Tutorial de PyGTK 2.0 versión 2.3

John Finlay, Rafael Villar Burke, Lorenzo Gil Sánchez,
Iñigo Serna, y Fernando San Martín Woerner

7 de octubre de 2012

Tutorial de PyGTK 2.0 versión 2.3

by John Finlay, Rafael Villar Burke, Lorenzo Gil Sánchez, Iñigo Serna, y Fernando San Martín Woerner

Índice general

1. Introducción **1**
1.1. Exploración de PyGTK . . . 2

2. Primeros Pasos **5**
2.1. Hola Mundo en PyGTK . . . 7
2.2. Teoría de Señales y Retrollamadas . . . 9
2.3. Eventos . . . 10
2.4. Hola Mundo Paso a Paso . . . 12

3. Avanzando **15**
3.1. Más sobre manejadores de señales . . . 15
3.2. Un Hola Mundo Mejorado . . . 15

4. Empaquetado de Controles **19**
4.1. Teoría de Cajas Empaquetadoras . . . 19
4.2. Las Cajas en detalle . . . 19
4.3. Programa de Ejemplo de Empaquetado . . . 21
4.4. Uso de Tablas para el Empaquetado . . . 26
4.5. Ejemplo de Empaquetado con Tablas . . . 27

5. Perspectiva General de Controles **31**
5.1. Jerarquía de Controles . . . 31
5.2. Controles sin Ventana . . . 34

6. El Control de Botón **35**
6.1. Botones Normales . . . 35
6.2. Botones Biestado (Toggle Buttons) . . . 38
6.3. Botones de Activación (Check Buttons) . . . 40
6.4. Botones de Exclusión Mútua (Radio Buttons) . . . 42

7. Ajustes **45**
7.1. Creación de un Ajuste . . . 45
7.2. Utilización de Ajustes de la Forma Fácil . . . 46
7.3. Interioridades de un Ajuste . . . 46

8. Controles de Rango **49**
8.1. Barras de Desplazamiento . . . 49
8.2. Controles de Escala . . . 49
8.2.1. Creación de un Control de Escala . . . 49
8.2.2. Métodos y Señales (bueno, al menos métodos) . . . 50
8.3. Métodos Comunes de los Rangos . . . 50
8.3.1. Establecimiento de la Política de Actualización . . . 50
8.3.2. Obtención y Cambio de Ajustes . . . 51
8.4. Atajos de Teclas y Ratón . . . 51
8.5. Ejemplo de Control de Rango . . . 51

9. Miscelánea de Controles **57**
9.1. Etiquetas . . . 57
9.2. Flechas (Arrow) . . . 60
9.3. El Objeto Pistas (Tooltip) . . . 62
9.4. Barras de Progreso (ProgressBar) . . . 63
9.5. Diálogos . . . 67
9.6. Imágenes . . . 68
9.6.1. Pixmaps . . . 70
9.7. Reglas . . . 76

9.8. Barras de Estado . . . 79
9.9. Entradas de Texto (Entry) . . . 81
9.10. Botones Aumentar/Disminuir . . . 84
9.11. Lista Desplegable (Combo) . . . 89
9.12. Calendario . . . 90
9.13. Selección de Color . . . 96
9.14. Selectores de Fichero . . . 100
9.15. Diálogo de Selección de Fuentes . . . 102

10. Controles Contenedores **105**
10.1. La Caja de Eventos (EventBox) . . . 105
10.2. El control Alineador . . . 106
10.3. Contenedor Fijo (Fixed) . . . 107
10.4. Contenedor de Disposición (Layout) . . . 109
10.5. Marcos (Frame) . . . 111
10.6. Marcos Proporcionales (AspectFrame) . . . 113
10.7. Controles de Panel (HPaned y VPaned) . . . 115
10.8. Vistas (Viewport) . . . 118
10.9. Ventanas de Desplazamiento (ScrolledWindow) . . . 118
10.10Cajas de Botones (ButtonBoxes) . . . 121
10.11Barra de Herramientas (Toolbar) . . . 124
10.12Fichas (Notebook) . . . 129
10.13Elementos incrustables y puntos de conexión (Plugs y Sockets) . . . 133
10.13.1.Elementos incrustables (Plugs) . . . 133
10.13.2.Puntos de Conexión (Sockets) . . . 134

11. Control Menú **137**
11.1. Creación Manual de Menús . . . 137
11.2. Ejemplo de Menú Manual . . . 139
11.3. Uso de la Factoria de Elementos . . . 141
11.4. Ejemplo de Factoria de Elementos - ItemFactory . . . 141

12. Área de Dibujo **145**
12.1. Contexto Gráfico . . . 145
12.2. Métodos de Dibujo . . . 149

13. Control de Vista de Texto **157**
13.1. Perspectiva general de la Vista de Texto . . . 157
13.2. Vistas de Texto . . . 157
13.3. Buffers de Texto . . . 163
13.3.1. Información de estado de un Buffer de Texto . . . 163
13.3.2. Creación de Iteradores de Texto . . . 164
13.3.3. Inserción, Obtención y Eliminación de Texto . . . 164
13.3.4. Marcas de Texto (TextMark) . . . 165
13.3.5. Creación y Uso de Etiquetas de Texto . . . 166
13.3.6. Inserción de Imágenes y Controles . . . 167
13.4. Iteradores de Texto . . . 168
13.4.1. Atributos de los Iteradores de Texto . . . 168
13.4.2. Atributos de Texto de un Iterador de Texto . . . 168
13.4.3. Copiar un Iterador de Texto . . . 169
13.4.4. Recuperar Texto y Objetos . . . 169
13.4.5. Comprobar Condiciones en un Iterador de Texto . . . 169
13.4.6. Comprobar la posición en un Texto . . . 170
13.4.7. Movimiento a través del Texto . . . 171
13.4.8. Moverse a una Posición Determinada . . . 172
13.4.9. Búsqueda en el Texto . . . 172
13.5. Marcas de Texto . . . 172
13.6. Etiquetas de Texto y Tablas de Etiquetas . . . 173
13.6.1. Etiquetas de Texto . . . 173

13.6.2. Tablas de Etiquetas de Texto . . . 175
13.7. Un ejemplo de Vista de Texto . . . 176

14. Control de Vista de Árbol (TreeView) 179
14.1. Introducción . . . 179
14.2. La Interfaz y Almacén de Datos TreeModel . . . 182
14.2.1. Introducción . . . 182
14.2.2. Creación de Objetos TreeStore (árbol) y ListStore (lista) . . . 183
14.2.3. Cómo referirse a las filas de un modelo TreeModel . . . 183
14.2.3.1. Caminos de árbol (Tree Paths) . . . 184
14.2.3.2. Iteradores TreeIter . . . 184
14.2.3.3. Referencias persistentes a filas (TreeRowReferences) . . . 185
14.2.4. Adición de filas . . . 185
14.2.4.1. Adición de filas a un almacén de datos del tipo ListStore . . . 185
14.2.4.2. Adición de filas a un almacén de datos del tipo TreeStore . . . 186
14.2.4.3. Almacenes de datos de gran tamaño . . . 186
14.2.5. Eliminación de Filas . . . 187
14.2.5.1. Eliminación de filas de un almacén ListStore . . . 187
14.2.5.2. Eliminación de filas de un almacén TreeStore . . . 187
14.2.6. Gestión de los datos de las filas . . . 187
14.2.6.1. Establecemiento y obtención de los valores de los datos . . . 187
14.2.6.2. Reorganización de filas en almacenes ListStore . . . 188
14.2.6.3. Reorganización de filas en almacenes TreeStore . . . 189
14.2.6.4. Gestión de múltiples filas . . . 189
14.2.7. Soporte del protocolo de Python . . . 191
14.2.8. Señales de TreeModel . . . 192
14.2.9. Ordenación de filas de modelos TreeModel . . . 193
14.2.9.1. La interfaz TreeSortable . . . 193
14.2.9.2. Clasificación en almacenes ListStore y TreeStore . . . 194
14.3. TreeViews (Vistas de árbol) . . . 194
14.3.1. Creación de un TreeView (vista de árbol) . . . 194
14.3.2. Obtención y establecimiento del Modelo de un TreeView . . . 194
14.3.3. Definición de las propiedades de un TreeView . . . 195
14.4. Visualizadores de Celda (CellRenderer) . . . 196
14.4.1. Introducción . . . 196
14.4.2. Tipos de Visualizadores CellRenderer . . . 197
14.4.3. Propiedade de un CellRenderer . . . 197
14.4.4. Atributos de un CellRenderer . . . 199
14.4.5. Función de Datos de Celda . . . 200
14.4.6. Etiquetas de Marcado en CellRendererText . . . 202
14.4.7. Celdas de Texto Editables . . . 203
14.4.8. Celdas Biestado Activables . . . 204
14.4.9. Programa de Ejemplo de Celda Editable and Activable . . . 204
14.5. TreeViewColumns (columnas de vista de árbol) . . . 207
14.5.1. Creación de TreeViewColumns (columnas de vista de árbol) . . . 207
14.5.2. Gestión de los CellRenderers (Intérpretes de celda) . . . 207
14.6. Manipulación de TreeViews . . . 208
14.6.1. Gestión de las Columnas . . . 208
14.6.2. Expansión y Contracción de Filas Hijas . . . 209
14.7. Señales de TreeView . . . 209
14.8. Selecciones TreeSelections . . . 210
14.8.1. Obtención de TreeSelection . . . 210
14.8.2. Modos de una selección TreeSelection . . . 210
14.8.3. Obtención de la Selección . . . 210
14.8.4. Uso de una Función de TreeSelection . . . 211
14.8.5. Selección y Deselección de Filas . . . 212
14.9. Arrastrar y Soltar en TreeView . . . 212
14.9.1. Reordenación mediante Arrastrar y Soltar . . . 212
14.9.2. Arrastar y Soltar Externo . . . 212

14.9.3. Ejemplo de Arrastrar y Soltar en TreeView . . . 214
14.10TreeModelSort y TreeModelFilter . . . 217
14.10.1.TreeModelSort (Modelo de Árbol Ordenado) . . . 217
14.10.2.TreeModelFilter (Modelo de árbol filtrado) . . . 218
14.11El Modelo de Árbol Genérico (GenericTreeModel) . . . 221
14.11.1.Visión general de GenericTreeMode . . . 221
14.11.2.La Interfaz GenericTreeModel . . . 222
14.11.3.Adición y Eliminación de Filas . . . 225
14.11.4.Gestión de Memoria . . . 227
14.11.5.Otras Interfaces . . . 228
14.11.6.Utilización de GenericTreeModel . . . 228
14.12El Visualizador de Celda Genérico (GenericCellRenderer) . . . 229

15. Nuevos Controles de PyGTK 2.2 **231**
15.1. Portapapeles (Clipboard) . . . 231
15.1.1. Creación de un objeto Clipboard (Portapapeles) . . . 231
15.1.2. Utilización de Clipboards con elementos Entry, Spinbutton y TextView . . . 231
15.1.3. Incorporación de datos en un portapapeles . . . 232
15.1.4. Obtención de los Contenidos del Portapapeles . . . 233
15.1.5. Ejemplo de Portapapeles . . . 234

16. Nuevos Controles de PyGTK 2.4 **235**
16.1. Objetos de Acción (Action) y Grupo de Acciones (ActionGroup) . . . 236
16.1.1. Acciones (Actions) . . . 236
16.1.1.1. Creación de acciones . . . 236
16.1.1.2. Uso de acciones . . . 237
16.1.1.3. Creación de Controles intermedios (Proxy) . . . 238
16.1.1.4. Propiedades de Acción . . . 241
16.1.1.5. Acciones y Aceleradores . . . 242
16.1.1.6. Acciones Conmutables . . . 243
16.1.1.7. Acciones con Exclusión . . . 243
16.1.1.8. Un ejemplo de Acciones . . . 244
16.1.2. Grupos de Acciones (ActionGroups) . . . 244
16.1.2.1. Creación de grupos de acciones (ActionGroups) . . . 244
16.1.2.2. Adición de Acciones . . . 245
16.1.2.3. Obtención de iconos . . . 246
16.1.2.4. Control de las Acciones . . . 246
16.1.2.5. Un ejemplo de grupo de acciones (ActionGroup) . . . 246
16.1.2.6. Señales de los grupos de acciones (ActionGroup) . . . 247
16.2. Controles de Lista Desplegable (ComboBox) y Lista Desplegable con Entrada (ComboBox-Entry) . . . 247
16.2.1. Controles ComboBox . . . 247
16.2.1.1. Uso Básico de ComboBox . . . 247
16.2.1.2. Uso Avanzado de ComboBox . . . 249
16.2.2. Controles ComboBoxEntry . . . 251
16.2.2.1. Uso Básico de ComboBoxEntry . . . 251
16.2.2.2. Uso Avanzado de ComboBoxEntry . . . 252
16.3. Controles Botón de Color y de Fuente (ColorButton y FontButton) . . . 253
16.3.1. Control Botón de Color (ColorButton) . . . 253
16.3.2. Control Botón de Fuente (FontButton) . . . 254
16.4. Controles de Entrada con Completado (EntryCompletion) . . . 256
16.5. Controles de Expansión (Expander) . . . 258
16.6. Selecciones de Archivos mediante el uso de Controles basados en el Selector de Archivos FileChooser . . . 259
16.7. El gestor de Interfaces de Usuario UIManager . . . 261
16.7.1. Perspectiva general . . . 261
16.7.2. Creación de un gestor UIManager . . . 262
16.7.3. Adición y Eliminación de Grupos de Acciones (ActionGroups) . . . 262
16.7.4. Descripciones de la Interfaz de Usuario . . . 263

16.7.5. Adición y Eliminación de Descripciones de Interfaz de Usuario . . . 264
16.7.6. Acceso a los Controles de la Interfaz de Usuario . . . 265
16.7.7. Ejemplo sencillo de Gestor de Interfaz UIManager . . . 266
16.7.8. Combinación de Descripciones de Interfaz de Usuario . . . 267
16.7.9. Señales de UIManager . . . 269

17. Controles sin documentar **271**
17.1. Etiqueta de Aceleración (Atajo) . . . 271
17.2. Menú de Opciones . . . 271
17.3. Elementos de Menú . . . 271
17.3.1. Elemento de Menú de Activación . . . 271
17.3.2. Elemento de Menú de Exclusión Mútua . . . 271
17.3.3. Elemento de Menú de Separación . . . 271
17.3.4. Elemento de Menú de Cascada . . . 271
17.4. Curvas . . . 271
17.5. Diálogo de Mensaje . . . 271
17.6. Curva Gamma . . . 271

18. Establecimiento de Atributos de Controles **273**
18.1. Métodos de Banderas de los Controles . . . 273
18.2. Métodos de Visualización de Controles . . . 274
18.3. Atajos de Teclado de los Controles . . . 274
18.4. Métodos relacionados con el Nombre de los Controles . . . 275
18.5. Estilo de los Controles . . . 275

19. Temporizadores, Entrada/Salida y Funciones de Inactividad **279**
19.1. Temporizadores . . . 279
19.2. Monitorizar la Entrada/Salida . . . 279
19.3. Funciones de Inactividad . . . 280

20. Procesamiento Avanzado de Eventos y Señales **281**
20.1. Métodos de Señales . . . 281
20.1.1. Conectar y Desconectar Manejadores de Señal . . . 281
20.1.2. Bloqueo y Desbloqueo de Manejadores de Señal . . . 282
20.1.3. Emisión y Parada de Señales . . . 282
20.2. Emisión y Propagación de Señales . . . 282

21. Tratamiento de Selecciones **283**
21.1. Descripción General de la Selección . . . 283
21.2. Recuperar la Selección . . . 283
21.3. Proporcionar la Selección . . . 287

22. Arrastrar y Soltar **291**
22.1. Descripción General de Arrastrar y Soltar . . . 291
22.2. Propiedades de Arrastrar y Soltar . . . 292
22.3. Métodos de Arrastrar y Soltar . . . 292
22.3.1. Configuración del Control Origen . . . 292
22.3.2. Señales en el Control Fuente . . . 293
22.3.3. Configuración de un Control Destino . . . 293
22.3.4. Señales en el Control Destino . . . 294

23. Ficheros rc de GTK+ **299**
23.1. Funciones para Ficheros rc . . . 299
23.2. Formato de los Ficheros rc de GTK+ . . . 300
23.3. Ejemplo de fichero rc . . . 301

24. Scribble: Un Ejemplo Sencillo de Programa de Dibujo **305**
24.1. Perspectiva General de Scribble . . . 305
24.2. Manejo de Eventos . . . 305
24.2.1. Scribble - Manejo de Eventos . . . 310
24.3. El Control del Área de Dibujo, y Dibujar . . . 312

25. Trucos para Escribir Aplicaciones PyGTK **315**
25.1. El usario debería manejar la interfaz, no al contrario . . . 315
25.2. Separa el modelo de datos de la interfaz . . . 315
25.3. Cómo separar los Métodos de Retrollamada de los Manejadores de Señal . . . 316
25.3.1. Introducción . . . 316
25.3.2. Herencia . . . 316
25.3.3. Herencia aplicada a PyGTK . . . 316

26. Contribuir **321**

27. Créditos **323**
27.1. Créditos Original de GTK+ . . . 323

28. Copyright del Tutorial y Nota de Permisos **325**

A. Señales de GTK **327**
A.1. gtk.Object . . . 327
A.2. gtk.Widget . . . 327
A.3. GtkData . . . 329
A.4. gtk.Container . . . 329
A.5. gtk.Calendar . . . 329
A.6. gtk.Editable . . . 329
A.7. gtk.Notebook . . . 330
A.8. gtk.List . . . 330
A.9. gtk.MenuShell . . . 330
A.10.gtk.Toolbar . . . 330
A.11.gtk.Button . . . 331
A.12.gtk.Item . . . 331
A.13.gtk.Window . . . 331
A.14.gtk.HandleBox . . . 331
A.15.gtk.ToggleButton . . . 331
A.16.gtk.MenuItem . . . 331
A.17.gtk.CheckMenuItem . . . 331
A.18.gtk.InputDialog . . . 332
A.19.gtk.ColorSelection . . . 332
A.20.gtk.StatusBar . . . 332
A.21.gtk.Curve . . . 332
A.22.gtk.Adjustment . . . 332

B. Ejemplos de Código **333**
B.1. scribblesimple.py . . . 333

C. ChangeLog **337**

Índice alfabético **345**

Índice de figuras

2. Primeros Pasos
2.1. Ventana Simple PyGTK 6
2.2. Programa de ejemplo: Hola Mundo 8

3. Avanzando
3.1. Ejemplo mejorado de Hola Mundo 17

4. Empaquetado de Controles
4.1. Empaquetado: Cinco variaciones 20
4.2. Empaquetado con Spacing y Padding 21
4.3. Empaquetado con pack_end() 21
4.4. Empaquetado haciendo uso de una Tabla 28

6. El Control de Botón
6.1. Botón con Pixmap y Etiqueta 36
6.2. Ejemplo de Botón Biestado 39
6.3. Ejemplo de Botón de Activación 41
6.4. Ejemplo de Botones de Exclusión Mútua 43

8. Controles de Rango
8.1. Ejemplo de Controles de Rango 52

9. Miscelánea de Controles
9.1. Ejemplos de Etiquetas 58
9.2. Ejemplos de Botones con Flechas 60
9.3. Ejemplo de Pistas 62
9.4. Ejemplo de Barra de Progreso 65
9.5. Ejemplo de Imágenes en Botones 69
9.6. Ejemplo de Pixmap en un Botón 72
9.7. Ejemplo de Ventana con Forma 73
9.8. Ejemplo de Reglas 78
9.9. Ejemplo de Barra de Estado 80
9.10. Ejemplo de Entrada 82
9.11. Ejemplo de Botón Aumentar/Disminuir 86
9.12. Ejemplo de Calendario 92
9.13. Ejemplo de Diálogo de Selección de Color 98
9.14. Ejemplo de Selección de Ficheros 101
9.15. Diálogo de Selección de Fuentes 102

10. Controles Contenedores
10.1. Ejemplo de Caja de Eventos 105
10.2. Ejemplo de Fijo 107
10.3. Ejemplo de Disposición 110
10.4. Ejemplo de Marco 112
10.5. Ejemplo de Marco Proporcional 114
10.6. Ejemplo de Panel 116
10.7. Ejemplo de Ventana de Desplazamiento 119
10.8. Ejemplo de Barra de Herramientas 129

10.9. Ejemplo de Fichas 131

11. Control Menú
11.1. Ejemplo de Menú 139
11.2. Ejemplo de Factoria de Elementos 142

12. Área de Dibujo
12.1. Ejemplo de Área de Dibujo 152

13. Control de Vista de Texto
13.1. Ejemplo básico de Vista de Texto 160
13.2. Ejemplo de Vista de Texto 177

14. Control de Vista de Árbol (TreeView)
14.1. Programa elemental de ejemplo de TreeView 182
14.2. TreeViewColumns con CellRenderers 196
14.3. Función de Datos de Celda 201
14.4. Ejemplo de Listado de Archivos Utilizando Funciones de Datos de Celda 202
14.5. Etiquetas de Marcado para CellRendererText 203
14.6. Celdas Editables y Activables 207
14.7. Flecha de Expansión en la segunda Columna 209
14.8. Ejemplo de Arrastrar y Soltar en TreeView 216
14.9. Ejemplo de TreeModelSort 218
14.10Ejemplo de Visibilidad en TreeModelFilter 220
14.11Programa de Ejemplo de Modelo de Árbol Genérico 225

15. Nuevos Controles de PyGTK 2.2
15.1. Programa de ejemplo de Portapapeles 234

16. Nuevos Controles de PyGTK 2.4
16.1. Ejemplo Simple de Acción 238
16.2. Ejemplo Básico de Acción 241
16.3. Ejemplo de Acciones 244
16.4. Ejemplo de ActionGroup 247
16.5. ComboBox Básica 248
16.6. ComboBox con una Disposición Asociada 250
16.7. ComboBoxEntry Básica 252
16.8. Ejemplo de Botón de Color - ColorButton 254
16.9. Ejemplo de Botón de Fuente - FontButton 256
16.10Entrada con Completado (EntryCompletion) 257
16.11Control de Expansión 259
16.12Ejemplo de Selección de Archivos 260
16.13Programa sencillo de Gestor de Interfaz de Usuario UIManager 267
16.14Ejemplo UIMerge 269

21. Tratamiento de Selecciones
21.1. Ejemplo de Obtención de la Selección 285
21.2. Ejemplo de Fijar la Selección 288

22. Arrastrar y Soltar
22.1. Ejemplo de Arrastrar y Soltar 295

24. Scribble: Un Ejemplo Sencillo de Programa de Dibujo
24.1. Ejemplo de Programa de Dibujo Scribble 305
24.2. Ejemplo sencillo - Scribble 311

Índice de cuadros

22. Arrastrar y Soltar
22.1. Señales del Control Fuente 293
22.2. Señales del Control Destino 295

Resumen

Este tutorial describe el uso del módulo de Python PyGTK.

