell, 'text', column)
```

The signature of `func` is:

```python
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.

```python
... 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

![Expander Widget](image)

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 areFileChooserWidget 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
chooser.set_filename(filename)
```

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

```python
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:

```python
chooser.set_select_multiple(select_multiple)
```

where `select_multiple` 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:

```python
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:

```python
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:

```python
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:
The source code for the filechooser.py example program is:

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

# example filechooser.py

import pygtk
pygtk.require('2.0')

import gtk

# Check for new pygtk: this is new class in PyGtk 2.4
if gtk.pygtk_version < (2,3,90):
    print "PyGtk 2.3.90 or later required for this example"
    raise SystemExit

dialog = gtk.FileChooserDialog("Open..",
    None,
    gtk.FILE_CHOOSER_ACTION_OPEN,
    (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
    gtk.STOCK_OPEN, gtk.RESPONSE_OK))

dialog.set_default_response(gtk.RESPONSE_OK)

filter = gtk.FileFilter()
filter.set_name("All files")
filter.add_pattern(" *")
dialog.add_filter(filter)

filter = gtk.FileFilter()
filter.set_name("Images")
dialog.add_filter(filter)
```

Figure 16.12 File Selection Example
The UIManager

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.

Creating a UIManager

A UIManager instance is created by the constructor:

```python
uimanager = gtk.UIManager()
```

A new UIManager is created with an associated AccelGroup that can be retrieved using the method:

```python
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:
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:

```python
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:

```python
uimanager.remove_action_group(action_group)
```

A list of the ActionGroup objects associated with a UIManager can be retrieved using the method:

```python
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, 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:

```xml
<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:

```python
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:

```python
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:

```python
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:

```python
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:

```xml
<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:

```python
uimanager.ensure_update()
```

### 16.7.6 Accessing UI Widgets

You access a widget in the UI widget hierarchy by using the method:

```python
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:

```xml
<menubar name="MenuBar">
  <menu action="File">
    <menutitem action="Quit"/>
  </menu>
  <menu action="Sound">
    <menutitem action="Mute"/>
  </menu>
  <menu action="RadioBand">
    <menutitem action="AM"/>
    <menutitem action="FM"/>
    <menutitem 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:

```python
window = gtk.Window()
vbox = gtk.VBox()
menubar = uimanager.get_widget('/MenuBar')
toolbar = uimanager.get_widget('/Toolbar')
vbox.pack_start(menubar, 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:

```python
ssb = uimanager.get_widget('/Toolbar/RadioBandItems/SSB')
```

As a convenience all the top level widgets of a type can be retrieved using the method:

```python
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:
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.

![Simple UIManager Example](image)

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/>
```

...
16.7. THE UIMANAGER

is added to our example UI description:

```xml
<toolitem action="Loudness"/>
<separator/>
<placeholder name="RadioBandItems">
  <toolitem action="CB"/>
  <toolitem action="Shortwave"/>
</placeholder>
</toolbar>

the following merged UI description will be created:

```xml
<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"/>
    <toolitem action="CB"/>
    <toolitem 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"/>
  </placeholder>
</toolbar>
```
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:

```python
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:

```python
widget.activate()
```

causes the widget to emit the "activate" signal.

The method:

```python
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:

```python
widget.set_size_request(width, height)
```

sets the widget size to the given width and height.

18.1 Widget Flag Methods

The methods:

```python
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
The method:

```python
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:

```python
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:

```python
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.):
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:

```python
accel_group = gtk.AccelGroup()
```

The `accel_group` is attached to a top level widget with the following method:

```python
window.add_accel_group(accel_group)
```

An example of adding an accelerator:

```python
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:

```python
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:

```python
widget.set_style(style)
style = widget.get_style()
```

The following function:

```python
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:
A style contains the following attributes:

```python
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
bgPixmap  # 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:

```python
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:

```python
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:

```python
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.

```python
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:

```python
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:

```python
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:

```python
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:

```plaintext
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:

```python
gobject.source_remove(source_id)
```

The callback function should be similar to:

```python
def input_callback(source, condition):
    # code here
```

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:

```python
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:

```python
def callback(...):
    # code here
```

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:

```python
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

```python
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:

```python
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`:

```python
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:

```python
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:

```
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
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:

```python
    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:

```python
    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:

```python
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
```
CHAPTER 21. MANAGING SELECTIONS

21.2. RETRIEVING THE SELECTION

```python
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         if str(selection_data.type) == "STRING":
26             print "STRING TARGET: %s" % selection_data.get_text()
27         elif str(selection_data.type) == "ATOM":
28             targets = selection_data.get_targets()
29             for target in targets:
30                 name = str(target)
31                 if name != None:
32                     print "%s" % name
33                 else:
34                     print "(bad target)"
35             else:
36                 print "Selection was not returned as "STRING" or "ATOM"!"
37             return False
38
39 def __init__(self):
40         # Create the toplevel window
41         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
42         window.set_title("Get Selection")
43         window.set_border_width(10)
44         window.connect("destroy", lambda w: gtk.main_quit())
45
46         vbox = gtk.VBox(False, 0)
47         window.add(vbox)
48         vbox.show()
49
50         # Create a button the user can click to get the string target
51         button = gtk.Button("Get String Target")
52         eventbox = gtk.EventBox()
53         eventbox.add(button)
54         button.connect_object("clicked", self.get_stringtarget, eventbox)
55         eventbox.connect("selection_received", self.selection_received)
56         vbox.pack_start(eventbox)
57         eventbox.show()
58
59         # Create a button the user can click to get targets
60         button = gtk.Button("Get Targets")
61         eventbox = gtk.EventBox()
62         eventbox.add(button)
63         button.connect_object("clicked", self.get_targets, eventbox)
64         eventbox.connect("selection_received", self.selection_received)
65         vbox.pack_start(eventbox)
66         eventbox.show()
67
68 def main():
69         gtk.main()
70         return 0
71
```

276
Supposing 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:

```python
widget.selection_add_target(selection, target, info)
```

The callback has the signature:

```python
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:

```python
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:

```python
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:

```python
selection_data.set_text(str, len)
```

When prompted by the user, you claim ownership of the selection by calling:

```python
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:
The `setselection.py` source code is:

```python
#!/usr/bin/env python
#
# example setselection.py
#
import pygtk
pygtk.require('2.0')
import gtk
import time

class SetSelectionExample:
    # Callback when the user toggles the selection
    def selection_toggled(self, widget, window):
        if widget.get_active():
            self.have_selection = window.selection_owner_set("PRIMARY")
            # if claiming the selection failed, we return the button to
            # the out state
            if not self.have_selection:
                widget.set_active(False)
            else:
                if self.have_selection:
                    # Not possible to release the selection in PyGTK
                    # just mark that we don’t have it
                    self.have_selection = False
                return

    # Called when another application claims the selection
    def selection_clear(self, widget, event):
        self.have_selection = False
        widget.set_active(False)
        return True

    # Supplies the current time as the selection.
    def selection_handle(self, widget, selection_data, info, time_stamp):
        current_time = time.time()
        timestr = time.asctime(time.localtime(current_time))

        # When we return a single string, it should not be null terminated.
        # That will be done for us
        selection_data.set_text(timestr, len(timestr))
        return

    def __init__(self):
        self.have_selection = False
        # Create the toplevel window
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_title("Set Selection")
        window.set_border_width(10)
        window.connect("destroy", lambda w: gtk.main_quit())
        self.window = window
        # Create an eventbox to hold the button since it no longer has
        # a GdkWindow
        eventbox = gtk.EventBox()
        eventbox.show()
```

Figure 21.2 Set Selection Example
window.add(eventbox)

# Create a toggle button to act as the selection
selection_button = gtk.ToggleButton("Claim Selection")
eventbox.add(selection_button)

selection_button.connect("toggled", self.selection_toggled, eventbox)
eventbox.connect_object("selection_clear_event", self.selection_clear, selection_clear, selection_button)

eventbox.selection_add_target("PRIMARY", "STRING", 1)
eventbox.selection_add_target("PRIMARY", "COMPOUND_TEXT", 1)
eventbox.connect("selection_get", self.selection_handle)
selection_button.show()
window.show()

def main():
gtk.main()
return 0

if __name__ == "__main__":
SetSelectionExample()
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 handlers 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. `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.