Capítulo 1

Introducción

PyGTK 2.0 es un conjunto de módulos que componen una interfaz Python para GTK+ 2.0. En el resto de este documento cuando se menciona PyGTK se trata de la versión 2.0 o posterior de PyGTK, y en el caso de GTK+, también a su versión 2.0 y siguientes. El sitio web de referencia sobre PyGTK es www.pygtk.org. El autor principal de PyGTK es:

- James Henstridge james@daa.com.au

que es ayudado por los desarrolladores citados en el archivo AUTHORS de la distribución PyGTK y por la comunidad PyGTK.

Python es un lenguaje de programación interpretado, ampliable y orientado a objetos que se distribuye con un amplio conjunto de módulos que permiten el acceso a un gran número de servicios del sistema operativo, servicios de internet (como HTML, XML, FTP, etc.), gráficos (incluidos OpenGL, TK, etc.), funciones de manejo de cadenas, servicios de correo (IMAP, SMTP, POP3, etc.), multimedia (audio, JPEG) y servicios de criptografía. Existen además multitud de módulos proporcionados por terceros que añaden otros servicios. Python se distribuye bajo términos similares a los de la licencia GPL y está disponible para los sistemas operativos Linux, Unix, Windows y Macintosh. En www.python.org hay más informacion disponible sobre Python. Su autor principal es:

- Guido van Rossum guido@python.org

GTK+ (GIMP Toolkit) es una librería que permite crear interfaces gráficas de usuario. Se distribuye bajo la licencia LGPL, por lo que posibilita el desarrollo de software abierto, software libre, e incluso software comercial no libre que use GTK sin necesidad de pagar licencias o derechos.

Se le conoce como el toolkit de GIMP porque originalmente se escribió para desarrollar el Programa de Manipulación de Imágenes de GNU GIMP, pero GTK+ se usa ya en numerosos proyectos de software, incluído el proyecto de escritorio GNOME (Entorno de Modelo de Objetos orientados a Red). GTK+ está diseñada sobre GDK (Kit de Dibujo de GIMP) que, básicamente, es una abstracción de las funciones de bajo nivel que acceden al sistema de ventanas (Xlib en el caso del sistema de ventanas X). Los principales autores de GTK+ son:

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

Actualmente GTK+ es mantenida por:

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

GTK+ es fundamentalmente un interfaz orientada a objetos para programadores de aplicaciones (API). Aunque está escrita completamente en C, está implementada usando la idea de clases y funciones de retrollamada (punteros a función).

Existe un tercer componente, llamado Glib, que contiene diversas funciones que reemplazan algunas llamadas estandard, así como funciones adicionales para manejar listas enlazadas, etc. Las funciones de reemplazo se usan para aumentar la portabilidad de GTK+ ya que algunas de las funciones que

implementa no están disponibles o no son estándar en otros UNIX, tales como `g_strerror`(). Otras incluyen mejoras a las versiones de libc, tales como `g_malloc`, que posee capacidades de depuración mejoradas.

Desde su versión 2.0, GLib incluye el sistema de tipos que forma la base de la jerarquía de clases de GTK+, el sistema de señales usado en ésta, una API de hebras que abstrae las diferentes APIs nativas para programación multihilo en las diversas plataformas, y la capacidad de cargar módulos.

Como último componente, GTK+ usa la librería Pango para la salida de texto internacionalizado.

Este tutorial describe la interfaz de Python con GTK+ y está basado en el tutorial de GTK+ 2.0 escrito por Tony Gale e Ian Main. En él se intenta documentar en la medida posible todo PyGTK, pero en ningún caso es completo.

Este tutorial presupone algún conocimiento previo de Python, así de cómo se crean y ejecutan programas escritos en Python. Si no se está familiarizado con Python, es recomendable previamente la lectura del Tutorial de Python. Este tutorial no presupone ningún conocimiento previo sobre GTK+ y si se utiliza PyGTK para aprender GTK+ sería interesante recibir comentarios acerca de este tutorial, y qué aspectos resultan problemáticos. Este tutorial no describe cómo compilar o instalar Python, GTK+ o PyGTK.

Este tutorial está basado en:

- GTK+ desde la versión 2.0 hasta la 2.4
- Python 2.2
- PyGTK desde la versión 2.0 hasta la 2.4

Los ejemplos se escribieron y probaron en una RedHat 9.0.

Este documento está "en obras". Por favor, consúltese www.pygtk.org para localizar las actualizaciones.

Me gustaría mucho escuchar los problemas que aparezcan al aprender PyGTK a partir de este documento, y se aprecian los comentarios sobre cómo mejorarlo. Por favor, mira la sección Cómo Contribuir para más información. Si encuentra fallos, por favor rellene un informe de fallo en bugzilla.gnome.org en el proyecto pygtk. Para ello, la información que se encuentra en www.pygtk.org sobre Bugzilla puede resultar de gran utilidad.

El manual de referencia de PyGTK 2.0 se encuentra disponible en http://www.pygtk.org/pygtkreference. Dicho manual describe detalladamente las clases de PyGTK y, por ahora, sólo se encuentra disponible en inglés.

La página web de PyGTK (www.pygtk.org) contiene otros recursos útiles para aprender PyGTK, incluido un enlace a la extensa FAQ (Lista de Preguntas Frecuentes, también en inglés únicamente), y otros artículos y cursos, así como una lista de correo activa y un canal IRC (consúltese www.pygtk.org para los detalles).

1.1. Exploración de PyGTK

Johan Dahlin escribió un pequeño programa en Python (pygtkconsole.py) que se ejecuta en Linux y permite la exploración interactiva de PyGTK. Ese progama proporciona una interfaz de intérprete interactivo al estilo de la de Python, que se comunica con un proceso hijo que ejecuta los comandos introducidos. Los módulos PyGTK se cargan por defecto al arrancar el programa. Un ejemplo simple de sesión es:

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

```
¡Hola Mundo!
¡Hola Mundo!

>>> b.set_label("Hola a todos")
>>>
```

En este ejemplo se crea una ventana que contiene un botón que imprime un mensaje ('¡Hola Mundo!') cuando se hace clic en él. El programa permite probar así fácilmente los diversos controles de GTK+ y sus interfaces PyGTK.

También es útil el programa desarrollado por Brian McErlean para la receta de Activestate 65109 junto con algunas modificaciones para que funcione con PyGTK 2.X. En este curso lo llamamos gpython.py y funciona de forma parecida al programa pygtkconsole.py.

NOTA

Estos dos programas no funcionan en Microsoft Windows porque necesitan funciones específicas de Unix.

Capítulo 2

Primeros Pasos

Para empezar nuestra introducción a PyGTK, comenzaremos con el programa más simple posible. Este programa (base.py) creará una ventana de 200x200 píxeles y no es posible salir de él excepto terminando el proceso desde la consola.

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

Se puede ejecutar el programa anterior escribiendo en la línea de órdenes:

```
python base.py
```

Si base.py es hecho ejecutable y se puede encontrar en la variable `PATH`, es posible ejecutarlo usando:

```
base.py
```

En este caso, la línea 1 pedirá al intérprete de Python que ejecute base.py. Las líneas 5-6 ayudan a diferenciar entre las distintas versiones de PyGTK que puedan estar instaladas en el equipo. Estas líneas indican que se desea usar la versión 2.0 de PyGTK, que comprende todas las versiones de PyGTK con 2 como número principal. Ello impide que el programa utilice versiones anteriores de PyGTK, en caso de que se encuentren instaladas en el sistema. Las líneas 18-20 comprueban si la variable `__name__` es `"__main__"`, lo cual indica que el programa está siendo ejecutado directamente por python y no está siendo importado en un intérprete Python. En el primer caso el programa crea una nueva instancia de la clase `Base` y guarda una referencia a ella en la variable `base`. Después llama la función `main()` para iniciar el bucle de procesamiento de eventos de GTK.

Una ventana similar a Figura 2.1 debería aparecer en tu pantalla.

Figura 2.1 Ventana Simple PyGTK

La primera línea permite al programa base.py ser invocado desde una consola Linux o Unix asumiendo que **python** se encuentre en el `PATH`. Esta línea aparecerá como primera línea en todos los programas de ejemplo.

Las líneas 5-7 importan el módulo PyGTK 2 e inicializan el entorno GTK+. El módulo PyGTK define las interfaces Python de las funciones GTK+ que se usarán en el programa. Para quienes estén familiarizados con GTK+ hay que advertir que la inicialización incluye la llamada a la función `gtk_init`(). También se configuran algunas cosas por nosotros, tales como el visual por defecto, el mapa de colores, manejadores de señales predeterminados. Asimismo comprueba los argumentos que se pasan al programa desde la línea de comandos, en busca de alguno entre:

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

En este caso, los borra de la lista de argumentos y deja los no coincidentes que no reconoce para que el programa lo procese o ignore. El anterior conjunto de argumentos son los que aceptan de forma estándar todos los programas GTK+.

Las líneas 9-15 definen una clase de Python llamada `Base` que define un método de inicialización de instancia __init__(). La función __init__() crea una ventana de nivel superior (línea 11) y ordena a GTK+ que la muestre (línea 12). La `gtk.Window` se crea en la línea 11 con el argumento `gtk.WINDOW_TOPLEVEL` que indica que se desea una ventana sometida a las decoraciones y posicionamiento del manejador de ventanas. En vez de crear una ventana de tamaño 0x0, una ventana sin hijos tiene un tamaño predeterminado de 200x200 de forma que se pueda manipular.

Las líneas 14-15 definen el método main() que llama a la función PyGTK main(), que invoca el bucle principal de procesamiento de eventos de GTK+ para manejar eventos de ratón y de teclado, así como eventos de ventana.

Las líneas 18-20 permiten al programa comenzar automáticamente si es llamado directamente o pasado como argumento al intérprete de Python. En estos casos, el nombre de programa que hay en la variable `__name__` será la cadena `"__main__"` y el código entre las líneas 18-20 se ejecutará. Si el programa se carga en un intérprete de Python en ejecución, las líneas 18-20 no serán ejecutadas.

La línea 19 crea una instancia de la clase `Base` llamada *`base`*. Crea una `gtk.Window` y la muestra como resultado.

La línea 20 llama al método main() de la clase `Base`, la cual comienza el bucle de procesamiento de eventos de GTK+. Cuando el control llega a este punto, GTK+ se dormirá a la espera de eventos de las X (como pulsaciones de teclas o botones), alarmas, o notificaciones de entrada/salida de ficheros. En el ejemplo, sin embargo, los eventos son ignorados.

2.1. Hola Mundo en PyGTK

Ahora seguimos con un programa con un control (un botón). Es la versión PyGTK del clásico programa hola mundo (helloworld.py).

```
 1  #!/usr/bin/env python
 2
 3  # ejemplo helloworld.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class HelloWorld:
10
11      # Esta es una función de retrollamada. Se ignoran los argumentos de datos
12      # en este ejemplo. Más sobre retrollamadas más abajo.
13      def hello(self, widget, data=None):
14          print "Hello World"
15
16      def delete_event(self, widget, event, data=None):
17          # Si se devuelve FALSE en el gestor de la señal "delete_event",
18          # GTK emitirá la señal "destroy". La devolución de TRUE significa
19          # que no se desea la destrucción de la ventana.
20          # Esto sirve para presentar diálogos como: '¿Está seguro de que desea salir?'
21          #
22          print "delete event occurred"
23
24          # Si se cambia FALSE a TRUE la ventana principal no se
25          # destruirá con "delete_event".
26          return gtk.FALSE
27
28      # Otra retrollamada
29      def destroy(self, widget, data=None):
30          gtk.main_quit()
31
32      def __init__(self):
33          # se crea una ventana nueva
34          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
35
36          # Cuando se envía a una ventana la señal "delete_event" (esto lo hace
37          # generalmente el gestor de ventanas, usualmente con "cerrar", o con el icono
38          # de la ventana de título), pedimos que llame la función delete_event ()
```

```
39          # definida arriba. Los datos pasados a la retrollamada son
40          # NULL y se ignoran en la función de retrollamada.
41          self.window.connect("delete_event", self.delete_event)
42
43          # Conectamos el evento "destroy" a un manejador de señal.
44          # Este evento sucede cuando llamamos gtk_widget_destroy() para la ↩
     ventana,
45          # o si devolvemos FALSE en la retrollamada "delete_event".
46          self.window.connect("destroy", self.destroy)
47
48          # Establece el grosor del borde de la ventana.
49          self.window.set_border_width(10)
50
51          # Crea un nuevo botón con la etiqueta "Hello World".
52          self.button = gtk.Button("Hello World")
53
54          # Cuando el botón recibe la señal "clicked", llamará la
55          # función hello() a la que pasa None como argumento.  La función  ↩
    hello()
56          # se define más arriba.
57          self.button.connect("clicked", self.hello, None)
58
59          # Esto causará la destrucción de la ventana al llamar a
60          # gtk_widget_destroy(window) cuando se produzca "clicked".  De  ↩
    nuevo,
61          # la señal podría venir de aquí o del gestor de ventanas.
62          self.button.connect_object("clicked", gtk.Widget.destroy, self. ↩
    window)
63
64          # Esto empaqueta el botón en la ventana (un contenedor de GTK+).
65          self.window.add(self.button)
66
67          # El paso final es mostrar el control recién creado.
68          self.button.show()
69
70          # y la ventana
71          self.window.show()
72
73      def main(self):
74          # Todas las aplicaciones de PyGTK deben tener una llamada a gtk. ↩
    main(). Aquí se deja
75          # el control y se espera que suceda un evento (como un evento de  ↩
    teclado o ratón).
76          gtk.main()
77
78   # Si el programa se ejecuta directamente o se pasa como argumento al  ↩
    intérprete
79   # de Python, entonces se crea una instancia de HelloWorld y se muestra
80   if __name__ == "__main__":
81       hello = HelloWorld()
82       hello.main()
```

Figura 2.2 muestra la ventana creada por helloworld.py.

Figura 2.2 Programa de ejemplo: Hola Mundo

Las variables y funciones que se definen en el módulo PyGTK se llaman de la forma `gtk.*`. Por ejemplo, el programa helloworld.py usa:

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

del módulo PyGTK. En futuras secciones no se especificará el prefijo del módulo gtk, pero se dará por asumido. Naturalmente, los programas de ejemplo usarán los prefijos del módulo.

2.2. Teoría de Señales y Retrollamadas

NOTA

En la versión 2.0 de GTK+, el sistema de señales se ha movido de GTK+ a GLib. No entraremos en detalles sobre las extensiones que GLib 2.0 tiene en relación con el sistema de señales de GTK 1.2. Las diferecias no deberían notarse en el uso de PyGTK.

Antes de entrar en detalle en helloworld.py, discutiremos las señales y las retrollamadas. GTK+ es una biblioteca orientada a eventos, lo que significa que se dormirá en la función gtk.main() hasta que un evento ocurra y el control pase a la función apropiada.

Esta delegación del control se realiza usando la idea de "señales". (Nótese que estas señales no son las mismas que las señales de los sistemas Unix, y no se implementan usando éstas, aunque la terminología es casi idéntica) Cuando ocurre un evento, como cuando presionamos un botón del ratón, la señal apropiada se "emite" por el el control que fué presionado. Así es cómo GTK+ hace la mayoría de su trabajo útil. Hay señales que todos los controles heredan, como "destroy", y hay señales que son específicas de cada control, como "toggled" en el caso de un botón de activación.

Para hacer que un botón realice una acción, debemos configurar un manejador de señales que capture estas señales y llame a la función apropiada. Esto se hace usando un método de `gtk.Widget` (heredado de la clase `GObject`) como por ejemplo:

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

donde *`object`* es la instancia de `gtk.Widget` (un control) que estará emitiendo la señal, y el primer argumento *`name`* es una cadena que contiene el nombre de la señal que se desea capturar. El segundo argumento, *`func`*, es la función que se quiere llamar cuando se produce el evento. El tercer argumento, *`func_data`*, son los datos que se desean pasar a la función *`func`*. El método devuelve un handler_id que se puede usar para desconectar o bloquear el uso del manejador.

La función especificada en el tercer argumento se llama "función de retrollamada", y generalmente tiene la forma:

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

donde el primer argumento será una referencia al *`widget`* (control) que emitió la señal, y el segundo (*`callback_data`*) una referencia a los datos dados como último argumento en el método connect() mostrado antes.

Si la función de retrollamada es un método de un objeto entonces tendrá la forma general siguiente:

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

donde *`self`* es la instancia del objeto que invoca este método. Esta es la forma usada en el programa de ejemplo helloworld.py.

NOTA

La forma anterior de declaración de una función de retrollamada a señales es sólo una guía general, ya que las señales específicas de los distintos controles generan diferentes parámetros de llamada.

Otra llamada que se usa en el ejemplo helloworld.py es:

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

connect_object() es idéntica a connect(), exceptuando que una función de retrollamada sólo usa un argumento, y un método de retrollamada, dos argumentos:

```
def callback_func(object)
def callback_meth(self, object)
```

donde *object* normalmente es un control. connect_object() permite usar los métodos de controles PyGTK qué solo admiten un argumento (*self*) como manejadores de señales.

2.3. Eventos

Además del mecanismo de señales descrito anteriormente, hay un conjunto de eventos que reflejan el mecanismo de eventos de X. Las retrollamadas también se pueden conectar a estos eventos. Estos eventos son:

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

Para conectar una función de retrollamada a uno de estos eventos se usa el método connect(), como se ha dicho anteriormente, usando uno de los nombres de eventos anteriores en el parámetro *name*. La función (o método) de retrollamada para eventos es ligeramente diferente de la usada para señales:

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

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

`gdk.Event` es un tipo de objetos Python cuyos atributos de tipo indicarán cuál de los eventos anteriores ha ocurrido. Los otros atributos del evento dependerán del tipo de evento. Los valores posibles para los tipos son:

```
NOTHING
DELETE
DESTROY
EXPOSE
MOTION_NOTIFY
BUTTON_PRESS
_2BUTTON_PRESS
_3BUTTON_PRESS
BUTTON_RELEASE
KEY_PRESS
KEY_RELEASE
ENTER_NOTIFY
LEAVE_NOTIFY
FOCUS_CHANGE
CONFIGURE
MAP
UNMAP
PROPERTY_NOTIFY
SELECTION_CLEAR
SELECTION_REQUEST
SELECTION_NOTIFY
PROXIMITY_IN
PROXIMITY_OUT
DRAG_ENTER
DRAG_LEAVE
DRAG_MOTION
DRAG_STATUS
DROP_START
DROP_FINISHED
CLIENT_EVENT
VISIBILITY_NOTIFY
NO_EXPOSE
SCROLL
WINDOW_STATE
SETTING
```

Para acceder a estos valores se añade el prefijo `gtk.gdk.` al tipo de evento. Por ejemplo, `gtk.gdk.DRAG_ENTER`.

Por tanto, para conectar una función de retrollamada a uno de estos eventos se usaría algo como:

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

Esto asume que *`button`* es un control `gtk.Button`. Entonces, cuando el ratón esté sobre el botón y se pulse un botón del ratón, se llamará a la función *`button_press_callback`*. Esta función se puede definir así:

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

El valor que devuelve esta función indica si el evento debe ser propagado por el sistema de manejo de eventos GTK+. Devolviendo `gtk.TRUE` indicamos que el evento ha sido procesado, y que no debe ser propagado. Devolviendo `gtk.FALSE` se continua el procesamiento normal del evento. Es aconsejable consultar la sección Procesamiento Avanzado de Eventos y Señales para obtener más detalles sobre el sistema de propagación.

Las APIs de selección y arrastrar y soltar de GDK también emiten unos cuantos eventos que se reflejan en GTK+ por medio de señales. Consulta Señales en el Control de Orígen y Señales en el Control de Destino para obtener más detalles sobre la sintaxis de las funciones de retrollamada para estas señales:

```
selection_received
selection_get
drag_begin_event
drag_end_event
```

```
drag_data_delete
drag_motion
drag_drop
drag_data_get
drag_data_received
```

2.4. Hola Mundo Paso a Paso

Ahora que ya conocemos la teoría general, vamos a aclarar el programa de ejemplo helloworld.py paso a paso.

Las líneas 9-76 definen la clase `HelloWorld`, que contiene todas las retrollamadas como métodos de objeto y el método de inicialización de objetos. Examinemos los métodos de retrollamada:

Las líneas 13-14 definen el método de retrollamada hello() que será llamado al pulsar el botón. Cuando se llama a este método, se imprime "Hello World" en la consola. En el ejemplo ignoramos los parámetros de la instancia del objeto, el control y los datos, pero la mayoría de las retrollamadas los usan. El parámetro *data* se define con un valor predeterminado igual a `None` porque PyGTK no pasará ningún valor para los datos si no son incluidos en la llamada a connect(); esto produciría un error ya que la retrollamada espera tres argumentos y únicamente recibe dos. La definición de un valor por defecto `None` permite llamar a la retrollamada con dos o tres parámetros sin ningún error. En este caso el parámetro de datos podría haberse omitido, ya que el método hello() siempre será llamado con sólo dos parámetros (nunca se usan los datos de usuario). En el siguiente ejemplo usaremos el argumento de datos para saber qué botón fue pulsado.

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

La siguiente retrollamada (líneas 16-26) es un poco especial. El evento "delete_event" se produce cuando el manejador de ventanas manda este evento al programa. Tenemos varias posibilidades en cuanto a qué hacer con estos eventos. Podemos ignorarlos, realizar algun tipo de respuesta, o simplemente cerrar el programa.

El valor que se devuelva en esta retrollamada le permite a GTK+ saber qué acción realizar. Si devolvemos `TRUE` hacemos saber que no queremos que se emita la señal "destroy", y así nuestra aplicación sigue ejecutándose. Si devolvemos `FALSE` pedimos que se emita la señal "destroy", que a su vez llamará a nuestro manejador de la señal "destroy". Nótese que se han quitado los comentarios para mayor claridad.

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

El método de retrollamada destroy() (líneas 29-30) hace que el programa termine mediante la lllamada a `gtk.main_quit()`. Esta función indica a GTK que debe salir de la función `gtk.main()` cuando el control le sea transferido.

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

Las líneas 32-71 definen el método de inicialización de instancia __init__() del objeto `HelloWorld`, el cual crea la ventana y los controles que se usan en el programa.

La línea 34 crea una nueva ventana, pero no se muestra directamente hasta que comunicamos a GTK+ que lo haga, casi al final del programa. La referencia a la ventana se guarda en un atributo de instancia (self.window) para poder acceder a ella después.

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

Las líneas 41 y 46 ilustran dos ejemplos de cómo conectar un manejador de señal a un objeto, en este caso a `window`. Aquí se captura el evento "delete_event" y la señal "destroy". El primero se emite cuando cerramos la ventana a través del manejador de ventanas o cuando usamos la llamada al método destroy() de `gtk.Widget`. La segunda se emite cuando, en el manejador de "delete_event", devolvemos `FALSE`.

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

La línea 49 establece un atributo de un objeto contenedor (en este caso `window`) para que tenga un área vacía de 10 píxeles de ancho a su alrededor en donde no se sitúe ningún control. Hay otras funciones similares que se tratarán en la sección Establecimiento de Atributos de Controles

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

La línea 52 crea un nuevo botón y guarda una referencia a él en `self.button`. El botón tendrá la etiqueta "Hello World" cuando se muestre.

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

En la línea 57 conectamos un manejador de señal al botón, de forma que, cuando emita la señal "clicked", se llame a nuestro manejador de retrollamada hello(). No pasamos ningún dato a hello() así que simplemente se entrega `None` como dato. Obviamente, la señal "clicked" se emite al hacer clic en el botón con el cursor del ratón. El valor del parámetro de los datos `None` no es imprescindible y podría omitirse. La retrollamada se llamará con un parámetro menos.

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

También vamos a usar este botón para salir del programa. La línea 62 muestra cómo la señal "destroy" puede venir del manejador de ventanas, o de nuestro programa. Al hacer clic en el botón, al igual que antes, se llama primero a la retrollamada hello(), y después a la siguiente en el orden en el que han sido configuradas. Se pueden tener todas las retrollamadas que sean necesarias, y se ejecutarán en el orden en el que se hayan conectado.

Como queremos usar el método destroy() de la clase `gtk.Widget` que acepta un argumento (el control que se va a destruir - en este caso `window`), utilizamos el método connect_object() y le pasamos la referencia a la ventana. El método connect_object() organiza el primer argumento de la retrollamada para que sea `window` en vez del botón.

Cuando se llama el método destroy() de la clase `gtk.Widget` se emite la señal "destroy" desde la ventana, lo que a su vez provocará la llamada al método destroy() de la clase `HelloWorld` que termina el programa.

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

La línea 65 es una llamada de colocación, que se explicará en profundidad más tarde, en Colocación de Controles, aunque es bastante fácil de entender. Simplemente indica a GTK+ que el botón debe situarse en la ventana en donde se va a mostrar. Ha de tenerse en cuenta que un contenedor GTK+ únicamente puede contener un control. Otros controles, descritos más adelante, están diseñados para posicionar varios controles de otras maneras.

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

Ahora lo tenemos todo configurado como queremos. Con todos los manejadores de señales, y el botón situado en la ventana donde debería estar, pedimos a GTK (líneas 66 y 69) que muestre los controles en pantalla. El control de ventana se muestra en último lugar, para que la ventana entera aparezca de una vez y no primero la ventana y luego el botón dentro de ella dibujándose. Sin embargo, con un ejemplo tan simple, sería difícil apreciar la diferencia.

```
    self.button.show()

    self.window.show()
```

Las líneas 73-75 definen el método `main`() que llama a la función `gtk.main`()

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

Las líneas 80-82 permiten al programa ejecutarse automáticamente si es llamado directamente o como argumento del intérprete de python. La línea 81 crea una instancia de la clase `HelloWorld` y guarda una referencia a ella en la variable `hello`. La línea 82 llama al método `main`() de la clase `HelloWorld` para empezar el bucle de procesamiento de eventos GTK.

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

Ahora, cuando hagamos clic con el botón del ratón en el botón GTK, el control emitirá una señal "clicked". Para poder usar esta información, nuestro programa configura un manejador de señal que capture esta señal, la cual llama a la función que decidamos. En nuestro ejemplo, cuando se pulsa el botón que hemos creado, se llama el método hello() con un argumento `None`, y después se llama el siguiente manejador para esta señal. El siguiente manejador llama a la función destroy() del control con la ventana como su argumento y de esta manera causa que la ventana emita la señal "destroy", que es capturada y llama al método destroy() de la clase `HelloWorld`

Otra función de los eventos es usar el manejador de ventanas para eliminar la ventana, lo que causará que se emita "delete_event". Esto llamará a nuestro manejador de "delete_event". Si devolvemos `TRUE` aquí, la ventana se quedará como si nada hubiera pasado. Devolviendo `FALSE` hará que GTK+ emita la señal "destroy", que llama a la retrollamada "destroy" de la clase `HelloWorld` cerrando GTK+.

Capítulo 3

Avanzando

3.1. Más sobre manejadores de señales

Veamos otra vez la llamada a connect() .

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

El valor de retorno de connect() es un número entero que identifica la retrollamada. Como ya se ha mencionado, es posible disponer de tantas retrollamadas por señal como sea necesario, y cada una de ellas se ejecutará por turnos, en el mismo orden de conexión.

Este identificador permite eliminar la retrollamada de la lista de retrollamadas activas mediante el método:

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

Así, pasando el identificador devuelto por los métodos de conexión, es posible desconectar un manejador de señal.

También es posible deshabilitar temporalmente un manejador de señal mediante los métodos handler_block() y handler_unblock().

```
object.handler_block(handler_id)

object.handler_unblock(handler_id)
```

3.2. Un Hola Mundo Mejorado

```
 1  #!/usr/bin/env python
 2
 3  # Ejemplo helloworld2.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class HelloWorld2:
10
11      # La retrollamada mejorada. Los datos que se pasan a esta función
12      # se imprimen por la salida estándar.
13      def callback(self, widget, data):
14          print "Hello again - %s was pressed" % data
15
16      # otra retrollamada
17      def delete_event(self, widget, event, data=None):
18          gtk.main_quit()
19          return gtk.FALSE
20
21      def __init__(self):
```

```
22          # Creamos una ventana
23          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
24
25          # Esta llamada establece el título de la
26          # ventana como "Hello Buttons!"
27          self.window.set_title("Hello Buttons!")
28
29          # Aquí establecemos un manejador para delete_event que
30          # sale inmediatamente de GTK+.
31          self.window.connect("delete_event", self.delete_event)
32
33          # Establece el grosor del borde de la ventana
34          self.window.set_border_width(10)
35
36          # Creamos una caja en la que empaquetar los controles. Esto se ↩
    describe detalladamente
37          # en la sección de "empaquetado". La caja no es visible en ↩
    realidad sino que simplemente
38          # facilita la organización de los controles.
39          self.box1 = gtk.HBox(gtk.FALSE, 0)
40
41          # Introducimos la caja en la ventana principal
42          self.window.add(self.box1)
43
44          # Crea un nuevo botón con la etiqueta "Button 1".
45          self.button1 = gtk.Button("Button 1")
46
47          # Ahora, cuando se pulsa el botón, llamamos al método "callback"
48          # con un puntero a "button 1" como argumento
49          self.button1.connect("clicked", self.callback, "button 1")
50
51          # En vez de usar add(), empaquetamos este botón en la caja ↩
    visible
52          # que ha sido introducida en la ventana.
53          self.box1.pack_start(self.button1, gtk.TRUE, gtk.TRUE, 0)
54
55          # Hay que recordar siempre este paso, que indica a GTK+ que la ↩
    preparación del
56          # botón ha terminado y que ya es posible mostrarlo.
57          self.button1.show()
58
59          # Seguimos los mismos pasos para crear el segundo botón
60          self.button2 = gtk.Button("Button 2")
61
62          # Llamamos la misma retrollamada pero con un argumento diferente,
63          # haciendo referencia a "button 2" esta vez.
64          self.button2.connect("clicked", self.callback, "button 2")
65
66          self.box1.pack_start(self.button2, gtk.TRUE, gtk.TRUE, 0)
67
68          # El orden en que mostramos los botones no es muy importante, ↩
    pero es recomendable
69          # mostrar la ventana en último lugar, puesto que así aparece todo ↩
     de una vez.
70          self.button2.show()
71          self.box1.show()
72          self.window.show()
73
74   def main():
75       gtk.main()
76
77   if __name__ == "__main__":
78       hello = HelloWorld2()
79       main()
```

Al ejecutar helloworld2.py se genera la ventana de la Figura 3.1.

Figura 3.1 Ejemplo mejorado de Hola Mundo

Esta vez se puede ver que no hay forma fácil de salir del programa, y resulta necesario usar el gestor de ventanas o la línea de comandos para eliminarlo. Un buen ejercicio para el lector sería insertar un tercer botón "Salir" que cerrara el programa. Sería interesante jugar con las opciones de pack_start() al tiempo que se lee la siguiente sección, así como probar a cambiar de tamaño la ventana y observar qué sucede.

Como nota, hay que mencionar otra constante útil para `gtk.Window()` - `WINDOW_DIALOG`. Este tipo de ventana interactúa de forma distinta con el gestor de ventanas y debe usarse en ventanas de uso transitorio.

A continuación se describen en orden las pequeñas diferencias del código respecto a la versión inicial del programa "Hola Mundo":

Como ya se ha dicho, no existe manejador del evento "destroy" en esta versión mejorada de "Hola Mundo".

Las líneas 13-14 definen un método de retrollamada similar a la retrollamada hello() del ejemplo inicial. La diferencia reside en que ahora la retrollamada imprime un mensaje que incluye los datos que se le suministran.

La línea 27 pone título a la barra de título de la ventana (véase la Figura 3.1).

La línea 39 crea una caja horizontal (`gtk.HBox`) que almacena los dos botones que se crean en las líneas 45 y 60. La línea 42 añade la caja horizontal al contenedor de la ventana.

Las líneas 49 y 64 conectan el método callback() a la señal "clicked" de los botones. Y cada botón establece una cadena diferente que se pasa al método callback() al ser invocado.

Las líneas 53 y 66 empaquetan los botones en la caja horizontal. Y, finalmente, las líneas 57 y 70 indican a GTK+ que muestre los botones.

Las líneas 71-72 piden finalmente a GTK+ que muestre la caja y la ventana.

Capítulo 4

Empaquetado de Controles

Normalmente, cuando se crea un programa, se desea poner más de un control en la ventana. Nuestro primer ejemplo "Hola Mundo" usaba un único control para poder llamar simplemente al método `add()` de la clase `gtk.Container` para "empaquetar" el control en la ventana. Sin embargo, en cuanto se quiera poner más de un control en una ventana, ¿cómo se determina la posición en la que se sitúa el control?. Aquí es donde el "empaquetado" de controles entra en juego.

4.1. Teoría de Cajas Empaquetadoras

La mayoría del empaquetado se realiza utilizando cajas. Éstas son contenedores invisibles de controles y son de dos tipos: cajas horizontales y cajas verticales. En el primer tipo los los objetos se insertan horizontalmente, de izquierda a derecha o de derecha a izquierda, en función de la llamada que se use; mientras que en el segundo tipo, las cajas verticales, los controles se empaquetan de arriba a abajo o viceversa. Es posible utilizar combinaciones de cajas insertadas en otras cajas y obtener cualquier efecto que se desee.

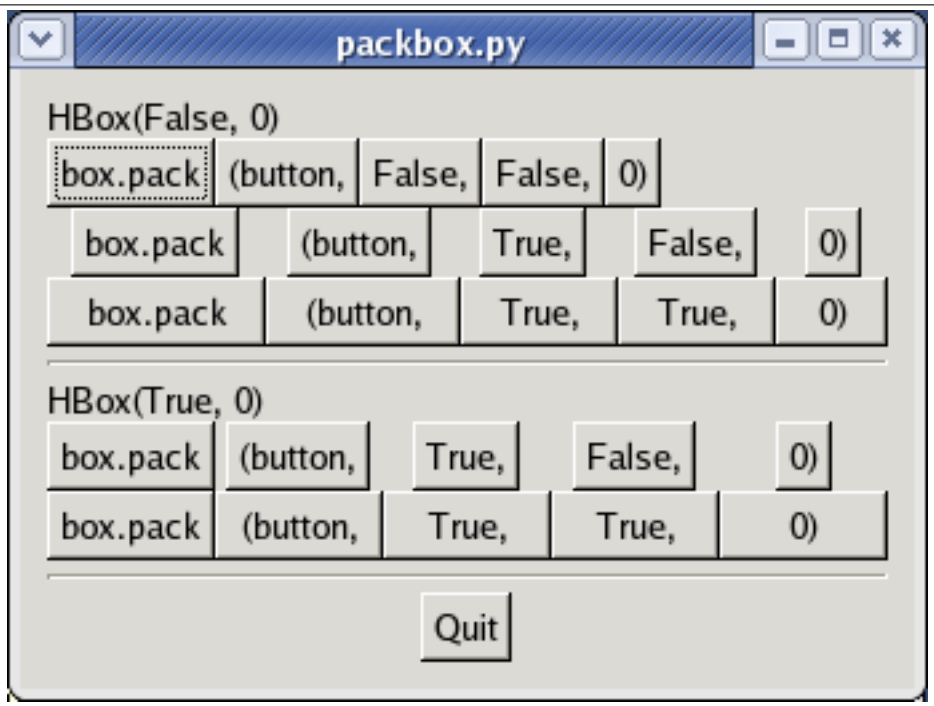
Para crear una nueva caja horizontal se usa la llamada `gtk.HBox`(), y con cajas verticales `gtk.VBox`() . Los métodos pack_start() y pack_end() se utilizan para colocar los objetos dentro de estos contenedores. El primer método, pack_start(), inserta los objetos yendo de arriba hacia abajo en una caja vertical, y de izquierda a derecha en una caja horizontal. El método pack_end() muestra el comportamiento opuesto, empaqueta de abajo hacia arriba en una caja vertical, y de derecha a izquierda en una caja horizontal. Con estos métodos se pueden alinear a la derecha o a la izquierda los controles, de tal forma que se consiga el efecto buscado. A lo largo de los ejemplos de este tutorial se usará fundamentalmente el método pack_start(). Un objeto puede ser además bien otro contenedor o bien un control. De hecho, muchos controles son en realidad también contenedores, como ocurre con los botones, aunque normalmente se use sólo una etiqueta en su interior.

Con las llamadas anteriores se indica a GTK+ cómo ha de situar los controles y así es capaz de cambiar su tamaño y otras propiedades interesantes de forma automática. Y, como es de esperar, dicho método proporciona además gran flexibilidad a la hora de situar y crear controles.

4.2. Las Cajas en detalle

A causa de esta flexibilidad, el empaquetado de cajas puede resultar confuso al principio, dado que admite muchas opciones cuyo funcionamiento conjunto no resulta obvio. Sin embargo, existen básicamente cinco estilos. La Figura 4.1 muestra el resultado de la ejecución del programa packbox.py con un argumento de 1:

Figura 4.1 Empaquetado: Cinco variaciones



Cada línea contiene una caja horizontal (hbox) con varios botones. La llamada a pack es una copia de la llamada a pack en cada uno de los botones de la `Hbox`. Cada botón se empaqueta en la hbox de la misma manera (con los mismos argumentos al método pack_start()).

Este es un ejemplo del método pack_start():

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

box es la caja donde se empaqueta el objeto. El primer argumento, *child*, es el objeto que se va a empaquetar. Por ahora los objetos serán botones, con lo que estaríamos empaquetando botones dentro de cajas.

El argumento *expand* de pack_start() y pack_end() controla si los controles se disponen de forma que ocupen todo el espacio extra de la caja y, de esta manera, ésta se expande hasta ocupar todo el área reservada para ella (`TRUE`); o si se encoge para ocupar el espacio justo de los controles (`FALSE`). Poner expand a `FALSE` permite justificar a la derecha y a la izquierda los controles. Si no, se expandirán para llenar la caja, y el mismo efecto podría obtenerse usando sólo o pack_start() o pack_end().

El argumento *fill* controla si el espacio extra se utiliza en los propios objetos (`TRUE`) o como espacio extra en la caja alrededor de los objetos (`FALSE`). Sólo tiene efecto si el argumento expand también es `TRUE`.

Python permite definir un método o función con valores de argumento predeterminados y argumentos con nombre. A lo largo de este tutorial se verá la definición de las funciones y métodos con valores predeterminados y argumentos con nombre cuando sean de aplicación. Por ejemplo, el método pack_start se define así:

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

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

child, *expand*, *fill* y *padding* son palabras clave (argumentos con nombre). Los argumentos *expand*, *fill* y *padding* tienen los valores predeterminados (o "por defecto") mostrados arriba. El argumento *child* debe especificarse obligatoriamente al no tener un valor predeterminado.

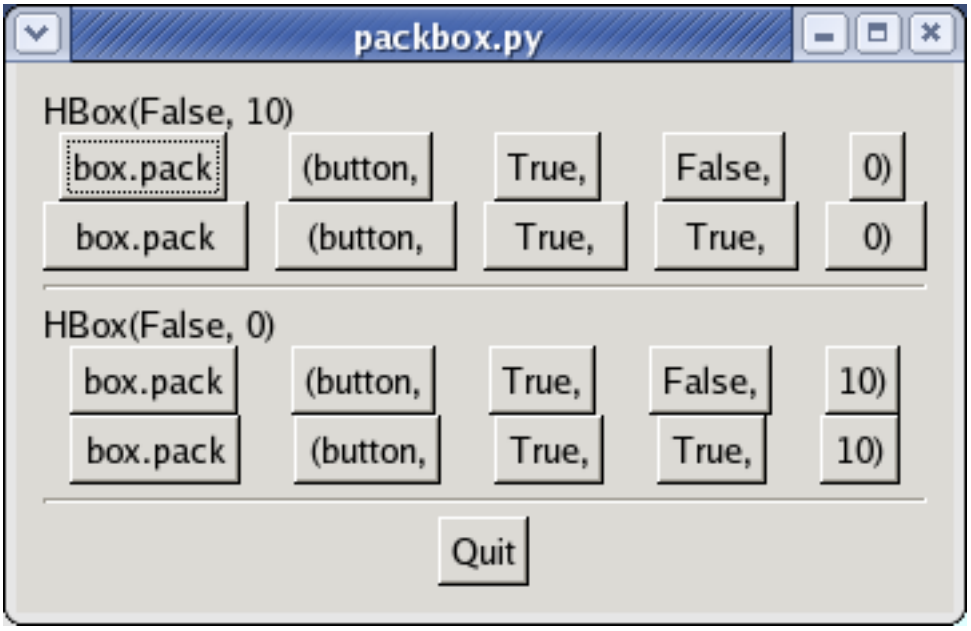
Las funciones que nos permiten crear una caja nueva son:

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

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

El argumento *homogeneous* de `gtk.HBox()` y `gtk.VBox()` controla si cada objeto de la caja tiene el mismo tamaño (por ejemplo, el mismo ancho en una hbox, o la misma altura en una vbox). Si se usa, las rutinas de empaquetado funcionan basicamente como si el argumento expand estuviera siempre activado.

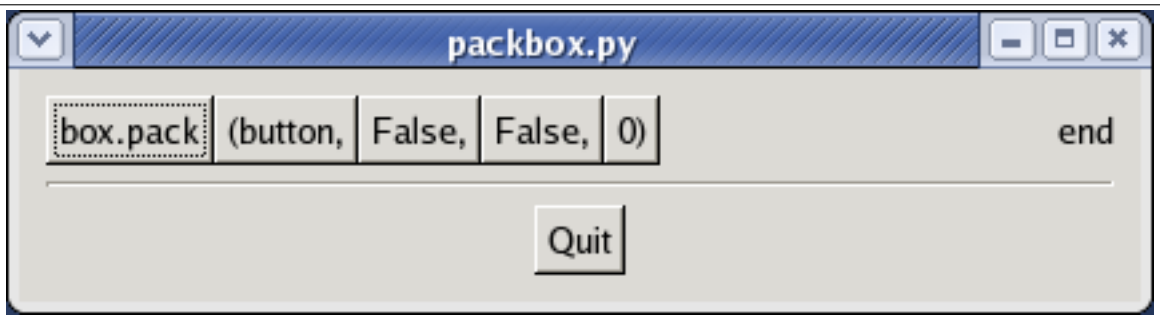
¿Qué diferencia existe entre *spacing* (se fija al crear la caja) y *padding* (se determina al empaquetar los elementos)? El *spacing* se añade entre objetos, y el *padding* se añade a cada lado de un objeto. La Figura 4.2 ilustra la diferencia, pasando un argumento de 2 a packbox.py :

Figura 4.2 Empaquetado con Spacing y Padding



La Figura 4.3 ilustra el uso del método pack_end() (pasa un argumento de 3 a packbox.py). La etiqueta "end" se empaqueta con el método pack_end(). Se mantendrá en el borde derecho de la ventana cuando ésta se redimensione.

Figura 4.3 Empaquetado con pack_end()



4.3. Programa de Ejemplo de Empaquetado

Aquí está el código usado para crear la imagen anterior. Está profusamente comentado, así que no resultará complicado seguirlo. Es recomendable su ejecución y jugar posteriormente con él.