```python
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:

```python
(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:

```python
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.

```python
widget.drag_source_unset()
```
### 22.3. DND METHODS

#### 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 initial 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 handler.

- 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.

```python
def drag_dest_set(widget, flags, targets, actions)
def drag_dest_unset(widget)
```

- `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 successful, GTK+ will call `drag_finish()`. If the action was a move and the drag was successful, 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:

```python
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:

```python
#!/usr/local/env python

import pygtk
pygtk.require('2.0')
import gtk

def motion_cb(wid, context, x, y, time):
    context.drag_status(gtk.gdk.ACTION_COPY, time)
    return True

def drop_cb(wid, context, x, y, time):
    l.set_text('
'.join([str(t) for t in context.targets]))
    context.finish(True, False, time)
    return True

w = gtk.Window()
w.set_size_request(200, 150)
w.drag_dest_set(0, [], 0)
w.connect('drag_motion', motion_cb)
w.connect('drag_drop', drop_cb)
w.connect('destroy', lambda w: gtk.main_quit())
l = gtk.Label()
w.add(l)
w.show_all()
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>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drag_motion</code></td>
<td>def drag_motion_cb(widget, drag_context, x, y, time, data):</td>
</tr>
<tr>
<td><code>drag_drop</code></td>
<td>def drag_drop_cb(widget, drag_context, x, y, time, data):</td>
</tr>
<tr>
<td><code>drag_data_received</code></td>
<td>def drag_data_received_cb(widget, drag_context, x, y, selection_data, info, time, data):</td>
</tr>
</tbody>
</table>

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.
The `dragndrop.py` source code is:

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

# example dragndrop.py

import pygtk
pygtk.require('2.0')
import gtk
import string, time

import gtkxpm

class DragNDropExample:
    HEIGHT = 600
    WIDTH = 600
    TARGET_TYPE_TEXT = 80
    TARGET_TYPE_PIXMAP = 81
    fromImage = [( "text/plain", 0, TARGET_TYPE_TEXT ),
                 ( "image/x-xpixmap", 0, TARGET_TYPE_PIXMAP )]
    toButton = [( "text/plain", 0, TARGET_TYPE_TEXT )]
    toCanvas = [( "image/x-xpixmap", 0, TARGET_TYPE_PIXMAP )]

    def layout_resize(self, widget, event):
        x, y, width, height = widget.get_allocation()
        if width > self.lwidth or height > self.lheight:
            self.lwidth = max(width, self.lwidth)
            self.lheight = max(height, self.lheight)
            widget.set_size(self.lwidth, self.lheight)

    def makeLayout(self):
        def layout_resize(self, widget, event):
            x, y, width, height = widget.get_allocation()
            if width > self.lwidth or height > self.lheight:
                self.lwidth = max(width, self.lwidth)
                self.lheight = max(height, self.lheight)
                widget.set_size(self.lwidth, self.lheight)