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

```
 8 import sys, string
 9
10 # Función auxiliar que crea una nueva HBox llena de botones con etiqueta. Los argumentos
11 # de las variables en las que tenemos interés se pasan a esta función. No mostramos la
12 # caja pero sí todo lo que contiene.
13
14 def make_box(homogeneous, spacing, expand, fill, padding):
15
16     # Creamos una nueva HBox con los parámetros homogeneous
17     # y spacing adecuados.
18     box = gtk.HBox(homogeneous, spacing)
19
20     # Creamos una serie de botones con los parámetros adecuados
21     button = gtk.Button("box.pack")
22     box.pack_start(button, expand, fill, padding)
23     button.show()
24
25     button = gtk.Button("(button,")
26     box.pack_start(button, expand, fill, padding)
27     button.show()
28
29     # Creamos un botón con una etiqueta que depende del valor de
30     # expand.
31     if expand == gtk.TRUE:
32         button = gtk.Button("TRUE,")
33     else:
34         button = gtk.Button("FALSE,")
35
36     box.pack_start(button, expand, fill, padding)
37     button.show()
38
39     # Aquí hacemos lo mismo que en la creación del botón de "expand"
40     # anterior, pero usa la forma abreviada.
41     button = gtk.Button(("FALSE,", "TRUE,")[fill==gtk.TRUE])
42     box.pack_start(button, expand, fill, padding)
43     button.show()
44
45     padstr = "%d)" % padding
46
47     button = gtk.Button(padstr)
48     box.pack_start(button, expand, fill, padding)
49     button.show()
50     return box
51
52 class PackBox1:
53     def delete_event(self, widget, event, data=None):
54         gtk.main_quit()
55         return gtk.FALSE
56
57     def __init__(self, which):
58
59         # Creamos nuestra ventana
60         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
61
62         # Siempre debemos recordar la conexión de la señal delete_event
63         # a la ventana principal. Esto es muy importante de cara a un comportamiento
64         # intuitivo adecuado
65         self.window.connect("delete_event", self.delete_event)
66         self.window.set_border_width(10)
67
68         # Creamos una caja vertical (vbox) en la que empaquetar las cajas
```

```
    horizontales.
 69         # Esto nos permite apilar las cajas horizontales llenas de botones
 70         # una encima de otra en esta vbox.
 71         box1 = gtk.VBox(gtk.FALSE, 0)
 72
 73         # qué ejemplo mostramos. Éstos se corresponden a las imágenes ↩
    anteriores.
 74         if which == 1:
 75             # creamos una etiqueta nueva.
 76             label = gtk.Label("HBox(FALSE, 0)")
 77
 78             # Alineamos la etiqueta al lado izquierdo. Comentaremos este y ↩
    otros
 79             # métodos en la sección sobre Atributos de los Controles.
 80             label.set_alignment(0, 0)
 81
 82             # Empaquetamos la etiqueta en la caja vertical (vbox box1). ↩
    Recuérdese que
 83             # los controles que se añaden a una caja vertical se apilan uno ↩
     encima del otro
 84             # en orden.
 85             box1.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
 86
 87             # Mostramos la etiqueta
 88             label.show()
 89
 90             # Llamamos a nuestra función de crear caja - homogeneous = ↩
    FALSE, spacing = 0,
 91             # expand = FALSE, fill = FALSE, padding = 0
 92             box2 = make_box(gtk.FALSE, 0, gtk.FALSE, gtk.FALSE, 0)
 93             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
 94             box2.show()
 95
 96             # Llamamos a nuestra función de crear caja - homogeneous = ↩
    FALSE, spacing = 0,
 97             # expand = TRUE, fill = FALSE, padding = 0
 98             box2 = make_box(gtk.FALSE, 0, gtk.TRUE, gtk.FALSE, 0)
 99             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
100             box2.show()
101
102             # Los argumentos son: homogeneous, spacing, expand, fill, ↩
    padding
103             box2 = make_box(gtk.FALSE, 0, gtk.TRUE, gtk.TRUE, 0)
104             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
105             box2.show()
106
107             # Crea un separador, que veremos qué hacen más adelante,
108             # aunque son muy simples.
109             separator = gtk.HSeparator()
110
111             # Empaquetamos el separador en la vbox. Recuérdese que ↩
    empaquetamos todos estos
112             # controles en una vbox, por lo que se apilarán
113             # verticalmente.
114             box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 5)
115             separator.show()
116
117             # Creamos otra etiqueta y la mostramos.
118             label = gtk.Label("HBox(TRUE, 0)")
119             label.set_alignment(0, 0)
120             box1.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
121             label.show()
122
123             # Los argumentos son: homogeneous, spacing, expand, fill, ↩
```

```
    padding
124             box2 = make_box(gtk.TRUE, 0, gtk.TRUE, gtk.FALSE, 0)
125             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
126             box2.show()
127
128             # Los argumentos son: homogeneous, spacing, expand, fill,  ↩
    padding
129             box2 = make_box(gtk.TRUE, 0, gtk.TRUE, gtk.TRUE, 0)
130             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
131             box2.show()
132
133             # Otro separador.
134             separator = gtk.HSeparator()
135             # Los últimos 3 argumentos de pack_start son:
136             # expand, fill, padding.
137             box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 5)
138             separator.show()
139         elif which == 2:
140             # Creamos una etiqueta nueva, recordando que box1 es una vbox  ↩
    creada
141             # cerca del comienzo de __init__()
142             label = gtk.Label("HBox(FALSE, 10)")
143             label.set_alignment( 0, 0)
144             box1.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
145             label.show()
146
147             # Los argumentos son: homogeneous, spacing, expand, fill,  ↩
    padding
148             box2 = make_box(gtk.FALSE, 10, gtk.TRUE, gtk.FALSE, 0)
149             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
150             box2.show()
151
152             # Los argumentos son: homogeneous, spacing, expand, fill,  ↩
    padding
153             box2 = make_box(gtk.FALSE, 10, gtk.TRUE, gtk.TRUE, 0)
154             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
155             box2.show()
156
157             separator = gtk.HSeparator()
158             # Los últimos 3 argumentos de pack_start son:
159             # expand, fill, padding.
160             box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 5)
161             separator.show()
162
163             label = gtk.Label("HBox(FALSE, 0)")
164             label.set_alignment(0, 0)
165             box1.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
166             label.show()
167
168             # Los argumentos son: homogeneous, spacing, expand, fill,  ↩
    padding
169             box2 = make_box(gtk.FALSE, 0, gtk.TRUE, gtk.FALSE, 10)
170             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
171             box2.show()
172
173             # Los argumentos son: homogeneous, spacing, expand, fill,  ↩
    padding
174             box2 = make_box(gtk.FALSE, 0, gtk.TRUE, gtk.TRUE, 10)
175             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
176             box2.show()
177
178             separator = gtk.HSeparator()
179             # Los últimos 3 argumentos de pack_start son:
180             # expand, fill, padding.
```

```
181             box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 5)
182             separator.show()
183 
184         elif which == 3:
185 
186             # Esto ilustra la posibilidad de usar pack_end() para
187             # alinear los controles a la derecha. Primero creamos una caja nueva, como antes.
188             box2 = make_box(gtk.FALSE, 0, gtk.FALSE, gtk.FALSE, 0)
189 
190             # Creamos la etiqueta que pondremos al final.
191             label = gtk.Label("end")
192             # La empaquetamos con pack_end(), por lo que se pone en el extremo derecho
193             # de la hbox creada en la llamada a make_box().
194             box2.pack_end(label, gtk.FALSE, gtk.FALSE, 0)
195             # Mostramos la etiqueta.
196             label.show()
197 
198             # Empaquetamos la box2 en box1
199             box1.pack_start(box2, gtk.FALSE, gtk.FALSE, 0)
200             box2.show()
201 
202             # Un separador para la parte de abajo.
203             separator = gtk.HSeparator()
204 
205             # Esto establece explícitamente el ancho del separador a 400 píxeles y 5
206             # píxeles de alto. Así la hbox que creamos también tendría 400
207             # píxeles de ancho, y la etiqueta "end" estará separada de las otras
208             # de la hbox. En otro caso, todos los controles de la
209             # hbox estarían empaquetados lo más juntos posible.
210             separator.set_size_request(400, 5)
211             # empaquetamos el separador en la vbox (box1) creada cerca del principio
212             # de __init__()
213             box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 5)
214             separator.show()
215 
216         # Creamos otra hbox nueva. ¡Recordemos que podríamos usar cuantas queramos!
217         quitbox = gtk.HBox(gtk.FALSE, 0)
218 
219         # Nuestro botón de salida.
220         button = gtk.Button("Quit")
221 
222         # Configuramos la señal que finalice el programa al pulsar el botón
223         button.connect("clicked", lambda w: gtk.main_quit())
224         # Empaquetamos el botón en la quitbox.
225         # Los 3 últimos argumentos de pack_start son:
226         # expand, fill, padding.
227         quitbox.pack_start(button, gtk.TRUE, gtk.FALSE, 0)
228         # empaquetamos la quitbox en la vbox (box1)
229         box1.pack_start(quitbox, gtk.FALSE, gtk.FALSE, 0)
230 
231         # Empaquetamos la vbox (box1), que ahora contiene todos los controles,
232         # en la ventana principal.
233         self.window.add(box1)
234 
235         # Y mostramos todo lo que queda
236         button.show()
237         quitbox.show()
```

```
238
239         box1.show()
240         # Mostrando la ventana al final de forma que todo aparezca de una ↩
    vez.
241         self.window.show()
242
243 def main():
244     # y, naturalmente, el bucle de eventos principal.
245     gtk.main()
246     # El control se devuelve a este punto cuando se llama a main_quit().
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()
```

El pequeño tour por el código packbox.py empieza por las lineas 14-50 que definen una función auxiliar `make_box` que crea una caja horizontal y la rellena con botones según los parámetros específicados. Devuelve una referencia a la caja horizontal.

Las lineas 52-241 definen el método de inicialización __init__() de la clase `PackBox1` que crea una ventana y una caja vertical en ella que se rellena con una configuración de controles que depende del argumento que recibe. Si se le pasa un 1, las lineas 75-138 crean una ventana que muestra las cinco únicas posibilidades que resultan de variar los parámetros homogeneous, expand y fill. Si se le pasa un 2, las lineas 140-182 crean una ventana que muesstra las diferentes combinaciones de fill con spacing y padding. Finalmente, si se le pasa un 3, las lineas 188-214 crean una ventana que muestra el uso del método pack_start() para justificar los botones a la izquierda y el método `pack_end`() para justificar una etiqueta a la derecha. Las lineas 215-235 crean una caja horizontal que contiene un botón que se empaqueta dentro de la caja vertical. La señal 'clicked' del botón está conectada a la función PyGTK `gtk.main_quit`() para terminar el programa.

Las lineas 250-252 comprueban los argumentos de la línea de comandos y terminan el programa usando la función `sys.exit`() si no hay exactamente un argumento. La línea 251 crea una instancia de PackBox1. La línea 253 llama a la función `gtk.main`() para empezar el bucle de procesamiento de eventos GTK+.

En este programa de ejemplo, las referencias a los controles (excepto a la ventana) no se guardan en los atributos de instancia del objeto porque no se necesitan más tarde.

4.4. Uso de Tablas para el Empaquetado

Veamos otra manera de empaquetado. Mediante tablas. Pueden ser extremadamente útiles en determinadas situaciones.

Al usar tablas, creamos una rejilla en donde podemos colocar los controles, que pueden ocupar tantos espacios como especifiquemos.

Lo primero que debemos mirar es, obviamente, la función `gtk.Table`() :

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

El primer argumento es el número de filas de la tabla, mientras que el segundo, obviamente, es el número de columnas.

El argumento *`homogeneous`* tiene que ver con el tamaño de las celdas de la tabla. Si *`homogeneous`* es `TRUE`, las celdas de la tabla tienen el tamaño del mayor control en ella. Si *`homogeneous`* es `FALSE`, el tamaño de las celdas viene dado por el control más alto de su misma fila, y el control más ancho de su columna.

Las filas y las columnas se disponen de 0 a n, donde n es el número que se especificó en la llamada a `gtk.Table`(). Por tanto, si se especifica rows (filas) = 2 y columns (columnas) = 2, la dispoisición quedaría así:

```
 0          1          2
```

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

Obsérvese que el sistema de coordenadas tiene su orgien en la esquina superior izquierda. Para introducir un control en una caja, se usa el siguiente método:

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

La instancia table es la tabla que se creó con `gtk.Table()`. El primer parámetro ("child") es el control que se desea introducir en la tabla.

Los argumentos *left_attach*, *right_attach*, *top_attach* y *bottom_attach* especifican dónde situar el control, y cuántas cajas usar. Si se quiere poner un botón en la esquina inferior derecha de una tabla 2x2, y se quiere que ocupe SÓLO ese espacio, *left_attach* sería = 1, *right_attach* = 2, *top_attach* = 1, *bottom_attach* = 2.

Ahora, si se desea que un control ocupe la fila entera de la tabla 2x2, se pondría *left_attach* = 0, *right_attach* = 2, *top_attach* = 0, *bottom_attach* = 1.

Los argumentos *xoptions* e *yoptions* se usan para especificar opciones de colocación y pueden unirse mediante la operación `OR`, permitiendo así múltiples opciones.

Estas opciones son:

`FILL`	Si la caja es más grande que el control, y se especifica `FILL`, el control se expandirá hasta usar todo el espacio disponible.
`SHRINK`	Si se le asigna menos espacio a la tabla del que solicitó (normalmente porque el usuario ha redimensionado la ventana), entonces los controles normalmente sería empujados a la parte inferior de la ventana y desaparecerían. Si se especifica `SHRINK`, los controles se encojerán con la tabla.
`EXPAND`	Esto hará que la tabla se expanda para usar el espacio sobrante en la ventana.

El Padding es igual que en las cajas, ya que crea un espacio vacío especificado en píxeles alrededor del control.

También tenemos los métodos set_row_spacing() y set_col_spacing(), que añaden espacio entre las filas en la columna o fila especificada.

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

y

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

Obsérvese que, para las columnas, el espacio va a la derecha de la columna, y, para las filas, el espacio va debajo de la fila.

También se puede poner un espacio igual en todas las filas y/o columnas con:

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

y,

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

Obsérvese que con éstas funciones, la última fila y la última columna no obtienen ningún espacio.

4.5. Ejemplo de Empaquetado con Tablas

El programa de ejemplo table.py crea una ventana con tres botones en una tabla 2x2. Los primeros dos botones se colocarán en la fila superior. Un tercer botón, para salir, se coloca en la fila inferior, y ocupa las dos columnas. La Figura 4.4 ilustra la ventana resultante:

Figura 4.4 Empaquetado haciendo uso de una Tabla

Aquí sigue el código fuente:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo table.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class Table:
10     # Nuestra retrollamada.
11     # Los datos que se pasan a este método se imprimen por la salida ↩
   estándar.
12     def callback(self, widget, data=None):
13         print "Hello again - %s was pressed" % data
14
15     # Esta retrollamada sale del programa.
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return gtk.FALSE
19
20     def __init__(self):
21         # Creamos una ventana nueva
22         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         # Establecemos el título de la ventana
25         self.window.set_title("Table")
26
27         # Fijamos un manejador para delete_event que sale de GTK+
28         # inmediatamente.
29         self.window.connect("delete_event", self.delete_event)
30
31         # Fijamos el grosor de los bordes.
32         self.window.set_border_width(20)
33
34         # Creamos una tabla 2x2.
35         table = gtk.Table(2, 2, gtk.TRUE)
36
37         # Ponemos la tabla en la ventana principal
38         self.window.add(table)
39
40         # Creamos el primer botón
41         button = gtk.Button("button 1")
42
43         # Cuando se pulsa el botón llamamos al método "callback"
44         # con una indicación a "button 1" como argumento.
45         button.connect("clicked", self.callback, "button 1")
46
47
48         # Insertamos el botón 1 en el cuadrante superior izquierdo de la ↩
```

```
    tabla.
49          table.attach(button, 0, 1, 0, 1)
50
51          button.show()
52
53          # Creamos el segundo botón.
54
55          button = gtk.Button("button 2")
56
57          # Cuando se pulsa el botón llamamos al método "callback"
58          # con una indicación a "button 2" como argumento.
59          button.connect("clicked", self.callback, "button 2")
60          # Insertamos el botón 2 en el cuadrante superior derecho de la  ↩
    tabla.
61          table.attach(button, 1, 2, 0, 1)
62
63          button.show()
64
65          # Creamos el botón de salida "Quit"
66          button = gtk.Button("Quit")
67
68          # Cuando se pulsa el botón llamamos a la función main_quit y
69          # el programa se termina
70          button.connect("clicked", lambda w: gtk.main_quit())
71
72          # Insertamos el botón de salida en los dos cuadrantes inferiores de ↩
     la tabla
73          table.attach(button, 0, 2, 1, 2)
74
75          button.show()
76
77          table.show()
78          self.window.show()
79
80 def main():
81      gtk.main()
82      return 0
83
84 if __name__ == "__main__":
85      Table()
86      main()
```

La clase `Table` se define en las líneas 9-78. Las líneas 12-13 definen el método callback() que se llama al hacer "click" en dos de los botones. La retrollamada sólo imprime un mensaje en la consola indicando qué botón se pulsó usando los datos que se le pasan.

Las lineas 16-18 definen el método delete_event() que se llama cuando el manejador de ventanas le pide a la ventana que se elimine.

Las líneas 20-78 definen el constructor de la clase `Table` __init__(). Crea una ventana (línea 22), le pone el título (línea 25), conecta la retrollamada delete_event() a la señal "delete_event" (línea 29), y le pone el ancho al borde (línea 32). Se crea una `gtk.Table` en la línea 35 y se añade a la ventana en la línea 38.

Los dos botones superiores se crean en las líneas 41 y 55, sus señales "clicked" se conectan al método callback() en las líneas 45 y 59; y se añaden a la tabla en la primera fila en las líneas 49 y 61. Las líneas 66-72 crean el botón "Quit", conectan su señal "clicked" a la función `mainquit`() y lo añaden a la tabla ocupando la fila inferior completa.

Capítulo 5

Perspectiva General de Controles

Los pasos generales para usar un control (widget) en PyGTK son:

- Se llama a gtk.* (una de las múltiples funciones para crear un nuevo control, y que se detallan en esta sección.
- Se conectan todas la señales y eventos que queramos usar a los manejadores apropiados.
- Se establecen los atributos del control.
- Se empaqueta el control dentro de un contenedor usando una llamada como gtk.Container.add() o gtk.Box.pack_start().
- Se llama a gtk.Widget.show() en el control.

`show`() le permite saber a GTK que hemos terminado de configurar los atributos del control, y está listo para ser mostrado. También se puede usar `gtk.Widget.hide`() para que desaparezca otra vez. El orden en el que se muestran los controles no es importante, pero es conveniente que se muestre la ventana al final de modo que la toda la ventana aparezca de una vez y no se vea como van apareciendo los controles individuales en la ventana a medida que se van formando. Los hijos de un control (una ventana también es un control) no se mostrarán hasta que la propia ventana se muestre usando el método `show`() .

5.1. Jerarquía de Controles

Como referencia, aquí aparece el árbol de la jerarquía utilizada para implementar los controles. (Se han omitido los controles obsoletos y las clases auxiliares).

```
gobject.GObject
|
+gtk.Object (Objeto)
| +gtk.Widget (Control)
| | +gtk.Misc (Miscélaneo)
| | | +gtk.Label (Etiqueta)
| | | | `gtk.AccelLabel (EtiquetaAceleradora)
| | | +gtk.Arrow (Flecha)
| | | `gtk.Image (Imagen)
| | +gtk.Container (Contenedor)
| | | +gtk.Bin (Binario)
| | | | +gtk.Alignment (Alineador)
| | | | +gtk.Frame (Marco)
| | | | | `gtk.AspectFrame (Marco Proporcional)
| | | | +gtk.Button (Botón)
| | | | | +gtk.ToggleButton (Botón Biestado)
| | | | | | `gtk.CheckButton (Botón Activación)
| | | | | |   `gtk.RadioButton (Botón Exclusión Mútua)
| | | | | +gtk.ColorButton (Botón de selección de Color)
| | | | | +gtk.FontButton (Botón de selección de Fuente)
```

```
| | | | | `gtk.OptionMenu (Menú Opciones)
| | | | +gtk.Item (Elemento)
| | | | | +gtk.MenuItem (Elemento de Menú)
| | | | |   +gtk.CheckMenuItem (Elemento Activable de Menú)
| | | | |   | `gtk.RadioMenuItem (Elemento de Exclusión Mútua de Menú)
| | | | |   +gtk.ImageMenuItem (Elemento de Imagen de Menú)
| | | | |   +gtk.SeparatorMenuItem (Elemento de Separación de Menú)
| | | | |   `gtk.TearoffMenuItem (Menú Desprendible)
| | | | +gtk.Window (Ventana)
| | | | | +gtk.Dialog (Diálogo)
| | | | | | +gtk.ColorSelectionDialog (Diálogo de Selección de Colores)
| | | | | | +gtk.FileChooserDialog (Diálogo de Selección de Ficheros)
| | | | | | +gtk.FileSelection (Selector de Ficheros)
| | | | | | +gtk.FontSelectionDialog (Diálogo de Selección de Tipos de Letra)
| | | | | | +gtk.InputDialog (Diálogo de Entrada de Datos)
| | | | | | `gtk.MessageDialog (Diálogo de Mensaje)
| | | | | `gtk.Plug (Conectable)
| | | | +gtk.ComboBox (Caja con Lista Desplegable)
| | | | | `gtk.ComboBoxEntry (Entrada de Caja de Lista Desplegable)
| | | | +gtk.EventBox (Caja de Eventos)
| | | | +gtk.Expander (Expansión)
| | | | +gtk.HandleBox (Manejador de Caja)
| | | | +gtk.ToolItem (Elemento de Barra de Herramientas)
| | | | | +gtk.ToolButton (Botón de Barra de Herramientas)
| | | | | | +gtk.ToggleToolButton (Botón Biestado de Barra de Herramientas)
| | | | | | | `gtk.RadioToolButton (Botón de Exclusión Mútua de Barra de ↩
    Herramientas)
| | | | | `gtk.SeparatorTooItem (Separador de Elementos de Barra de Herramientas)
| | | | +gtk.ScrolledWindow (Ventana de Desplazamiento)
| | | | `gtk.Viewport (Vista)
| | | +gtk.Box (Caja)
| | | | +gtk.ButtonBox (Caja de Botones)
| | | | | +gtk.HButtonBox (Caja de Botones Horizontal)
| | | | | `gtk.VButtonBox (Caja de Botones Vertical)
| | | | +gtk.VBox (Caja Vertical)
| | | | | +gtk.ColorSelection (Selector de Colores)
| | | | | +gtk.FontSelection (Selector de Tipos de Letra)
| | | | | `gtk.GammaCurve (Curva Gamma)
| | | | `gtk.HBox (Caja Horizontal)
| | | |   +gtk.Combo (Lista Desplegable)
| | | |   `gtk.Statusbar (Barra de Estado)
| | | +gtk.Fixed (Fijo)
| | | +gtk.Paned (Panel)
| | | | +gtk.HPaned (Panel Horizontal)
| | | | `gtk.VPaned (Panel Vertical)
| | | +gtk.Layout (Disposición)
| | | +gtk.MenuShell (Consola de Menú)
| | | | +gtk.MenuBar (Barra de Menú)
| | | | `gtk.Menu (Menú)
| | | +gtk.Notebook (Cuaderno de Fichas)
| | | +gtk.Socket (Socket)
| | | +gtk.Table (Tabla)
| | | +gtk.TextView (Vista de Texto)
| | | +gtk.Toolbar (Barra de Herramientas)
| | | `gtk.TreeView (Vista de Árbol)
| | +gtk.Calendar (Calendario)
| | +gtk.DrawingArea (Área de Dibujo)
| | | `gtk.Curve (Curva)
| | +gtk.Entry (Entrada de Texto)
| | | `gtk.SpinButton (Botón Aumentar/Disminuir)
| | +gtk.Ruler (Regla)
| | | +gtk.HRuler (Regla Horizontal)
| | | `gtk.VRuler (Regla Vertical)
| | +gtk.Range (Rango)
```

```
| | | +gtk.Scale (Escala)
| | | | +gtk.HScale (Escala Horizontal)
| | | | `gtk.VScale (Escala Vertical)
| | | `gtk.Scrollbar (Barra de Desplazamiento)
| | |   +gtk.HScrollbar (Barra de Desplazamiento Horizontal)
| | |   `gtk.VScrollbar (Barra de Desplazamiento Vertical)
| | +gtk.Separator (Separador)
| | | +gtk.HSeparator (Separador Horizontal)
| | | `gtk.VSeparator (Separador Vertical)
| | +gtk.Invisible (Invisible)
| | +gtk.Progress (Elemento de Progreso)
| | | `gtk.ProgressBar (Barra de Progreso)
| +gtk.Adjustment (Ajuste)
| +gtk.CellRenderer (Visualizador de Celda)
| | +gtk.CellRendererPixbuf (Visualizador de Imágen de Celda)
| | +gtk.CellRendererText (Visualizador de Texto de Celda)
| | +gtk.CellRendererToggle (Visualizador de Activación de Celda)
| +gtk.FileFilter (Filtro de Selección de Archivos)
| +gtk.ItemFactory (Factoría de Elementos)
| +gtk.Tooltips (Pistas)
| `gtk.TreeViewColumn (Columna de Vista de Árbol)
+gtk.Action (Acción)
| +gtk.ToggleAction (Acción Biestado)
| | `gtk.RadioAction (Acción de Exclusión Mútua)
+gtk.ActionGroup (Grupo de Acciones)
+gtk.EntryCompletion (Completado de Entrada)
+gtk.IconFactory (Factoría de Iconos)
+gtk.IconTheme (Tema de Iconos)
+gtk.IMContext (Contexto de Método de Entrada)
| +gtk.IMContextSimple (Contexto Simple de Método de Entrada)
| `gtk.IMMulticontext (Contexto Múltiple de Método de Entrada)
+gtk.ListStore (Almacén en Lista)
+gtk.RcStyle (Recurso de Estilos)
+gtk.Settings (Opciones)
+gtk.SizeGroup (Grupo de Tamaño)
+gtk.Style (Estilo)
+gtk.TextBuffer (Buffer de texto)
+gtk.TextChildAnchor (Anclaje Hijo de Texto)
+gtk.TextMark (Marca en Texto)
+gtk.TextTag (Etiqueta de Texto)
+gtk.TextTagTable (Tabla de Etiquetas de Texto)
+gtk.TreeModelFilter (Modelo en Árbol Filtrado)
+gtk.TreeModelSort (Modelo en Árbol Ordenado)
+gtk.TreeSelection (Selección de Árbol)
+gtk.TreeStore (Almacén en Árbol)
+gtk.UIManager (Gestor de Interfaces de Usuario)
+gtk.WindowGroup (Grupo de Ventanas)
+gtk.gdk.DragContext (Contexto de Arrastre)
+gtk.gdk.Screen (Pantalla)
+gtk.gdk.Pixbuf (Buffer de píxeles)
+gtk.gdk.Drawable (Dibujable)
| +gtk.gdk.Pixmap (Mapa de Bits)
+gtk.gdk.Image (Imagen)
+gtk.gdk.PixbufAnimation (Animación)
+gtk.gdk.Device (Dispositivo de Entrada)

gobject.GObject
|
+gtk.CellLayout (Disposición de Celdas)
+gtk.Editable (Editable)
+gtk.CellEditable (Editor de Celda)
+gtk.FileChooser (Selección de Ficheros)
+gtk.TreeModel (Modelo en Árbol)
+gtk.TreeDragSource (Fuente de Arrastre en Árbol)
```

```
+gtk.TreeDragDest (Destino de Arrastre en Árbol)
+gtk.TreeSortable (Árbol Ordenable)
```

5.2. Controles sin Ventana

Los siguientes controles no tienen una ventana asociada. Si quieres caputar eventos, tendrás que usar `EventBox`. Mira la sección del control `EventBox`.

```
  gtk.Alignment (Alineador)
  gtk.Arrow (Flecha)
  gtk.Bin (Binario)
  gtk.Box (Caja)
  gtk.Button (Botón)
  gtk.CheckButton (Botón de Activación)
  gtk.Fixed (Fijo)
  gtk.Image (Imagen)
  gtk.Label (Etiqueta)
  gtk.MenuItem (Elemento de Menú)
  gtk.Notebook (Cuaderno de Fichas)
  gtk.Paned (Panel)
  gtk.RadioButton (Botón de Exclusión Mútua)
  gtk.Range (Rango)
  gtk.ScrolledWindow (Ventana de Desplazamiento)
  gtk.Separator (Separador)
  gtk.Table (Tabla)
  gtk.Toolbar (Barra de Herramientas)
  gtk.AspectFrame (Marco de Aspecto)
  gtk.Frame (Marco)
  gtk.VBox (Caja Vertical)
  gtk.HBox (Caja Horizontal)
  gtk.VSeparator (Separador Vertical)
  gtk.HSeparator (Separador Horizontal)
```

Seguiremos nuestra exploración de PyGTK examinando cada control, creando programas de ejemplo simples que los muestren.

Capítulo 6

El Control de Botón

6.1. Botones Normales

Ya hemos visto casi todo lo que hay que ver sobre el control de botón. Es bastante sencillo. Se puede usar la función `gtk.Button()` para crear un botón con una etiqueta pasándole un parámetro de cadena, o uno en blanco si no se especifica dicha cadena. Después depende uno el empaquetar objetos tales como una etiqueta o un pixmap en este nuevo botón. Para ello, se crea una nueva caja, y después se colocan los objetos en ella usando el típico pack_start(). Finalmente se usa add() para colocar la caja dentro del botón.

La función para crear un botón es:

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

si se especifica una etiqueta ésta se usa como texto del botón.Si se especifica stock éste se usa para elegir un icono de serie y una etiqueta para el botón. Los elementos de serie son:

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

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

El programa de ejemplo buttons.py proporciona un ejemplo del uso de `gtk.Button()` para crear un botón con una imagen y una etiqueta en él. Se ha separado el código para crear una caja del resto para que se pueda usar en más programas. Hay más ejemplos del uso de imágenes más adelante en el tutorial. La figura Figura 6.1 muestra la ventana con un botón que incluye una imagen y una etiqueta:

Figura 6.1 Botón con Pixmap y Etiqueta

El código fuente del programa buttons.py es:

```
1   #!/usr/bin/env python
2
3   # ejemplo de inicialización de botones buttons.py
4
5   import pygtk
6   pygtk.require('2.0')
7   import gtk
8
```

```
 9  # Creamos una nueva hbox con una imagen y una etiqueta empaquetadas en ↩
    ella
10  # y devolvemos la caja.
11
12  def xpm_label_box(parent, xpm_filename, label_text):
13      # Crear caja para xpm y etiqueta
14      box1 = gtk.HBox(gtk.FALSE, 0)
15      box1.set_border_width(2)
16
17      # Ahora nos ponemos con la imagen
18      image = gtk.Image()
19      image.set_from_file(xpm_filename)
20
21      # Creamos una etiqueta para el botón
22      label = gtk.Label(label_text)
23
24      # Empaquetamos el pixmap y la etiqueta en la caja
25      box1.pack_start(image, gtk.FALSE, gtk.FALSE, 3)
26      box1.pack_start(label, gtk.FALSE, gtk.FALSE, 3)
27
28      image.show()
29      label.show()
30      return box1
31
32  class Buttons:
33      # Nuestro método habitual de retrollamada (callback)
34      def callback(self, widget, data=None):
35          print "Hello again - %s was pressed" % data
36
37      def __init__(self):
38          # Creamos una ventana nueva
39          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
40
41          self.window.set_title("Image'd Buttons!")
42
43          # Es buena idea hacer esto para todas las ventanas
44          self.window.connect("destroy", lambda wid: gtk.main_quit())
45          self.window.connect("delete_event", lambda a1,a2:gtk.main_quit())
46
47          # Fijamos el ancho de borde de la ventana
48          self.window.set_border_width(10)
49
50          # Creamos un nuevo botón
51          button = gtk.Button()
52
53          # Conectamos la señal "clicked" a nuestra retrollamada
54          button.connect("clicked", self.callback, "cool button")
55
56          # Esto llama a nuestra función de creación de caja
57          box1 = xpm_label_box(self.window, "info.xpm", "cool button")
58
59          # Empaquetamos y mostramos todos los controles
60          button.add(box1)
61
62          box1.show()
63          button.show()
64
65          self.window.add(button)
66          self.window.show()
67
68  def main():
69      gtk.main()
70      return 0
71
```

```
72  if __name__ == "__main__":
73      Buttons()
74      main()
```

Las líneas 12-34 definen la función auxiliar `xpm_label_box`() que crea una caja horizontal con un borde de ancho 2 (lineas 14-15) y le pone una imagen (lineas 22-23) y una etiqueta (linea 26).

Las líneas 36-70 definen la clase `Buttons`. Las líneas 41-70 definen el método de inicialización de instancia que crea una ventana (linea 43), le pone el título (linea 45), le conecta las señales "delete_event" y "destroy" (lineas 48-49). La linea 55 crea el botón sin etiqueta. Su señal "clicked" se conecta al método callback() en la linea 58. La función `xpm_label_box`() se llama en la linea 61 para crear la imagen y la etiqueta que se pondrán en el botón en la linea 64.

La función `xpm_label_box`() podría usarse para empaquetar archivos xpm y etiquetas en cualquier control que pueda ser un contenedor.

El control Botón tiene las siguientes señales:

```
    pressed - se emite cuando el botón del puntero se presiona en el control ↩
        Botón

    released - se emite cuando el botón del puntero se suelta en el control ↩
        Botón

    clicked - se emite cuando el botón del puntero se presiona y luego se
suelta sobre el control Botón

    enter - se emite cuando el puntero entra en el control Botón

    leave - se emite cuando el puntero sale del control Botón
```

6.2. Botones Biestado (Toggle Buttons)

Los Botones Biestado derivan de los botones normales y son muy similares, excepto que siempre están en uno de dos estados, alternándolos con un clic. Puedan estar presionados, y cuando se vuelva a hacer clic, volverán a su estado inicial, levantados. Se hace clic otra vez y volverán a estar presionados.

Los botones Biestado son la base para los botones de activación y los botones de exclusión mútua, y por ello, muchas de las llamadas usadas con los botones biestado son heredados por los botones de activación y los botones de exclusión mútua. Volveremos a destacar este hecho cuando tratemos esos botones.

Creación de un nuevo botón biestado:

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

Como se puede imaginar, estas llamadas funcionan igual que las llamadas al control de botón normal. Si no se especifica etiqueta el botón estará vacio. El texto de la etiqueta se analiza para comprobar si contiene caracteres mnemotécnicos con el prefijo '_'

Para obtener el estado de un botón biestado, incluyendo los botones de exclusión mútua y los botones de activación, se utiliza el mecanismo del ejemplo anterior. Así se comprueba el estado del biestado, llamando al método get_active() del objeto botón biestado. La señal que nos interesa que emiten los botones biestado (el botón biestado, el botón de activación y el botón de exclusión mútua) es la señal "toggled". Para comprobar el estado de estos botones, se configura un manejador de señales para capturar la señal toggled, y se accede a los atributos del objeto para determinar su estado. La retrollamada será parecida a:

```
def toggle_button_callback(widget, data):
    if widget.get_active():
        # Si estamos aqui, el botón biestado está pulsado
    else:
        # Si estamos aqui, el botón biestado está levantado
```

Para forzar el estado de un botón biestado, y de sus hijos, el botón de exclusión mútua y el botón de activación, se utiliza este método:

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

El método anterior puede usarse para forzar el estado del botón biestado, y sus hijos los botones de activación y de exclusión mútua. Especificando un argumento `TRUE` o `FALSE` para el parámetro *`is_active`* indicamos si el botón debería estar pulsado o levantado. Cuando el botón biestado se crea, su valor predeterminado es levantado o `FALSE`.

Hay que fijarse en que, cuando se usa el método set_active(), y cambia realmente el estado del botón, se produce la emisión de las señales "clicked" y "toggled" por éste.

```
toggle_button.get_active()
```

Este método devuelve el estado actual del botón biestado como un valor booleano `TRUE` o `FALSE`.

El programa togglebutton.py proporciona un ejemplo simple del uso de botones biestado. La figura Figura 6.2 ilustra la ventana resultante con el segundo botón biestado activo:

Figura 6.2 Ejemplo de Botón Biestado

El código fuente del programa es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo togglebutton.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ToggleButton:
10     # Nuestra retrollamada.
11     # Los datos pasados a este método se imprimen a la salida estándar
12     def callback(self, widget, data=None):
13         print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active ↩
    ()])
14
15     # Esta retrollamada termina el programa
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return gtk.FALSE
19
20     def __init__(self):
21         # Crear una nueva ventana
22         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         # Establece el título de la ventana
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         # Establece el ancho del borde de la ventana
```

```
32          self.window.set_border_width(20)
33
34          # Crea una caja vertical
35          vbox = gtk.VBox(gtk.TRUE, 2)
36
37          # Inserta vbox en la ventana principal
38          self.window.add(vbox)
39
40          # Crea el primer botón
41          button = gtk.ToggleButton("toggle button 1")
42
43          # cuando se conmuta el botón llamamos el método "callback"
44          # con un puntero a "button" como argumento
45          button.connect("toggled", self.callback, "toggle button 1")
46
47
48          # Insertar el botón 1
49          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
50
51          button.show()
52
53          # Crear el segundo botón
54
55          button = gtk.ToggleButton("toggle button 2")
56
57          # Cuando se conmuta el botón llamamos el método "callback"
58          # con un puntero a "button 2" como argumento
59          button.connect("toggled", self.callback, "toggle button 2")
60          # Insertamos el botón 2
61          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
62
63          button.show()
64
65          # Crear el botón "Quit"
66          button = gtk.Button("Quit")
67
68          # Cuando se pulsa el botón llamamos la función main_quit
69          # y el programa finaliza
70          button.connect("clicked", lambda wid: gtk.main_quit())
71
72          # Insertar el botón de salida
73          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
74
75          button.show()
76          vbox.show()
77          self.window.show()
78
79 def main():
80      gtk.main()
81      return 0
82
83 if __name__ == "__main__":
84      ToggleButton()
85      main()
```

Las lineas interesantes son la 12-13, que definen el método callback() que imprime la etiqueta del botón biestado y su estado cuando es activado. Las lineas 45 y 59 conectan la señal "toggled" de los botones biestado al método `callback`().

6.3. Botones de Activación (Check Buttons)

Los botones de activación heredan muchas propiedades y métodos de los botones biestado vistos anteriormente, pero su apariencia es un poco diferente. En vez de ser botones con texto dentro de ellos,

son pequeñas cajas con un texto a su derecha. Normalmente se utilizan para opciones que pueden estar activadas o desactivadas en las aplicaciones.

El método de creación es similar al de los botones normales.

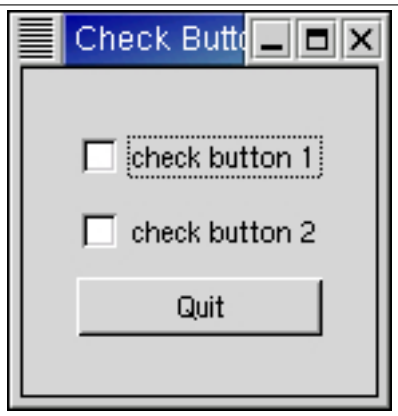
```
check_button = gtk.CheckButton(label=None)
```

Si el argumento *label* se especifica, el método crea un botón de activación con una etiqueta a su lado. El texto *label* de la etiqueta se analiza en busca de caracteres mnemotécnicos con prefijo '_'

Ver y modificar el estado de un botón de activación es igual que en un botón biestado.

El programa checkbutton.py proporciona un ejemplo del uso de los botones de activación. La figura Figura 6.3 ilustra la ventana resultante:

Figura 6.3 Ejemplo de Botón de Activación



El código fuente del programa checkbutton.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo checkbutton.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class CheckButton:
10     # Nuestra retrollamada
11     # Los datos pasados a este método se imprimen en la salida estándar
12     def callback(self, widget, data=None):
13         print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active ↩
    ()])
14
15     # Esta retrollamada termina el programa
16     def delete_event(self, widget, event, data=None):
17         gtk.main_quit()
18         return gtk.FALSE
19
20     def __init__(self):
21         # Crear una nueva ventana
22         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         # Establecer el título de la ventana
25         self.window.set_title("Check Button")
26
27         # Fijar un manejador para delete_event que
28         # salga inmediatamente de GTK.
29         self.window.connect("delete_event", self.delete_event)
30
31         # Fijamos el borde de la ventana
32         self.window.set_border_width(20)
```

```
33
34          # Creamos una caja vertical vbox
35          vbox = gtk.VBox(gtk.TRUE, 2)
36
37          # Insertamos vbox en la ventana principal
38          self.window.add(vbox)
39
40          # Creamos el primer botón
41          button = gtk.CheckButton("check button 1")
42
43          # Cuando se conmuta el botón llamamos el método "callback"
44          # con un puntero a "button" como argumento
45          button.connect("toggled", self.callback, "check button 1")
46
47
48          # Insertamos el botón 1
49          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
50
51          button.show()
52
53          # Creamos un segundo botón
54
55          button = gtk.CheckButton("check button 2")
56
57          # Cuando se conmuta el botón llamamos el método "callback"
58          # con un puntero a "button 2" como argumento
59          button.connect("toggled", self.callback, "check button 2")
60          # Insertamos el botón 2
61          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
62
63          button.show()
64
65          # Creamos el botón "Quit"
66          button = gtk.Button("Quit")
67
68          # Cuando se pulsa el botón llamamos la función mainquit
69          # y el programa termina
70          button.connect("clicked", lambda wid: gtk.main_quit())
71
72          # Insertamos el botón de salida
73          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
74
75          button.show()
76          vbox.show()
77          self.window.show()
78
79 def main():
80      gtk.main()
81      return 0
82
83 if __name__ == "__main__":
84      CheckButton()
85      main()
```

6.4. Botones de Exclusión Mútua (Radio Buttons)

Los botones de exclusión mútua son similares a los botones de activación excepto que se agrupan, de tal forma que sólo uno puede estar seleccionado/pulsado en un momento dado. Esto es bueno para aquellas situaciones en las que la aplicación necesita seleccionar un valor entre una pequeña lista de opciones.

La creación de un nuevo botón de exclusión mútua se hace con la siguiente llamada:

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

Es necesario fijarse en el argumento adicional de esta llamada. Los botones de exclusión mútua requieren un grupo *group* para funcionar correctamente.La primera llamada a `gtk.RadioButton()` debe pasarle `None` en el primer argumento y entonces se creará un nuevo grupo de botones de exclusión mútua con el nuevo botón de exclusión mútua como su único miembro.

Para añadir más botones de exclusión mútua a un grupo se pasa una referencia a un botón de exclusión mútua en *group* en las llamadas posteriores a `gtk.RadioButton()`.

Si se especifica un argumento *label* dicho texto se analizará para comprobar la presencia de caracteres mnemotécnicos con prefijo '_'.

También es buena idea especificar explícitamente qué botón será el que esté activado por defecto mediante:

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

Esto se describe en la sección de los botones biestado, y funciona exactamente de la misma manera. Una vez que los botones de exclusión mútua se agrupan, sólo uno de los pertenecientes al grupo puede estar activo al mismo tiempo. Si el usuario hace clic en un botón de exclusión mútua, y luego en otro, el primer botón de exclusión mútua emitirá una señal "toggled" (para informar de que va a estar inactivo), y luego el segundo botón emitirá su señal "toggled" (para informar de que va a estar activo).

El programa de ejemplo radiobuttons.py crea un grupo de botones de exclusión mútua con tres botones. La figura Figura 6.4 ilustra la ventana resultante:

Figura 6.4 Ejemplo de Botones de Exclusión Mútua

The source code for the example program is:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo radiobuttons.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class RadioButtons:
10     def callback(self, widget, data=None):
11         print "%s was toggled %s" % (data, ("OFF", "ON")[widget.get_active ↩
    ()])
12
13     def close_application(self, widget, event, data=None):
14         gtk.main_quit()
15         return gtk.FALSE
16
17     def __init__(self):
18         self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