        def makeLayout(self):
```

---

**Figure 22.1** Drag and Drop Example

![Drag and Drop Example Image](image-url)
self.lwidth = self.WIDTH
self.lheight = self.HEIGHT
box = gtk.VBox(False, 0)
box.show()
table = gtk.Table(2, 2, False)
table.show()
box.pack_start(table, True, True, 0)
layout = gtk.Layout()
self.layout = layout
layout.set_size(self.lwidth, self.lheight)
layout.connect("size-allocate", self.layout_resize)
layout.show()
table.attach(layout, 0, 1, 0, 1, gtk.FILL|gtk.EXPAND,
             gtk.FILL|gtk.EXPAND, 0, 0)
# create the scrollbars and pack into the table
vScrollbar = gtk.VScrollbar(None)
vScrollbar.show()
table.attach(vScrollbar, 1, 2, 0, 1, gtk.FILL|gtk.SHRINK,
             gtk.FILL|gtk.SHRINK, 0, 0)
hScrollbar = gtk.HScrollbar(None)
hScrollbar.show()
table.attach(hScrollbar, 0, 1, 1, 2, gtk.FILL|gtk.SHRINK,
             gtk.FILL|gtk.SHRINK, 0, 0)
# tell the scrollbars to use the layout widget’s adjustments
vAdjust = layout.get_vadjustment()
vScrollbar.set_adjustment(vAdjust)
hAdjust = layout.get_hadjustment()
hScrollbar.set_adjustment(hAdjust)
layout.connect("drag_data_received", self.receiveCallback)
layout.drag_dest_set(gtk.DEST_DEFAULT_MOTION |
gtk.DEST_DEFAULT_HIGHLIGHT |
gtk.DEST_DEFAULT_DROP,
self.toCanvas, gtk.gdk.ACTION_COPY)
def addImage(self, xpm, xd, yd):
    hadj = self.layout.get_hadjustment()
vadj = self.layout.get_vadjustment()
style = self.window.get_style()
pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(
    self.window.window, style.bg[gtk.STATE_NORMAL], xpm)
image = gtk.Image()
image.set_from_pixmap(pixmap, mask)
button = gtk.Button()
button.add(image)
button.connect("drag_data_get", self.sendCallback)
button.drag_source_set(gtk.gdk.BUTTON1_MASK, self.fromImage,
                      gtk.gdk.ACTION_COPY)
button.show_all()
    # have to adjust for the scrolling of the layout - event location
    # is relative to the viewable not the layout size
    self.layout.put(button, int(xd+hadj.value), int(yd+vadj.value))
return
def sendCallback(self, widget, context, selection, targetType, eventTime):
    if targetType == self.TARGET_TYPE_TEXT:
        now = time.time()
        str = time.ctime(now)
        selection.set(selection.target, 8, str)
    elif targetType == self.TARGET_TYPE_PIXMAP:
        selection.set(selection.target, 8,
                       string.join(gtkxpm.gtk_xpm, '\n'))

def receiveCallback(self, widget, context, x, y, selection, targetType,
                    time):
    if targetType == self.TARGET_TYPE_TEXT:
        label = widget.get_children()[0]
        label.set_text(selection.data)
    elif targetType == self.TARGET_TYPE_PIXMAP:
        self.addImage(string.split(selection.data, '\n'), x, y)

def __init__(self):
    self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    self.window.set_default_size(300, 300)
    self.window.connect("destroy", lambda w: gtk.main_quit())
    self.window.show()
    layout = self.makeLayout()
    self.window.add(layout)

def main():
    gtk.main()

if __name__ == "__main__":
    DragNDropExample()
    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:

```python
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:

```python
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:

```python
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:

```python
window = gtk.Window(gtk.WINDOW_TOPLEVEL)
window.set_name("main window")
```

And then the style is defined in the rc file using:

```plaintext
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:

\[
fg[\text{STATE}] = \{ \text{Red}, \text{Green}, \text{Blue} \}
\]

Where \text{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:

\[
\text{font} = \"<\text{font name}>\"
\]

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.

\[
\text{style} "\text{main_button}" = "\text{button}"
\]

\[
\text{font} = \"-\text{adobe-helvetica-medium-r-normal-\---100-\---*-\---*-}\"
\]

\[
\text{bg}[\text{PRELIGHT}] = \{ 0.75, 0, 0 \}
\]

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

```bash
# 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.
# 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.
#
# 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[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 sets the padding around the window to the pixmap specified.
    bg_pixmap[NORMAL] = "warning.xpm"
}
```
# 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.

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:

```plaintext
<table>
<thead>
<tr>
<th>type</th>
<th>window</th>
<th>time</th>
</tr>
</thead>
</table>
```
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTHING</td>
<td>a special code to indicate a null event.</td>
</tr>
<tr>
<td>DELETE</td>
<td>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.</td>
</tr>
<tr>
<td>DESTROY</td>
<td>the window has been destroyed.</td>
</tr>
<tr>
<td>EXPOSE</td>
<td>all or part of the window has become visible and needs to be redrawn.</td>
</tr>
<tr>
<td>MOTION_NOTIFY</td>
<td>the pointer (usually a mouse) has moved.</td>
</tr>
<tr>
<td>BUTTON_PRESS</td>
<td>a mouse button has been pressed.</td>
</tr>
<tr>
<td>_2BUTTON_PRESS</td>
<td>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.</td>
</tr>
<tr>
<td>_3BUTTON_PRESS</td>
<td>a mouse button has been clicked 3 times in a short period of time. Note that each click also generates a BUTTON_PRESS event.</td>
</tr>
<tr>
<td>BUTTON_RELEASE</td>
<td>a mouse button has been released.</td>
</tr>
<tr>
<td>KEY_PRESS</td>
<td>a key has been pressed.</td>
</tr>
<tr>
<td>KEY_RELEASE</td>
<td>a key has been released.</td>
</tr>
<tr>
<td>ENTER_NOTIFY</td>
<td>the pointer has entered the window.</td>
</tr>
<tr>
<td>LEAVE_NOTIFY</td>
<td>the pointer has left the window.</td>
</tr>
<tr>
<td>FOCUS_CHANGE</td>
<td>the keyboard focus has entered or left the window.</td>
</tr>
<tr>
<td>CONFIGURE</td>
<td>the size, position or stacking order of the window has changed. Note that GTK+ discards these events for GDK_WINDOW_CHILD windows.</td>
</tr>
<tr>
<td>MAP</td>
<td>the window has been mapped.</td>
</tr>
<tr>
<td>UNMAP</td>
<td>the window has been unmapped.</td>
</tr>
<tr>
<td>PROPERTY_NOTIFY</td>
<td>a property on the window has been changed or deleted.</td>
</tr>
<tr>
<td>SELECTION_CLEAR</td>
<td>the application has lost ownership of a selection.</td>
</tr>
</tbody>
</table>
24.2. EVENT HANDLING

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION_REQUEST</td>
<td>another application has requested a selection.</td>
</tr>
<tr>
<td>SELECTION_NOTIFY</td>
<td>a selection has been received.</td>
</tr>
<tr>
<td>PROXIMITY_IN</td>
<td>an input device has moved into contact with a sensing surface (e.g. a touchscreen or graphics tablet).</td>
</tr>
<tr>
<td>PROXIMITY_OUT</td>
<td>an input device has moved out of contact with a sensing surface.</td>
</tr>
<tr>
<td>DRAG_ENTER</td>
<td>the mouse has entered the window while a drag is in progress.</td>
</tr>
<tr>
<td>DRAG_LEAVE</td>
<td>the mouse has left the window while a drag is in progress.</td>
</tr>
<tr>
<td>DRAG_MOTION</td>
<td>the mouse has moved in the window while a drag is in progress.</td>
</tr>
<tr>
<td>DRAG_STATUS</td>
<td>the status of the drag operation initiated by the window has changed.</td>
</tr>
<tr>
<td>DROP_START</td>
<td>a drop operation onto the window has started.</td>
</tr>
<tr>
<td>DROP_FINISHED</td>
<td>the drop operation initiated by the window has completed.</td>
</tr>
<tr>
<td>CLIENT_EVENT</td>
<td>a message has been received from another application.</td>
</tr>
<tr>
<td>VISIBILITY_NOTIFY</td>
<td>the window visibility status has changed.</td>
</tr>
<tr>
<td>NO_EXPOSE</td>
<td>indicates that the source region was completely available when parts of a drawable were copied. This is not very useful.</td>
</tr>
<tr>
<td>SCROLL ?</td>
<td>?</td>
</tr>
<tr>
<td>WINDOW_STATE ?</td>
<td>?</td>
</tr>
<tr>
<td>SETTING ?</td>
<td>?</td>
</tr>
</tbody>
</table>

- `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:
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
CHAPTER 24. SCRIBBLE, A SIMPLE...

24.2. EVENT HANDLING

pressure
x_tilt
y_tilt
state
is_hint
source
deviceid
x_root
y_root

BUTTON_PRESS
_2BUTTON_PRESS
_3BUTTON_PRESS
BUTTON_RELEASE
time
x
y
pressure
x_tilt
y_tilt
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

PROPERTY_NOTIFY
atom
time
state

SELECTION_CLEAR
SELECTION_REQUEST
SELECTION_NOTIFY
selection
target
property
requestor

# no additional attributes

297
24.2. 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 `STRUCTURE_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.Window` method:

```python
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:

---

24.2.1 Scribble - Event Handling
The event handlers are connected to the drawing_area by the following lines:

```python
# Signals used to handle backing pixmap
drawing_area.connect("expose_event", expose_event)
drawing_area.connect("configure_event", configure_event)

# Event signals
drawing_area.connect("motion_notify_event", motion_notify_event)
drawing_area.connect("button_press_event", button_press_event)

drawing_area.set_events(gtk.gdk.EXPOSURE_MASK |
                        gtk.gdk.LEAVE_NOTIFY_MASK |
                        gtk.gdk.BUTTON_PRESS_MASK |
                        gtk.gdk.POINTER_MOTION_MASK |
                        gtk.gdk.POINTER_MOTION_HINT_MASK)
```

The `button_press_event()` and `motion_notify_event()` event handlers in `scribblesimple.py` are:

```python
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:

```python
darea = gtk.DrawingArea()
```

A default size for the widget can be specified by calling:

```python
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:

```python
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.

```python
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     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):

```python
43 # Redraw the screen from the backing pixmap
44 def expose_event(widget, event):
45     x, y, width, height = event.area
46    
47     widget.window.draw_drawable(widget.get_style().fg_gc[gtk.STATE_NORMAL ←
48             ],
49     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:
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:

```python
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:

```python
# Draw a rectangle on the screen
def draw_brush(widget, x, y):
rect = (int(x-5), int(y-5), 10, 10)
pixmap.draw_rectangle(widget.get_style().black_gc, True, rect[0], rect[1], rect[2], rect[3])
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:

```python
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 require 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 maintenance 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 benefits 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 let's 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:

```python
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:

```python
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.

```python
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 let's 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.

```python
# A file called: gui.py
import pygtk
import gtk

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):
        

    def __init__(self):
```

304
CHAPTER 25. TIPS FOR WRITING PYGTK...

25.3. HOW TO SEPARATE CALLBACK...

# 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.

```python
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.

```python
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.

```python
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 etc... 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.

```python
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.

```python
#A file called: gui.py
#(updated version)
#(with multiple inheritance)

import pygtk
import gtk

from io import file_io # 1. Import Your Classes
from undo import undo #
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)
```

306
25.3. HOW TO SEPARATE CALLBACK

```python
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 Koch werner.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 Vanderboom mvboom42@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)
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)
ext-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, direction, data)
kill-line(editable, direction, data)
cut-clipboard(editable, data)
copy-clipboard(editable, data)
paste-clipboard(editable, data)

A.7 GtkNotebook

switch-page(notebook, 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

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pressed</td>
<td>button, data</td>
</tr>
<tr>
<td>released</td>
<td>button, data</td>
</tr>
<tr>
<td>clicked</td>
<td>button, data</td>
</tr>
<tr>
<td>enter</td>
<td>button, data</td>
</tr>
<tr>
<td>leave</td>
<td>button, data</td>
</tr>
</tbody>
</table>

### A.12 GtkItem

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>item, data</td>
</tr>
<tr>
<td>deselect</td>
<td>item, data</td>
</tr>
<tr>
<td>toggle</td>
<td>item, data</td>
</tr>
</tbody>
</table>

### A.13 GtkWindow

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-focus</td>
<td>window, widget, data</td>
</tr>
</tbody>
</table>

### A.14 GtkHandleBox

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>child-attached</td>
<td>handle_box, widget, data</td>
</tr>
<tr>
<td>child-detached</td>
<td>handle_box, widget, data</td>
</tr>
</tbody>
</table>

### A.15 GtkToggleButton

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toggled</td>
<td>toggle_button, data</td>
</tr>
</tbody>
</table>