```
19
20          self.window.connect("delete_event", self.close_application)
21
22          self.window.set_title("radio buttons")
23          self.window.set_border_width(0)
24
25          box1 = gtk.VBox(gtk.FALSE, 0)
26          self.window.add(box1)
27          box1.show()
28
29          box2 = gtk.VBox(gtk.FALSE, 10)
30          box2.set_border_width(10)
31          box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
32          box2.show()
33
34          button = gtk.RadioButton(None, "radio button1")
35          button.connect("toggled", self.callback, "radio button 1")
36          box2.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
37          button.show()
38
39          button = gtk.RadioButton(button, "radio button2")
40          button.connect("toggled", self.callback, "radio button 2")
41          button.set_active(gtk.TRUE)
42          box2.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
43          button.show()
44
45          button = gtk.RadioButton(button, "radio button3")
46          button.connect("toggled", self.callback, "radio button 3")
47          box2.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
48          button.show()
49
50          separator = gtk.HSeparator()
51          box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 0)
52          separator.show()
53
54          box2 = gtk.VBox(gtk.FALSE, 10)
55          box2.set_border_width(10)
56          box1.pack_start(box2, gtk.FALSE, gtk.TRUE, 0)
57          box2.show()
58
59          button = gtk.Button("close")
60          button.connect_object("clicked", self.close_application, self. ↩
    window,
61                                None)
62          box2.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
63          button.set_flags(gtk.CAN_DEFAULT)
64          button.grab_default()
65          button.show()
66          self.window.show()
67
68  def main():
69      gtk.main()
70      return 0
71
72  if __name__ == "__main__":
73      RadioButtons()
74      main()
```

El código es bastante simple de seguir. Las lineas 63-64 hacen que el botón "close" sea el control por defecto, de manera que, al pulsar la tecla "Enter" cuando la ventana está activa, el botón "close" emitirá la señal "clicked".

Capítulo 7

Ajustes

GTK tiene varios controles que pueden ser ajustados visualmente por el usuario usando el ratón o el teclado, tales como los controles de rango, descritos en la sección Controles de Rango. También hay unos cuantos controles que visualizan una parte ajustable de un área de datos mayor, tales como el control de texto y el control de vista.

Obviamente, una aplicación necesita ser capaz de reaccionar ante los cambios que el usuario realiza en los controles de rango. Una forma de hacer esto sería que cada control emitiera su propio tipo de señal cuando su ajuste cambiara y, o bien pasar el nuevo valor al manejador de señal, o requerir que se mire dentro de la estructura de datos del control para ver el nuevo valor. Pero puede que también se quiera conectar los ajustes de varios controles juntos, para que ajustando uno se ajusten los otros. El ejemplo más obvio de esto es conectar una barra de desplazamiento a una vista o a un área de texto desplazable. Si cada control tuviera su propia manera de manipular el valor del ajuste, entonces el programador tendría que escribir sus propios manejadores de señales para traducir entre la salida de la señal de un control y la entrada del método de ajuste de otro control.

GTK soluciona este problema mediante el objeto `Adjustment`, que no es un control sino una manera de que los controles almacenen y pasen la información de ajuste de una forma abstracta y flexible. El uso más obvio de `Adjustment` es almacenar los parámetros de configuración y los valores de los controles de rango, tales como las barras de desplazamiento y los controles de escala. Sin embargo, como la clase `Adjustments` deriva de `Object`, también tiene unas características especiales más alla de ser estructuras de datos normales. La más importante es que pueden emitir señales, como los controles, y estas señales no sólo pueden ser usadas para permitir a tus programas reaccionar a la entrada de usuario en controles ajustables, sino que pueden propagar valores de ajuste de una forma transparente entre controles ajustables.

Se verá como encajan los ajustes en todo el sistema cuando se vean los controles que los incorporan: Barras de Progreso, Vistas, Ventanas de Desplazamiento, y otros.

7.1. Creación de un Ajuste

Muchos de los controles que usan ajustes lo hacen automáticamente, pero más tarde se mostrarán casos en los que puede ser necesario crearlos. Es posible crear un ajuste usando:

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

El argumento *value* es el valor inicial que se quiere dar al ajuste, y normalmente corresponde a la posición superior o la posición más a la izquierda de un control ajustable. El argumento *lower* especifica el valor más bajo que puede tomar el ajuste *adjustment*. El argumento *step_incr* especifica el incremento más pequeño de los dos incrementos por los que el usuario puede cambiar el valor, mientras que el argumento *page_incr* es el más grande de los dos. El argumento *page_size* normalmente se corresponde de alguna manera con el área visible de un control desplazable. El argumento *upper* se usa para representar la coordenada inferior o la más a la derecha de un control incluido en otro control desplazable. Por tanto no es siempre el número más grande que el valor puede tomar, ya que el *page_size* de tales controles normalmente es distinto de cero.

7.2. Utilización de Ajustes de la Forma Fácil

Los controles ajustables pueden dividirse más o menos en aquellos que usan y requieren unidades específicas para estos valores, y aquellos que los tratan como número arbitrarios. El grupo que trata los valores como números arbitrarios incluye los controles de rango (barras de desplazamiento y escalas, la barra de progreso y los botones de aumentar/disminuir). Todos estos controles normalmente se ajustan directamente por el usuario con el ratón o el teclado. Tratarán los valores inferior y superior de un ajuste como un rango dentro del cual el usuario puede manipular el valor del ajuste. Por defecto, solo modificarán el valor de un ajuste.

El otro grupo incluye el control de texto, el control de vista, el control de lista compuesta y el control de ventana de desplazamiento. Todos ellos usan valores de píxeles para sus ajustes. Estos controles normalmente se ajustan indirectamente usando barras de desplazamiento. Aunque todos los controles que usan ajustes pueden crear sus propios ajustes o usar los que se les proporcione, normalmente se les querrá dejar la tarea de crear sus propios ajustes. Habitualmente, sobreescribirán todos los valores de configuración de los ajustes que se les proporcionen, salvo el propio valor de ajuste, aunque los resultados son, en general, impredecibles. (Lo que significa que se tendrá que leer el código fuente para descubrirlo, y puede variar entre controles).

Ahora, probablemente se piense... puesto que los controles de texto y las vistas insisten en establecer todos sus parámetros de configuración excepto el valor de ajuste mientras que las barras de desplazamiento solamente tocan el valor del ajuste, si se comparte un objeto ajuste entre una barra de desplazamiento y un control de texto, al manipular la barra de desplazamiento, ¿se ajustará automágicamente el control de texto? ¡Por supuesto que lo hará! De la siguiente manera:

```
# crea sus propios ajustes
viewport = gtk.Viewport()
# usa los ajustes recién creados para la barra de desplazamiento también
vscrollbar = gtk.VScrollbar(viewport.get_vadjustment())
```

7.3. Interioridades de un Ajuste

Bien, se pensará, eso está bien, pero ¿qué sucede si se desea crear unos manejadores propios que respondan cuando el usuario ajuste un control de rango o un botón aumentar/disminuir, y cómo se obtiene el valor de un ajuste en estos manejadores? Para contestar a estas preguntas y a otras más, empezaremos echando un vistazo a los atributos de la propia clase `gtk.Adjustment`:

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

Dada una instancia *`adj`* de la clase `gtk.Adjustment`, cada uno de los atributos se obtienen o modifican usando *`adj.lower`*, *`adj.value`*, etc.

Ya que, cuando se determina el valor de un ajuste, generalmente también se quiere que el cambio afecte a todos los controles que usan este ajuste, PyGTK proporciona un método para hacer esto mismo:

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

Como se mencionó anteriormente, `Adjustment` es una subclase de `Object`, al igual que los demás controles, y, por tanto, es capaz de emitir señales. Esto es la causa, claro está, de por qué las actualizaciones ocurren automágicamente cuando se comparte un objeto ajuste entre una barra de desplazamiento y otro control ajustable; todos los controles ajustables conectan manejadores de señales a la señal "value_changed" de sus ajustes, como podría hacerlo cualquier programa. Esta es la definición de la retrollamada de esta señal:

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

Los diversos controles que usan el objeto `Adjustment` emitirán esta señal en un ajuste siempre que cambien su valor. Esto ocurre tanto cuando el usuario hace que el deslizador se mueva en un control de rango, como cuando el programa explícitamente cambia el valor con el método set_value(). Así, por

ejemplo, si se tiene un control de escala, y se quiere que cambie la rotación de una imagen siempre que su valor cambie, se podría crear una retrollamada como esta:

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

y conectarla al ajuste del control de escala así:

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

¿Y qué pasa cuando un control reconfigura los campos *upper* (superior) o *lower* (inferior) de su ajuste, tal y como sucede cuando un usario añade más texto al control de texto? En este caso, se emite la señal "changed", que es así:

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

Los controles `Range` normalmente conectan un manejador para esta señal, el cual cambia su apariencia para reflejar el cambio - por ejemplo, el tamaño del deslizador de una barra de desplazamiento aumentará o se reducirá en proporción inversa a la diferencia entre el valor superior e inferior de su ajuste.

Probablemente nunca será necesario que se haga la conexión de un manejador a esta señal, salvo que se esté escribiendo un nuevo tipo de control de rango. En cualquier caso, si se cambia directamente alguno de los valores de un `Adjustment`, se debe emitir esta señal para reconfigurar los controles que lo usan. Tal que así:

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

Capítulo 8

Controles de Rango

La categoría de los controles de rango incluye el ubícuo control de barra de desplazamiento y el menos común control de escala. Aunque estos dos tipos de controles se usan generalmente para propósitos diferentes, son bastante similares en función e implementación. Todos los controles de rango comparten un conjunto de elementos gráficos, cada uno de los cuales tiene su propia ventana X y recibe eventos. Todos ellos contienen una guía o canal y un deslizador. Arrastrar el deslizador con el puntero del ratón hace que se mueva hacia adelante y hacia atrás en el canal, mientras que haciendo clic en el canal avanza el deslizador hacia la localización del clic, ya sea completamente, o con una cantidad designada, dependiendo del botón del ratón que se use.

Como se mencionó en Adjustments más arriba, todos los controles de rango están asociados con un objeto ajuste, a partir del cual se calcula la longitud del deslizador y su posición con respecto al canal. Cuando el usuario manipula el deslizador, el control de rango cambiará el valor del ajuste.

8.1. Barras de Desplazamiento

Estas son las barras de desplazamiento estándar. Deberían usarse únicamente para desplazar otro control, tal como una lista, una caja de texto, o una vista (viewport), aunque, generalmente, es más fácil de usar la ventana de desplazamiento en la mayoría de los casos. Para otros propósitos, se deberían usar los controles de escala, ya que son más agradables y tienen más funciones.

Existen tipos separados para las barras de desplazamiento horizontales y verticales. No hay demasiado que decir sobre ellos. Los puedes crear con los siguientes métodos:

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

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

y eso es todo. El argumento *adjustment* puede ser, o bien una referencia a un Adjustment existente, o bien nada, en cuyo caso se creará uno. Especificar nada puede ser útil en el caso en el que se quiera pasar el ajuste recién creado al constructor de otro control que lo configurará por uno, tal como podría ocurrir con una caja de texto.

8.2. Controles de Escala

Los controles `Scale` (Escala) se usan para permitir al usuario seleccionar y manipular visualmente un valor dentro de un rango específico. Se puede usar un control de escala, por ejemplo, para ajustar el nivel de zoom en una previsualización de una imagen, o para controlar el brillo de un color, o para especificar el número de minutos de inactividad antes de que el protector de pantalla se active.

8.2.1. Creación de un Control de Escala

Al igual que con las barras de desplazamiento, hay controles separados para los controles de escala horizontales y verticales. (La mayoría de los programadres parecen usar los controles de escala horizontales.) Ya que esencialmente funcionan de la misma manera, no hay necesidad de tratarlos por separado aquí. Los siguientes métodos crean controles de escala verticales y horizontales, respectivamente:

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

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

El argumento *adjustment* puede ser bien un ajuste que ya haya sido creado con `gtk.Adjustment()`, o bien nada, en cuyo caso se crea un Adjustment anónimo con todos sus valores puestos a 0.0 (lo cual no es demasiado útil). Para evitar confusiones, probablemente sea mejor crear un ajuste con un valor `page_size` de 0.0 para que su valor `upper` realmente corresponda con el valor más alto que el usuario puede seleccionar. (Si esto resulta confuso, la sección sobre Ajustes da una explicación detallada sobre qué hacen exactamente los ajustes y cómo crearlos y manipularlos.)

8.2.2. Métodos y Señales (bueno, al menos métodos)

Los controles de escala pueden visualizar su valor como un número al lado del canal. El comportamiento por defecto es mostrar el valor, pero esto se puede cambiar con el método:

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

Como habrás imaginado, *draw_value* (representar valor) es o `TRUE` o `FALSE`, con consecuencias predecibles para cualquiera de los dos.

El valor que muestra un control de escala se redondea a un valor decimal por defecto, tal y como se hace con el campo valor en su Adjustment (Ajuste). Se puede cambiar esto con:

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

donde *digits* (dígitos) es el número de posiciones decimales que se representarán. Se puede poner el número que se desee, pero no se representarán en pantalla más de 13 dígitos decimales.

Finalmente, el valor se puede mostrar en diferentes posiciones relativas al canal:

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

El argumento *pos* (posición) puede tomar uno de los siguientes valores:

```
POS_LEFT
POS_RIGHT
POS_TOP
POS_BOTTOM
```

Si pones el valor en un "lado" del canal (por ejemplo, en la parte de arriba o de abajo de un control de escala horizontal), entonces seguirá al deslizador en su desplazamiento hacia arriba y hacia abajo del canal.

8.3. Métodos Comunes de los Rangos

La clase `Range` es bastante complicada internamente, pero, como todas las clases base de los controles, la mayoría de su complejidad solo resulta de interés si se quiere trastear con ellos. Además, la mayoría de los métodos y señales que define sólo son útiles al escribir controles derivados. Hay, en cualquier caso, unos cuantos métodos útiles que funcionarán con todos los controles de rango.

8.3.1. Establecimiento de la Política de Actualización

La "política de actualización" de un control de rango define en qué puntos de la interacción con el usuario se cambiará el campo de valor de su Adjustment y emitirá la señal "value_changed" en este Adjustment. Las políticas de actualización son:

UPDATE_CONTINUOUS Es es el valor predeterminado. La señal "value_changed" se emite contínuamente, por ejemplo, cada vez que el deslizador se mueve, incluso en las cantidades mas minúsculas.

UPDATE_DISCONTINUOUS La señal "value_changed" sólo se mite una vez que el deslizador ha parado de moverse y el usuario ha soltado el botón del ratón.

UPDATE_DELAYED La señal "value_changed" se emite cuando el usuario suelta el botón del ratón, o si el deslizador deja de moverse durante un corto período de tiempo.

La política de actualización de un control de rango puede cambiarse con este método:

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

8.3.2. Obtención y Cambio de Ajustes

La obtención y cambio del ajuste de un control de rango se puede hacer sobre la marcha, como era predecible, con:

```
adjustment = range.get_adjustment()

range.set_adjustment(adjustment)
```

El método get_adjustment() devuelve una referencia al *adjustment* que está conectado al rango.

El método set_adjustment() no hace absolutamente nada si se le pasa el *adjustment* que el *range* ya esté utilizando, independientemente de que se le hayan cambiado alguno de sus campos o no. Si se le pasa un nuevo Adjustment, se perderá la referencia al antiguo si existía (posiblemente se destruirá), se conectarán las señales apropiadas al nuevo, y se recalculará el tamaño y/o posición del deslizador y se repintará si es necesario. Como se mencionó en la sección de ajustes, si se desea reutilizar el mismo Adjustment, cuando se modifiquen sus valores directamente se debe emitir la señal "changed" desde él, como por ejemplo:

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

8.4. Atajos de Teclas y Ratón

Todos los controles de rango de GTK reaccionan a clics de ratón más o menos de la misma forma. Haciendo clic con el botón-1 en el canal hace que el valor *page_increment* del ajuste se añada o se reste a su *value*, y que el deslizador se mueva de forma acorde. Haciendo clic con el botón-2 en el canal hará que el deslizador salte al punto donde se ha hecho clic. Haciendo clic con cualquier botón en las flechas de una barra de desplazamiento se hará que el valor *value* de su ajuste cambie de una vez tanto como diga su propiedad *step_increment*.

Las barras de desplazamiento no pueden recibir el foco, por lo que no tienen atajos de teclado. Los atajos de teclado de otros controles de rango (que por supuesto sólo están ativos cuando el control tiene el foco) no se diferencian entre controles de rango horizontales y verticales.

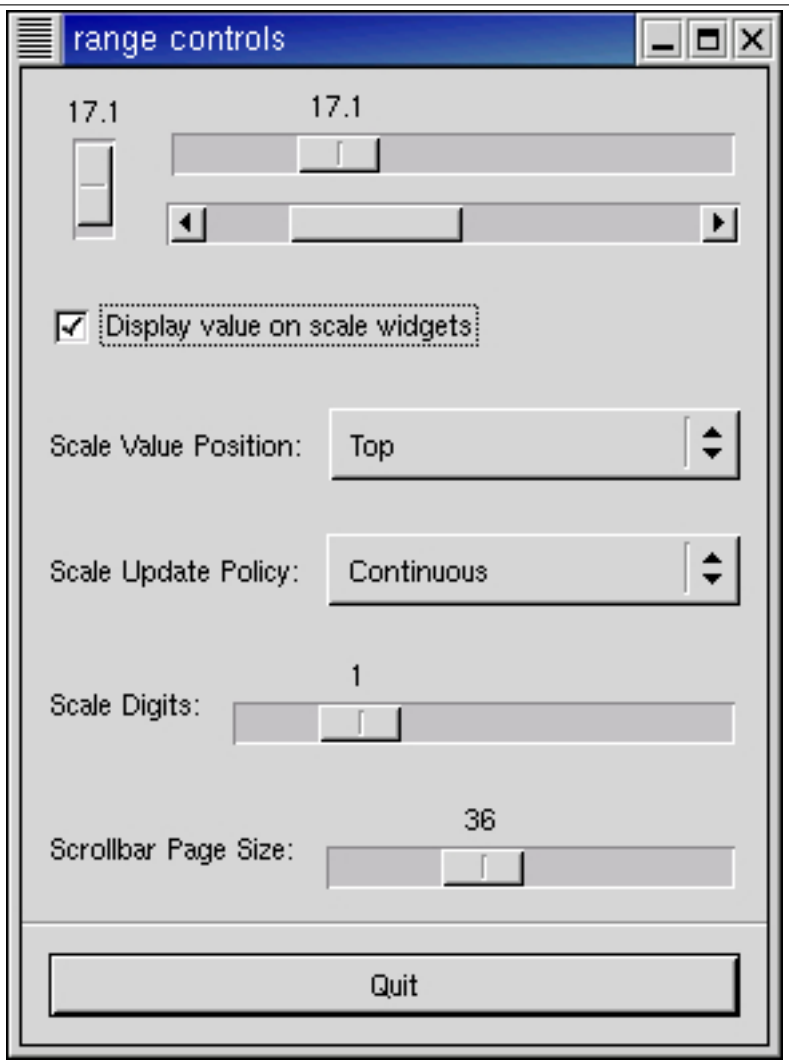
Todos los controles de rango pueden manejarse con las teclas de flecha izquierda, flecha derecha, flecha arriba y flecha abajo , así como con las teclas **Página Anterior** y **Página Siguiente**. Las flechas mueven el deslizador en cantidades iguales a *step_increment*, mientras que **Página Anterior** y **Página Siguiente** lo mueven en cantidades de *page_increment*.

El usuario también puede mover el deslizador directamente a un extremo u otro del canal usando el teclado. Esto se hace con las teclas **Inicio** y **Fin** .

8.5. Ejemplo de Control de Rango

El programa de ejemplo (rangewidgets.py) muestra una ventana con tres controles de rango todos conectados al mismo ajuste, y un par de controles para modificar algunos de los parámetros mencionados más arriba y en la sección de ajustes, por lo que se puede ver cómo afectan a la manera en la que estos controles se comportan para el usuario. La figura Figura 8.1 muestra el resultado de ejecutar el programa:

Figura 8.1 Ejemplo de Controles de Rango



El código fuente de rangewidgets.py es:

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

```
25          # Fijar la posición del valor para ambos ajustes
26          self.hscale.set_value_pos(pos)
27          self.vscale.set_value_pos(pos)
28
29      def cb_update_menu_select(self, item, policy):
30          # Establecer la política de actualización para ambos controles
31          self.hscale.set_update_policy(policy)
32          self.vscale.set_update_policy(policy)
33
34      def cb_digits_scale(self, adj):
35          # Fijar el número de posiciones decimales a las que se ajusta el ↩
    valor
36          self.hscale.set_digits(adj.value)
37          self.vscale.set_digits(adj.value)
38
39      def cb_page_size(self, get, set):
40          # Fijar el valor del tamaño de página y de su incremento
41          # para el ajuste al valor especificado por la escala "Page Size"
42          set.page_size = get.value
43          set.page_incr = get.value
44          # Ahora emitir la señal "changed" para reconfigurar todos los ↩
    controles
45          # que están ligados a este ajuste
46          set.emit("changed")
47
48      def cb_draw_value(self, button):
49          # Activar o desactivar la representación del valor en función del
50          # estado del botón de activación
51          self.hscale.set_draw_value(button.get_active())
52          self.vscale.set_draw_value(button.get_active())
53
54      # crea la ventana de ejemplo
55
56      def __init__(self):
57          # Creación de la ventana principal
58          self.window = gtk.Window (gtk.WINDOW_TOPLEVEL)
59          self.window.connect("destroy", lambda w: gtk.main_quit())
60          self.window.set_title("range controls")
61
62          box1 = gtk.VBox(gtk.FALSE, 0)
63          self.window.add(box1)
64          box1.show()
65
66          box2 = gtk.HBox(gtk.FALSE, 10)
67          box2.set_border_width(10)
68          box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
69          box2.show()
70
71          # value, lower, upper, step_increment, page_increment, page_size
72          # Obsérvese que el valor page_size solamente es significativo
73          # para controles de barra de desplazamiento y que el valor más alto ↩
     posible es
74          # (upper - page_size).
75          adj1 = gtk.Adjustment(0.0, 0.0, 101.0, 0.1, 1.0, 1.0)
76
77          self.vscale = gtk.VScale(adj1)
78          scale_set_default_values(self.vscale)
79          box2.pack_start(self.vscale, gtk.TRUE, gtk.TRUE, 0)
80          self.vscale.show()
81
82          box3 = gtk.VBox(gtk.FALSE, 10)
83          box2.pack_start(box3, gtk.TRUE, gtk.TRUE, 0)
84          box3.show()
85
```

```
 86         # Reutilizamos el mismo ajuste
 87         self.hscale = gtk.HScale(adj1)
 88         self.hscale.set_size_request(200, 30)
 89         scale_set_default_values(self.hscale)
 90         box3.pack_start(self.hscale, gtk.TRUE, gtk.TRUE, 0)
 91         self.hscale.show()
 92
 93         # Reutilizamos el mismo ajuste otra vez
 94         scrollbar = gtk.HScrollbar(adj1)
 95         # Obsérvese como esto hace que las escalas se actualicen
 96         # continuamente cuando se mueve la barra de desplazamiento
 97         scrollbar.set_update_policy(gtk.UPDATE_CONTINUOUS)
 98         box3.pack_start(scrollbar, gtk.TRUE, gtk.TRUE, 0)
 99         scrollbar.show()
100
101         box2 = gtk.HBox(gtk.FALSE, 10)
102         box2.set_border_width(10)
103         box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
104         box2.show()
105
106         # Un botón de activación para definir se se muestra el valor o no
107         button = gtk.CheckButton("Display value on scale widgets")
108         button.set_active(gtk.TRUE)
109         button.connect("toggled", self.cb_draw_value)
110         box2.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
111         button.show()
112
113         box2 = gtk.HBox(gtk.FALSE, 10)
114         box2.set_border_width(10)
115
116         # Un menú de opciones para cambiar la posición del valor
117         label = gtk.Label("Scale Value Position:")
118         box2.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
119         label.show()
120
121         opt = gtk.OptionMenu()
122         menu = gtk.Menu()
123
124         item = make_menu_item ("Top", self.cb_pos_menu_select, gtk.POS_TOP)
125         menu.append(item)
126
127         item = make_menu_item ("Bottom", self.cb_pos_menu_select,
128                                gtk.POS_BOTTOM)
129         menu.append(item)
130
131         item = make_menu_item ("Left", self.cb_pos_menu_select, gtk. ↩
    POS_LEFT)
132         menu.append(item)
133
134         item = make_menu_item ("Right", self.cb_pos_menu_select, gtk. ↩
    POS_RIGHT)
135         menu.append(item)
136
137         opt.set_menu(menu)
138         box2.pack_start(opt, gtk.TRUE, gtk.TRUE, 0)
139         opt.show()
140
141         box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
142         box2.show()
143
144         box2 = gtk.HBox(gtk.FALSE, 10)
145         box2.set_border_width(10)
146
147         # Otro menú de opciones más, esta vez para la política de  ↩
```

```
    actualización
148         # de los controles de escala
149         label = gtk.Label("Scale Update Policy:")
150         box2.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
151         label.show()
152 
153         opt = gtk.OptionMenu()
154         menu = gtk.Menu()
155 
156         item = make_menu_item("Continuous", self.cb_update_menu_select,
157                               gtk.UPDATE_CONTINUOUS)
158         menu.append(item)
159 
160         item = make_menu_item ("Discontinuous", self.cb_update_menu_select,
161                                gtk.UPDATE_DISCONTINUOUS)
162         menu.append(item)
163 
164         item = make_menu_item ("Delayed", self.cb_update_menu_select,
165                                gtk.UPDATE_DELAYED)
166         menu.append(item)
167 
168         opt.set_menu(menu)
169         box2.pack_start(opt, gtk.TRUE, gtk.TRUE, 0)
170         opt.show()
171 
172         box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
173         box2.show()
174 
175         box2 = gtk.HBox(gtk.FALSE, 10)
176         box2.set_border_width(10)
177 
178         # Un control HScale para ajustar el número de dígitos en las  ↩
    escalas
179         # de ejemplo
180         label = gtk.Label("Scale Digits:")
181         box2.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
182         label.show()
183 
184         adj2 = gtk.Adjustment(1.0, 0.0, 5.0, 1.0, 1.0, 0.0)
185         adj2.connect("value_changed", self.cb_digits_scale)
186         scale = gtk.HScale(adj2)
187         scale.set_digits(0)
188         box2.pack_start(scale, gtk.TRUE, gtk.TRUE, 0)
189         scale.show()
190 
191         box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
192         box2.show()
193 
194         box2 = gtk.HBox(gtk.FALSE, 10)
195         box2.set_border_width(10)
196 
197         # Y un último control HScale para ajustar el tamaño de página
198         # de la barra de desplazamiento.
199         label = gtk.Label("Scrollbar Page Size:")
200         box2.pack_start(label, gtk.FALSE, gtk.FALSE, 0)
201         label.show()
202 
203         adj2 = gtk.Adjustment(1.0, 1.0, 101.0, 1.0, 1.0, 0.0)
204         adj2.connect("value_changed", self.cb_page_size, adj1)
205         scale = gtk.HScale(adj2)
206         scale.set_digits(0)
207         box2.pack_start(scale, gtk.TRUE, gtk.TRUE, 0)
208         scale.show()
209 
```

```
210         box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
211         box2.show()
212
213         separator = gtk.HSeparator()
214         box1.pack_start(separator, gtk.FALSE, gtk.TRUE, 0)
215         separator.show()
216
217         box2 = gtk.VBox(gtk.FALSE, 10)
218         box2.set_border_width(10)
219         box1.pack_start(box2, gtk.FALSE, gtk.TRUE, 0)
220         box2.show()
221
222         button = gtk.Button("Quit")
223         button.connect("clicked", lambda w: gtk.main_quit())
224         box2.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
225         button.set_flags(gtk.CAN_DEFAULT)
226         button.grab_default()
227         button.show()
228         self.window.show()
229
230 def main():
231     gtk.main()
232     return 0
233
234 if __name__ == "__main__":
235     RangeWidgets()
236     main()
```

Se debe advertir que el programa no llama al método connect() para el evento "delete_event", sino solamente a la señal "destroy". Esto seguirá realizando la acción deseada, puesto que un evento "delete_event" sin tratar resultará en una señal "destroy" enviada a la ventana.

Capítulo 9

Miscelánea de Controles

9.1. Etiquetas

Las etiquetas son objetos de la clase `Label` y se usan mucho en GTK, siendo además relativamente simples. Los objetos `Label` no emiten señales ya que no tienen una ventana X asociada. Si se necesita capturar señales, o hacer recorte, se deben colocar dentro de un control EventBox (Caja de Eventos) o de un control Button (Botón).

Para crear una nueva etiqueta se usa:

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

El único argumento es la cadena de texto que se quiere que visualice la etiqueta. Para cambiar el texto de la etiqueta después de la creación se usa el método:

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

label es la etiqueta que se creó previamente, y *str* es la nueva cadena. El espacio que necesite la nueva cadena se ajustará automáticamente si es necesario. Se pueden hacer etiquetas multilinea poniendo saltos de linea en la cadena de la etiqueta.

Para obtener la cadena actual, se usa:

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

label es la etiqueta que se ha creado, y *str* es la cadena que devuelve. El texto de una etiqueta en *label* se puede justificar usando:

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

Los valores posibles de *jtype* son:

```
JUSTIFY_LEFT # valor predeterminado
JUSTIFY_RIGHT
JUSTIFY_CENTER
JUSTIFY_FILL # no funciona
```

El control de etiqueta también es capaz de partir el texto automáticamente. Esto se puede activar usando:

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

El argumento *wrap* puede tomar un valor `TRUE` o `FALSE`.

Si se quiere subrayado en la etiqueta, entonces se puede establecer un patrón para la etiqueta:

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

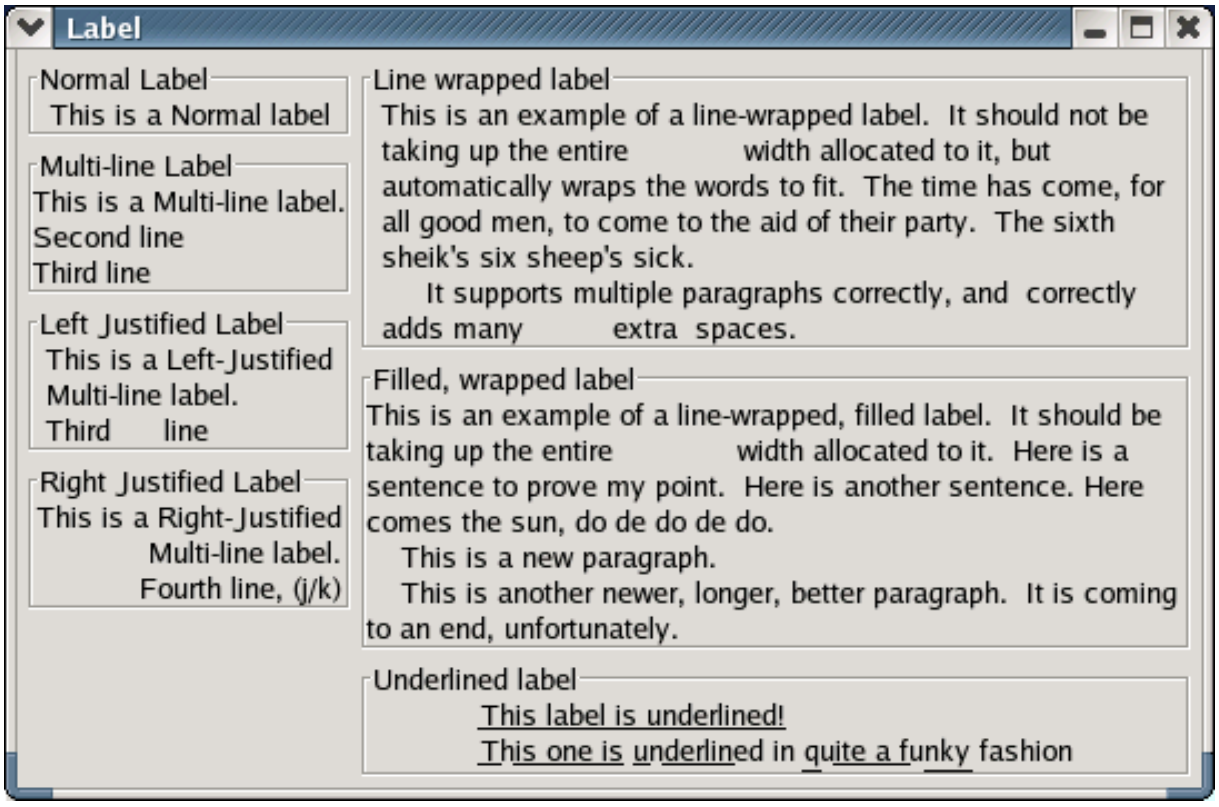
El argumento *pattern* (patrón) indica cómo se verá el subrayadao. Consiste en una cadena de signos de subrayado y caracteres de espacio. Un signo de subrayado indica que el caracter correspondiente en la etiqueta debe estar subrayado. Por ejemplo, la cadena "__ ___" subrayaría los primeros dos caracteres y los caracteres cuarto y quinto. Si sólo se quiere un atajo subrayado ("mnemónico") en la etiqueta, se debería usar set_text_with_mnemonic (*str*), no set_pattern().

El programa de ejemplo label.py es un ejemplo corto para ilustrar estos métodos. Este ejemplo hace uso del control Frame (Marco) para demostrar mejor los estilos de etiqueta. Se puede ignorar esto por ahora ya que el control `Frame` (Marco) se explica después.

En GTK+ 2.0, el texto de la etiqueta puede contener marcas para el tipo de letra y otros atributos del texto, y las etiquetas pueden ser seleccionables (para copiar y pegar). Estas características avanzadas no se explican aqui.

La figura Figura 9.1 ilustra el resultado de ejecutar el programa de ejemplo:

Figura 9.1 Ejemplos de Etiquetas



El código fuente de label.py es:

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

```
24          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
25
26          frame = gtk.Frame("Multi-line Label")
27          label = gtk.Label("This is a Multi-line label.\nSecond line\n"
28                            "Third line")
29          frame.add(label)
30          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
31
32          frame = gtk.Frame("Left Justified Label")
33          label = gtk.Label("This is a Left-Justified\n"
34                            "Multi-line label.\nThird      line")
35          label.set_justify(gtk.JUSTIFY_LEFT)
36          frame.add(label)
37          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
38
39          frame = gtk.Frame("Right Justified Label")
40          label = gtk.Label("This is a Right-Justified\nMulti-line label.\n"
41                            "Fourth line, (j/k)")
42          label.set_justify(gtk.JUSTIFY_RIGHT)
43          frame.add(label)
44          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
45
46          vbox = gtk.VBox(gtk.FALSE, 5)
47          hbox.pack_start(vbox, gtk.FALSE, gtk.FALSE, 0)
48          frame = gtk.Frame("Line wrapped label")
49          label = gtk.Label("This is an example of a line-wrapped label.  It  ↩
    "
50                            "should not be taking up the entire  ↩
                 "
51                            "width allocated to it, but automatically "
52                            "wraps the words to fit.  "
53                            "The time has come, for all good men, to come  ↩
    to "
54                            "the aid of their party.  "
55                            "The sixth sheik's six sheep's sick.\n"
56                            "     It supports multiple paragraphs  ↩
    correctly, "
57                            "and  correctly   adds "
58                            "many          extra  spaces. ")
59          label.set_line_wrap(gtk.TRUE)
60          frame.add(label)
61          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
62
63          frame = gtk.Frame("Filled, wrapped label")
64          label = gtk.Label("This is an example of a line-wrapped, filled  ↩
    label.  "
65                            "It should be taking "
66                            "up the entire              width allocated to ↩
     it.  "
67                            "Here is a sentence to prove "
68                            "my point.  Here is another sentence. "
69                            "Here comes the sun, do de do de do.\n"
70                            "    This is a new paragraph.\n"
71                            "    This is another newer, longer, better "
72                            "paragraph.  It is coming to an end, "
73                            "unfortunately.")
74          label.set_justify(gtk.JUSTIFY_FILL)
75          label.set_line_wrap(gtk.TRUE)
76          frame.add(label)
77          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
78
79          frame = gtk.Frame("Underlined label")
80          label = gtk.Label("This label is underlined!\n"
81                            "This one is underlined in quite a funky  ↩
```

```
    fashion")
82          label.set_justify(gtk.JUSTIFY_LEFT)
83          label.set_pattern(
84              "_____ _ _____ _ _____  __ _____  ↩
    ___")
85          frame.add(label)
86          vbox.pack_start(frame, gtk.FALSE, gtk.FALSE, 0)
87          self.window.show_all ()
88
89  def main():
90      gtk.main()
91      return 0
92
93  if __name__ == "__main__":
94      Labels()
95      main()
```

Obsérvese que la etiqueta "Filled, wrapped label" no tiene justificación completa (fill justified).

9.2. Flechas (Arrow)

El control `Arrow` (Flecha) dibuja la cabeza de una flecha, apuntando a un número de direcciones posibles y con un número de estilos posibles. Puede ser muy útil en un botón en muchas aplicaciones. Al igual que el control `Label` (Etiqueta), tampoco emite ninguna señal.

Sólo hay dos llamadas para manipular un control `Arrow` :

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

arrow.set(arrow_type, shadow_type)
```

La primera crea un control flecha con el tipo y apariencia indicados. La segunda permite cambiar cualquiera de estos valores. El argumento *`arrow_type`* puede tomar uno de lo siguientes valores:

```
ARROW_UP    #(Arriba)
ARROW_DOWN  #(Abajo)
ARROW_LEFT  #(Izquierda)
ARROW_RIGHT #(Derecha)
```

Estos valores obviamente indican la dirección hacia la que apunta la flecha. El argumento puede tomar uno de los siguientes valores:

```
SHADOW_IN
SHADOW_OUT   # valor predeterminado
SHADOW_ETCHED_IN
SHADOW_ETCHED_OUT
```

El programa de ejemplo arrow.py ilustra brevemente su uso. La figura Figura 9.2 muestra el resultado de ejecutar el programa:

Figura 9.2 Ejemplos de Botones con Flechas

El código fuente del programa arrow.py es:

```
1 #!/usr/bin/env python
2
3 # example arrow.py
```

```
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # Crea un control de Flecha con los parámetros especificados
10 # y lo empaqueta en un botón
11 def create_arrow_button(arrow_type, shadow_type):
12     button = gtk.Button();
13     arrow = gtk.Arrow(arrow_type, shadow_type);
14     button.add(arrow)
15     button.show()
16     arrow.show()
17     return button
18
19 class Arrows:
20     def __init__(self):
21         # Creamos una ventana nueva
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         window.set_title("Arrow Buttons")
25
26         # Es buena idea hacer esto con todas las ventanas
27         window.connect("destroy", lambda x: gtk.main_quit())
28
29         # Establecer el ancho del borde de ventana
30         window.set_border_width(10)
31
32         # Creamos una caja para poner las flechas/botones
33         box = gtk.HBox(gtk.FALSE, 0)
34         box.set_border_width(2)
35         window.add(box)
36
37         # Empaquetar y mostrar todos los controles
38         box.show()
39
40         button = create_arrow_button(gtk.ARROW_UP, gtk.SHADOW_IN)
41         box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
42
43         button = create_arrow_button(gtk.ARROW_DOWN, gtk.SHADOW_OUT)
44         box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
45
46         button = create_arrow_button(gtk.ARROW_LEFT, gtk.SHADOW_ETCHED_IN)
47         box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
48
49         button = create_arrow_button(gtk.ARROW_RIGHT, gtk.SHADOW_ETCHED_OUT ↩
   )
50         box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
51
52         window.show()
53
54 def main():
55     gtk.main()
56     return 0
57
58 if __name__ == "__main__":
59     Arrows()
60     main()
```

9.3. El Objeto Pistas (Tooltip)

Las `Tooltips` (Pistas) son pequeñas cadenas de texto que aparecen cuando se deja el cursor sobre un botón u otro control durante unos segundos.

Los controles que no reciben eventos (controles que no tienen su propia ventana) no funcionarán con las pistas.

La primera llamada que se usará crea una nueva pista. Sólo es necesario hacer esto una vez ya que el objeto que devuelve `gtk.Tooltips` puede usarse para crear múltiples pistas.

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

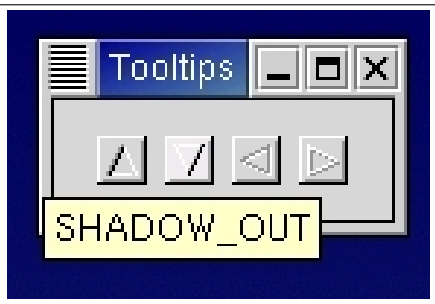
Una vez que se ha creado una nueva pista, y el control que se quiere que la use está preparado, simplemente se utiliza esta llamada para asociarlos:

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

El objeto *`tooltips`* es la pista que se acaba de crear. El primer argumento (*`widget`*) es el control que se quiere que muestre la pista; el segundo argumento (*`tip_text`*), el texto que se quiere visualizar. El último argumento (*`tip_private`*) es una cadena de texto que puede usarse como identificador.

El programa de ejemplo tooltip.py modifica el programa arrow.py para añadir una pista a cada botón. La figura Figura 9.3 ilustra la ventana resultante con la pista del segundo botón flecha activada:

Figura 9.3 Ejemplo de Pistas



El código fuente del programa tooltip.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo tooltip.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # Crear una Flecha con los parámetros especificados
10 # y empaquetarlo en un botón
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         # Creamos una ventana nueva
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23
24         window.set_title("Tooltips")
25
26         # It's a good idea to do this for all windows.
27         window.connect("destroy", lambda w: gtk.main_quit())
```

```
28
29          # Establece el grosor del borde de la ventana
30          window.set_border_width(10)
31
32          # Creamos una caja para poner las flechas/botones
33          box = gtk.HBox(gtk.FALSE, 0)
34          box.set_border_width(2)
35          window.add(box)
36
37          # creamos un objeto de pista
38          self.tooltips = gtk.Tooltips()
39
40          # Empaquetamos y mostramos todos los controles
41          box.show()
42
43          button = create_arrow_button(gtk.ARROW_UP, gtk.SHADOW_IN)
44          box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
45          self.tooltips.set_tip(button, "SHADOW_IN")
46
47          button = create_arrow_button(gtk.ARROW_DOWN, gtk.SHADOW_OUT)
48          box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
49          self.tooltips.set_tip(button, "SHADOW_OUT")
50
51          button = create_arrow_button(gtk.ARROW_LEFT, gtk.SHADOW_ETCHED_IN)
52          box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
53          self.tooltips.set_tip(button, "SHADOW_ETCHED_IN")
54
55          button = create_arrow_button(gtk.ARROW_RIGHT, gtk.SHADOW_ETCHED_OUT ↩
    )
56          box.pack_start(button, gtk.FALSE, gtk.FALSE, 3)
57          self.tooltips.set_tip(button, "SHADOW_ETCHED_OUT")
58
59          window.show()
60
61 def main():
62      gtk.main()
63      return 0
64
65 if __name__ == "__main__":
66      tt = Tooltips()
67      main()
```

Hay otros métodos que se pueden usar con las pistas. Simplemente los listaremos, junto con una breve descripción sobre su función.

```
tooltips.enable()
```

Activa un conjunto de pistas desactivadas.

```
tooltips.disable()
```

Desactiva un conjunto de pistas activadas.

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

Fija los milisegundos que deben transcurrir con el puntero sobre el control antes de que la pista aparezca. El valor predefinido es de 500 milisegundos (medio segundo).

Y esos son todos los métodos asociados con las pistas. Más de lo que siempre se querría saber :-)

9.4. Barras de Progreso (ProgressBar)

Las barras de progreso se usan para mostrar el estado de una operación. Son bastante fáciles de usar, como se verá en el código que sigue. Pero primero empecemos con una llamada para crear una nueva barra de progreso.

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

El argumento *adjustment* (ajuste) especifica un ajuste para usarlo con la barra de progreso *progressbar*. Si no se especifica se creará un ajuste. Ahora que la barra de progreso está creada ya podemos usarla.

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

El objeto *progressbar* es la barra de progreso con la que queremos operar, y el argumento (*fraction*) es la cantidad "completada", lo que significa la cantidad con la que se ha rellenado la barra de progreso desde 0 a 100%. Esto se le pasa al método como un número real entre 0 y 1.

Una barra de progreso puede orientarse de diversas formas usando el método:

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

El argumento *orientation* puede tomar uno de los siguientes valores para indicar la dirección en la que la barra de progreso se mueve:

```
PROGRESS_LEFT_TO_RIGHT  # izquierda a derecha
PROGRESS_RIGHT_TO_LEFT  # derecha a izquierda
PROGRESS_BOTTOM_TO_TOP  # abajo a arriba
PROGRESS_TOP_TO_BOTTOM  # arriba a abajo
```

Además de indicar la cantidad de progreso que se ha completado, la barra de progreso también puede usarse simplemente para indicar que ha habido alguna actividad. Esto puede ser útil en situaciones donde el progreso no se puede medir con un rango de valores. La siguiente función indica que se ha hecho algún progreso.