### A.16 GtkMenuItem

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activate</td>
<td>menu_item, data</td>
</tr>
<tr>
<td>activate-item</td>
<td>menu_item, data</td>
</tr>
</tbody>
</table>

### A.17 GtkCheckMenuItem

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toggled</td>
<td>check_menu_item, data</td>
</tr>
</tbody>
</table>
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

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

# example scribblesimple.py

# GTK - The GIMP Toolkit
# Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh MacDonald
# Copyright (C) 2001-2002 John Finlay
#
# This library is free software; you can redistribute it and/or
# modify it under the terms of the GNU Library General Public
# License as published by the Free Software Foundation; either
# version 2 of the License, or (at your option) any later version.
#
# This library is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# Library General Public License for more details.
#
# You should have received a copy of the GNU Library General Public
# License along with this library; if not, write to the
# Free Software Foundation, Inc., 59 Temple Place - Suite 330,
# Boston, MA 02111-1307, USA.

import gtk

# Backing pixmap for drawing area
pixmap = None

def configure_event(widget, event):
    global pixmap

    x, y, width, height = widget.get_allocation()
    pixmap = gtk.gdk.Pixmap(widget.window, width, height)
    pixmap.draw_rectangle(widget.get_style().white_gc,
                          True, 0, 0, width, height)
    return True

def expose_event(widget, event):
    x, y, width, height = event.area
    widget.window.draw_drawable(widget.get_style().fg_gc[gtk.STATE_NORMAL],
                                 pixmap, x, y, x, y, width, height)
```
APPENDIX B. CODE EXAMPLES

B.1. SCRIBBLESIMPLE.PY

```python
def draw_brush(widget, x, y):
    rect = (x - 5, y - 5, 10, 10)
    pixmap.draw_rectangle(widget.get_style().black_gc, True, rect[0], rect[1], rect[2], rect[3])
    widget.queue_draw_area(rect[0], rect[1], rect[2], rect[3])

def button_press_event(widget, event):
    if event.button == 1 and pixmap != None:
        draw_brush(widget, event.x, event.y)
    return True

def motion_notify_event(widget, event):
    if event.is_hint:
        x, y, state = event.window.get_pointer()
    else:
        x = event.x
        y = event.y
        state = event.state

    if state & gtk.gdk.BUTTON1_MASK and pixmap != None:
        draw_brush(widget, x, y)
    return True

def main():
    window = gtk.Window(gtk.WINDOW_TOPLEVEL)
    window.set_name("Test Input")
    vbox = gtk.VBox(False, 0)
    window.add(vbox)
    vbox.show()

    window.connect("destroy", gtk.mainquit)

    # Create the drawing area
    drawing_area = gtk.DrawingArea()
    drawing_area.set_size_request(200, 200)
    vbox.pack_start(drawing_area, True, True, 0)
    drawing_area.show()

    # Signals used to handle backing pixmap
    drawing_area.connect("expose_event", expose_event)
    drawing_area.connect("configure_event", configure_event)

    # Event signals
    drawing_area.connect("motion_notify_event", motion_notify_event)
    drawing_area.connect("button_press_event", button_press_event)

    drawing_area.set_events(gtk.gdk.EXPOSURE_MASK
        | gtk.gdk.LEAVE_NOTIFY_MASK
        | gtk.gdk.BUTTON_PRESS_MASK
        | gtk.gdk.POINTER_MOTION_MASK
        | gtk.gdk.POINTER_MOTION_HINT_MASK)

    # .. And a quit button
    button = gtk.Button("Quit")
    vbox.pack_start(button, False, False, 0)
    button.connect_object("clicked", lambda w: w.destroy(), window)
    button.show()
```

322
window.show()  
gtk.main()  
return 0  
if __name__ == "__main__":  
    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)
APPENDIX C.  CHANGELOG

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>
APPENDIX C. CHANGELOG

* 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.
APPENDIX C. CHANGEOLOG

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) (Pixmap) (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.

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>


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>
APPENDIX C. CHANGEOLOG

* 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)
APPENDIX C. CHANGEOLOG

* 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>


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.