```
progressbar.pulse()
```

El tamaño de paso de un indicador de actividad se establece usando la siguiente función, donde la fracción es un número entre 0.0 y 1.0.

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

Cuando no está en el modo actividad, la barra de progreso también puede mostrar una cadena de texto en su canal, usando el siguiente método:

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

NOTA

Téngase en cuenta que set_text() no soporta el formateo de texto al estilo `printf()` como lo hacía la barra de progreso de GTK+ 1.2.

Se puede desactivar la visualización de la cadena llamando a set_text() de nuevo sin argumentos.

La cadena de texto actual de la barra de progreso se puede obtener con el siguiente método:

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

Normalmente las Barras de Progreso usan cronómetros u otras funciones parecidas (mira la sección sobre Cronómetros, E/S y Funciones de Inactividad) para dar la ilusión de multitarea. Todas usarán los métodos set_fraction() o pulse() de la misma forma.

El programa progressbar.py proporciona un ejemplo de barra de progreso, actualizada usando cronómetros. Este código también muestra como reiniciar la Barra de Progreso. La figura Figura 9.4 muestra la ventana resultante:

Figura 9.4 Ejemplo de Barra de Progreso

El código fuente del programa progressbar.py es:

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

# ejemplo progressbar.py

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

# Actualizar el valor de la barra de progreso de manera
# que tengamos algo de movimiento
def progress_timeout(pbobj):
    if pbobj.activity_check.get_active():
        pbobj.pbar.pulse()
    else:
        # Calcular el valor de la barra de progreso usando el
        # valor del rango establecido en el objeto ajuste
        new_val = pbobj.pbar.get_fraction() + 0.01
        if new_val > 1.0:
            new_val = 0.0
        # Fijar el nuevo valor
        pbobj.pbar.set_fraction(new_val)

    # Puesto que esta es una función de cronómetro, devolver TRUE de manera
    # que continúe siendo llamada
    return gtk.TRUE

class ProgressBar:
    # Retrollamada que conmuta el dibujado del texto en el
    # canal de la barra de progreso
    def toggle_show_text(self, widget, data=None):
        if widget.get_active():
            self.pbar.set_text("some text")
        else:
            self.pbar.set_text("")

    # Retrollamada que conmuta el modo de actividad de
    # la barra de progreso
    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      # Retrollamada que conmuta la orientación de la barra de progreso
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      # Limpiamos la memoria reservada y eliminamos el temporizador
52      def destroy_progress(self, widget, data=None):
53          gtk.timeout_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(gtk.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(gtk.FALSE, 5)
66          vbox.set_border_width(10)
67          self.window.add(vbox)
68          vbox.show()
69
70          # Creamos un objeto de alineación centrador
71          align = gtk.Alignment(0.5, 0.5, 0, 0)
72          vbox.pack_start(align, gtk.FALSE, gtk.FALSE, 5)
73          align.show()
74
75          # Creamos la barra de progreso
76          self.pbar = gtk.ProgressBar()
77
78          align.add(self.pbar)
79          self.pbar.show()
80
81          # Añadimos una retrollamada temporizada para actualizar el valor de ↩
     la barra
82          self.timer = gtk.timeout_add (100, progress_timeout, self)
83
84          separator = gtk.HSeparator()
85          vbox.pack_start(separator, gtk.FALSE, gtk.FALSE, 0)
86          separator.show()
87
88          # filas, columnas, homogéneas
89          table = gtk.Table(2, 2, gtk.FALSE)
90          vbox.pack_start(table, gtk.FALSE, gtk.TRUE, 0)
91          table.show()
92
93          # Añadir un botón de activación para mostrar o no el texto del  ↩
    canal
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         # Añadir un botón de activación para conmutar el modo de actividad
102         self.activity_check = check = gtk.CheckButton("Activity mode")
```

```
103         table.attach(check, 0, 1, 1, 2,
104                      gtk.EXPAND | gtk.FILL, gtk.EXPAND | gtk.FILL,
105                      5, 5)
106         check.connect("clicked", self.toggle_activity_mode)
107         check.show()
108
109         # Añadir un botón de activación para conmutar de orientación
110         check = gtk.CheckButton("Right to Left")
111         table.attach(check, 0, 1, 2, 3,
112                      gtk.EXPAND | gtk.FILL, gtk.EXPAND | gtk.FILL,
113                      5, 5)
114         check.connect("clicked", self.toggle_orientation)
115         check.show()
116
117         # Añadir un botón para salir del programa
118         button = gtk.Button("close")
119         button.connect("clicked", self.destroy_progress)
120         vbox.pack_start(button, gtk.FALSE, gtk.FALSE, 0)
121
122         # Esto lo hace el botón por defecto
123         button.set_flags(gtk.CAN_DEFAULT)
124
125         # Esto hace que el botón tome el foco por defecto. Pulsando ↩
    simplemente
126         # la tecla "Enter" se producirá la activación de este botón.
127         button.grab_default ()
128         button.show()
129
130         self.window.show()
131
132 def main():
133     gtk.main()
134     return 0
135
136 if __name__ == "__main__":
137     ProgressBar()
138     main()
```

9.5. Diálogos

El control `Dialog` (Diálogo) es muy simple, y realmente es sólo una ventana con unas cuantas cosas ya empaquetadas. Simplemente crea una ventana, y luego empaqueta una `VBox` en ella, que contiene un separador y luego una `HBox` llamada "action_area" ("área de acción").

El control `Dialog` (Diálogo) se puede usar para mensajes emergentes para el usuario, y otras tareas similares. Es realmente básico, y sólo hay una función para la caja de diálogo, que es:

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

donde *`title`* (título) es el texto usado en la barra de título, *`parent`* (padre) es la ventana principal de la aplicación y *`flags`* establece varios modos de operación para el diálogo:

```
DIALOG_MODAL - hace el diálogo modal
DIALOG_DESTROY_WITH_PARENT - destruye el diálogo cuando su padre sea destruido
DIALOG_NO_SEPARATOR - omite el separador entre la vbox y el área de acción
```

El argumento *`buttons`* (botones) es una tupla de pares texto de botón y respuesta. Todos los argumentos tienen valores predeterminados y pueden especificarse usando palabras clave.

Esto creará la caja de diálogo, y ahora depende del desarrollador el usarla. Se podría empaquetar un botón en el área de acción:

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

Y se podría añadir, por ejemplo, una etiqueta, al área *vbox* usando el empaquetamiento, con algo así:

```
label = gtk.Label("Los diálogos molan")
dialog.vbox.pack_start(label, TRUE, TRUE, 0)
label.show()
```

Como ejemplo del uso de una caja de diálogo, se podrían poner dos botones en el área de acción, un botón de Cancelar, un botón de Aceptar y una etiqueta en el área *vbox*, haciendo una pregunta al usuario, informando de un error, etc. Luego se podrían conectar diferentes señales a cada botón y realizar la operación que el usuario seleccione.

Si la funcionalidad básica que proporcionan las cajas verticales y horizontales predeterminadas no dan el suficiente control para la aplicación, entonces se puede sencillamente empaquetar otro control dentro de las cajas proporcionadas. Por ejemplo, se podría empaquetar una tabla en la caja vertical.

9.6. Imágenes

Las `Images` (Imágenes) son estructuras de datos que contienen dibujos. Estos dibujos se pueden usar en varios sitios.

Las `Images` (Imágenes) se pueden crear a partir de `Pixbufs`, `Pixmaps`, archivos que contengan información de imagen (por ejemplo. XPM, PNG, JPEG, TIFF, etc.), e incluso ficheros de animación.

Las `Images` (Imágenes) se crean usando la función:

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

Después se carga la imagen usando alguno de los siguientes métodos:

```
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)
```

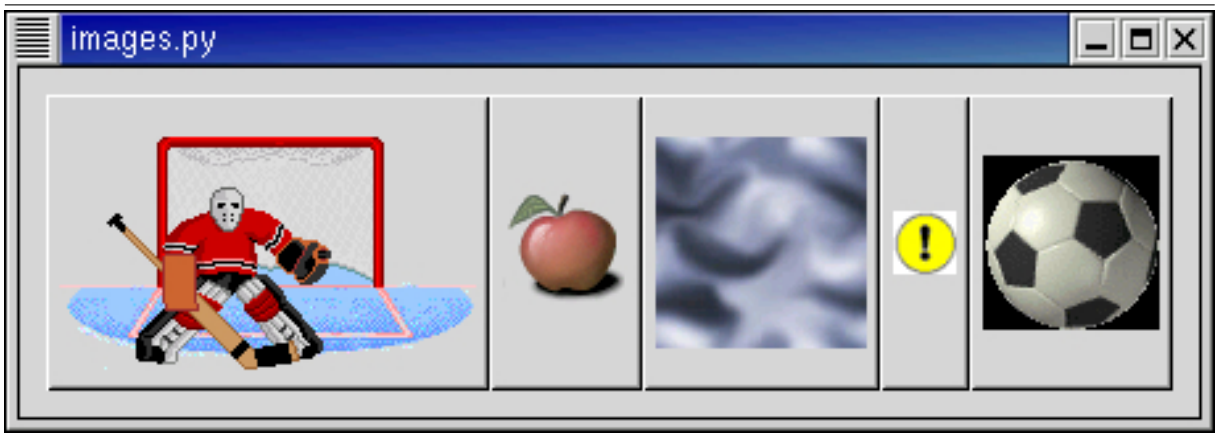
Donde *pixbuf* es un `GdkPixbuf`; *pixmap* y *mask* son `gtk.gdk.Pixmaps`; *image* es una `gtk.gdk.Image`; *stock_id* es el nombre de un `gtk.StockItem`; *icon_set* es un `gtk.IconSet`; y, *animation* es una `gtk.gdk.PixbufAnimation`. el argumento *size* (tamaño) es uno de:

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

La forma más fácil de crear una imagen es usar el método set_from_file() que automáticamente determina el tipo de imagen y la carga.

El programa images.py muestra cómo cargar varios tipos de imagen (goalie.gif, apple-red.png, chaos.jpg, important.tif, soccerball.gif) en imágenes que se colocan dentro de botones:

Figura 9.5 Ejemplo de Imágenes en Botones



El código fuente es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo images.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ImagesExample:
10     # cuando se invoca (con la señal delete_event), finaliza la aplicación
11     def close_application(self, widget, event, data=None):
12         gtk.main_quit()
13         return gtk.FALSE
14
15     # se invoca cuando el botón es pulsado.  Simplemente imprime un mensaje ↩
   .
16     def button_clicked(self, widget, data=None):
17         print "button %s clicked" % data
18
19     def __init__(self):
20         # crea la ventana principal y conecta la señal delete_event signal  ↩
   para finalizar
21         # la aplicación
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23         window.connect("delete_event", self.close_application)
24         window.set_border_width(10)
25         window.show()
26
27         # una caja horizontal que contenga los botones
28         hbox = gtk.HBox()
29         hbox.show()
30         window.add(hbox)
31
32         pixbufanim = gtk.gdk.PixbufAnimation("goalie.gif")
33         image = gtk.Image()
34         image.set_from_animation(pixbufanim)
35         image.show()
36         # un botón que contenga el control de imagen
37         button = gtk.Button()
38         button.add(image)
39         button.show()
40         hbox.pack_start(button)
41         button.connect("clicked", self.button_clicked, "1")
42
43         # crear varias imágenes con datos de archivos y cargarlos
```

```
44          # en botones
45          image = gtk.Image()
46          image.set_from_file("apple-red.png")
47          image.show()
48          # un botón que cotenga el control de imagen
49          button = gtk.Button()
50          button.add(image)
51          button.show()
52          hbox.pack_start(button)
53          button.connect("clicked", self.button_clicked, "2")
54
55          image = gtk.Image()
56          image.set_from_file("chaos.jpg")
57          image.show()
58          # un botón que cotenga el control de imagen
59          button = gtk.Button()
60          button.add(image)
61          button.show()
62          hbox.pack_start(button)
63          button.connect("clicked", self.button_clicked, "3")
64
65          image = gtk.Image()
66          image.set_from_file("important.tif")
67          image.show()
68          # un botón que cotenga el control de imagen
69          button = gtk.Button()
70          button.add(image)
71          button.show()
72          hbox.pack_start(button)
73          button.connect("clicked", self.button_clicked, "4")
74
75          image = gtk.Image()
76          image.set_from_file("soccerball.gif")
77          image.show()
78          # un botón que cotenga el control de imagen
79          button = gtk.Button()
80          button.add(image)
81          button.show()
82          hbox.pack_start(button)
83          button.connect("clicked", self.button_clicked, "5")
84
85
86  def main():
87      gtk.main()
88      return 0
89
90  if __name__ == "__main__":
91      ImagesExample()
92      main()
```

9.6.1. Pixmaps

Los `Pixmaps` son estructuras de datos que contienen dibujos. Estos dibujos se pueden usar en varios sitios, pero lo más común es usarlos como iconos en un escritorio X, o como cursores.

Un pixmap con sólo 2 colores se llama bitmap, y hay unas cuantas rutinas adicionales para trabajar con este caso especial.

Para entender los pixmaps, es de ayuda entender cómo funciona el sistema X Window. En X, las aplicaciones no necesitan ejecutarse en el mismo ordenador que interactua con el usuario. En cambio, estas aplicaciones, llamadas "clientes", se comunican con un programa que muestra los gráficos y maneja el teclado y el ratón. Este programa que interactua directamente con el usuario se llama un "servidor de visualización" o "servidor X". Ya que la comunicación puede tener lugar sobre una red, es importante mantener alguna información en el servidor X. Los `Pixmaps`, por ejemplo, se almacenan en la memoria

del servidor X. Esto significa que, una vez que los valores de un pixmap se establecen, no hay que seguir transmitiendolos por la red; en lugar de eso, se envía un comando para "mostrar el pixmap número XYZ aqui." Incluso si no se está usando X con GTK simultáneamente, usando construcciones como `Pixmaps` hará que los programas funcionen de forma aceptable en X.

Para usar pixmaps en PyGTK, primero debemos construir un `gtk.gdk.Pixmap` usando las funciones de gtk.gdk en PyGTK. Los `Pixmaps` se pueden crear a partir de datos en memoria, o a partir de datos leídos desde un fichero. Veamos cada una de las llamadas usadas para crear un pixmap.

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

Esta rutina se usa para crear un *pixmap* con la profundidad de color dada por el argumento *depth* a partir de los datos *data* en memoria. Si *depth* es -1 su valor se deduce de la de la ventana *window*. Cada pixel usa un número de bits de datos para representar el color que es igual a la profundidad de color. El *width*(ancho) y el *height* (alto) son en pixeles. El argumento *window* (ventana) debe referirse a una `gtk.gdk.Window` realizada, ya que los recursos de un pixmap sólo tienen sentido en el contexto de la pantalla donde se va a visualizar. *fg* y *bg* son los colores de frente y fondo del pixmap.

Se pueden crear pixmaps desde ficheros XPM usando:

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

El formato XPM es una representación legible de pixmap para el Sistema de Ventanas X. Es usado ampliamente y hay muchas utilidades disponibles para crear ficheros de imágenes en este formato. En la función `pixmap_create_from_xpm`() el primer argumento es del tipo `gtk.gdk.Window` . (La mayoría de los controles GTK tienen una `gtk.gdk.Window` subyacente que se puede obtener usando el atributo window (ventana) del control.) El fichero se especifica con *filename* y debe contener una imagen en formato XPM el cual se carga en la estructura del *pixmap*. La *mask* (máscara) es un bitmap que especifica qué bits del *pixmap* son opacos; se crea con la función. Todos los demás pixeles se colorean con el color especificado por *transparent_color*. Un ejemplo del uso de esta función se recoge a continuación.

Los Pixmaps también puede crearse a partir de datos en memoria usando la función:

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

Imagenes pequeñas se pueden incorporar a un programa como datos en el formato XPM usando la función anterior. Un pixmap se crea usando estos datos, en lugar de leerlo de un fichero. Un ejemplo de este tipo de datos es:

```
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.    ",
"   .........    ",
"                ",
"                "
]
```

La última forma para crear un pixmap en blanco disponible para operaciones de dibujo es:

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

window es o una `gtk.gdk.Window` o `None`. Si *window* es una `gtk.gdk.Window` entonces *depth* puede ser -1 para indicar que la profundidad de color se obtiene de la ventana. Si *window* es `None` entonces *depth* debe especificarse.

El programa pixmap.py es un ejemplo del uso de un pixmap en un botón. La figura Figura 9.6 muestra el resultado:

Figura 9.6 Ejemplo de Pixmap en un Botón

El código fuente es:

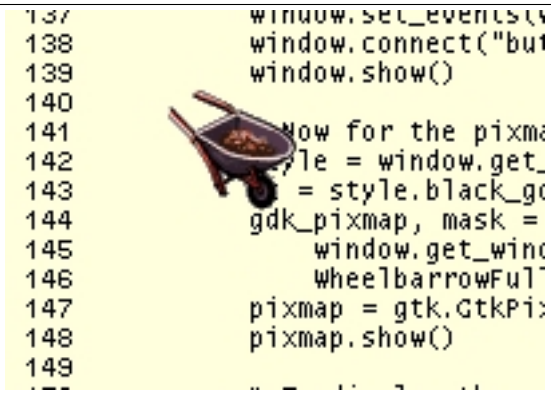
```
 1 #!/usr/bin/env python
 2
 3 # ejemplo pixmap.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # XPM data of Open-File icon
10 xpm_data = [
11 "16 16 3 1",
12 "       c None",
13 ".      c #000000000000",
14 "X      c #FFFFFFFFFFFF",
15 "                ",
16 "   ......       ",
17 "   .XXX.X.      ",
18 "   .XXX.XX.     ",
19 "   .XXX.XXX.    ",
20 "   .XXX.....    ",
21 "   .XXXXXXX.    ",
22 "   .XXXXXXX.    ",
23 "   .XXXXXXX.    ",
24 "   .XXXXXXX.    ",
25 "   .XXXXXXX.    ",
26 "   .XXXXXXX.    ",
27 "   .XXXXXXX.    ",
28 "   .........    ",
29 "                ",
30 "                "
31 ]
32
33 class PixmapExample:
34     # cuando se invoca (con la señal delete_event), finaliza la aplicación.
35     def close_application(self, widget, event, data=None):
36         gtk.main_quit()
37         return gtk.FALSE
38
39     # se invoca al pulsar el botón.  Simplemente imprime un mensaje
40     def button_clicked(self, widget, data=None):
41         print "button clicked"
42
43     def __init__(self):
```

```
44          # crea la ventana principal y conecta la señal delete_event para ↩
    finalizar
45          # la aplicación
46          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
47          window.connect("delete_event", self.close_application)
48          window.set_border_width(10)
49          window.show()
50
51          # ahora el pixmap desde datos XPM
52          pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(window.window,
53                                                          None,
54                                                          xpm_data)
55
56          # un control de imagen que contenga el pixmap
57          image = gtk.Image()
58          image.set_from_pixmap(pixmap, mask)
59          image.show()
60
61          # un botón que contenga el control de imagen
62          button = gtk.Button()
63          button.add(image)
64          window.add(button)
65          button.show()
66
67          button.connect("clicked", self.button_clicked)
68
69  def main():
70      gtk.main()
71      return 0
72
73  if __name__ == "__main__":
74      PixmapExample()
75      main()
```

Una desventaja de usar pixmaps es que que el objeto mostrado siempre es rectangular, independientemente de la imagen. Nos gustaría crear escritorios y aplicaciones con iconos que tengan formas más naturales. Por ejemplo, para la interfaz de un juego, nos gustaría tener botones redondos para pulsar. La forma de hacer esto es usar ventanas con forma.

Una ventana con forma es simplemente un pixmap en el que los pixeles de fondo son transparentes. De esta forma, cuando la imagen de fondo se colorea, no la sobreescribimos con un borde rectangular y que no encaja, de nuestro icono. El programa de ejemplo wheelbarrow.p muestra una imagen completa en el escritorio. La figura Figura 9.7 muestra la imagen sobre una ventana de terminal:

Figura 9.7 Ejemplo de Ventana con Forma



The source code for wheelbarrow.py is:

```
1 #!/usr/bin/env python
2
3 # ejemplo wheelbarrow.py
```

```
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 # XPM
10 WheelbarrowFull_xpm = [
11 "48 48 64 1",
12 "       c None",
13 ".      c #DF7DCF3CC71B",
14 "X      c #965875D669A6",
15 "o      c #71C671C671C6",
16 "O      c #A699A289A699",
17 "+      c #965892489658",
18 "@      c #8E38410330C2",
19 "#      c #D75C7DF769A6",
20 "$      c #F7DECF3CC71B",
21 "%      c #96588A288E38",
22 "&      c #A69992489E79",
23 "*      c #8E3886178E38",
24 "=      c #104008200820",
25 "-      c #596510401040",
26 ";      c #C71B30C230C2",
27 ":      c #C71B9A699658",
28 ">      c #618561856185",
29 ",      c #20811C712081",
30 "<      c #104000000000",
31 "1      c #861720812081",
32 "2      c #DF7D4D344103",
33 "3      c #79E769A671C6",
34 "4      c #861782078617",
35 "5      c #41033CF34103",
36 "6      c #000000000000",
37 "7      c #49241C711040",
38 "8      c #492445144924",
39 "9      c #082008200820",
40 "0      c #69A618611861",
41 "q      c #B6DA71C65144",
42 "w      c #410330C238E3",
43 "e      c #CF3CBAEAB6DA",
44 "r      c #71C6451430C2",
45 "t      c #EFBEDB6CD75C",
46 "y      c #28A208200820",
47 "u      c #186110401040",
48 "i      c #596528A21861",
49 "p      c #71C661855965",
50 "a      c #A69996589658",
51 "s      c #30C228A230C2",
52 "d      c #BEFBA289AEBA",
53 "f      c #596545145144",
54 "g      c #30C230C230C2",
55 "h      c #8E3882078617",
56 "j      c #208118612081",
57 "k      c #38E30C300820",
58 "l      c #30C2208128A2",
59 "z      c #38E328A238E3",
60 "x      c #514438E34924",
61 "c      c #618555555965",
62 "v      c #30C2208130C2",
63 "b      c #38E328A230C2",
64 "n      c #28A228A228A2",
65 "m      c #41032CB228A2",
66 "M      c #104010401040",
67 "N      c #492438E34103",
```

```
 68 "B      c #28A2208128A2",
 69 "V      c #A699596538E3",
 70 "C      c #30C21C711040",
 71 "Z      c #30C218611040",
 72 "A      c #965865955965",
 73 "S      c #618534D32081",
 74 "D      c #38E31C711040",
 75 "F      c #082000000820",
 76 "                                          ",
 77 "          .XoO                            ",
 78 "         +@#$%o&                          ",
 79 "         *=-;#::o+                        ",
 80 "           >,<12#:34                      ",
 81 "             45671#:X3                    ",
 82 "               +89<02qwo                  ",
 83 "e*                >,67;ro                 ",
 84 "ty>                 459@>+&&               ",
 85 "$2u+                  ><ipas8*             ",
 86 "%$;=*                *3:.Xa.dfg>           ",
 87 "Oh$;ya              *3d.a8j,Xe.d3g8+        ",
 88 " Oh$;ka           *3d$a8lz,,xxc:.e3g54      ",
 89 "  Oh$;kO        *pd$%svbzz,sxxxxfX..&wn>    ",
 90 "   Oh$@mO    *3dthwlsslszjzxxxxxxx3:td8M4   ",
 91 "    Oh$@g& *3d$XNlvvvlllm,mNwxxxxxxxfa.:,B* ",
 92 "     Oh$@,Od.czllllzlmmqV@V#V@fxxxxxxxf:%j5& ",
 93 "      Oh$1hd5lllslllCCZrV#r#:#2AxxxxxxxxxcdwM* ",
 94 "       OXq6c.%8vvvllZZiqqApA:mq:Xxcpcxxxxxfdc9* ",
 95 "        2r<6gde3bllZZrVi7S@SV77A::qApxxxxxxfdcM ",
 96 "        :,q-6MN.dfmZZrrSS:#riirDSAX@Af5xxxxxfevo",
 97 "         +A26jguXtAZZZC7iDiCCrVVii7Cmmmxxxxxx%3g",
 98 "          *#16jszN..3DZZZZrCVSA2rZrV7Dmmwxxxx&en",
 99 "           p2yFvzssXe:fCZZCiiD7iiZDiDSSZwwxx8e*>",
100 "           OA1<jzxwwc:$d%NDZZZZCCCZCCZZCmxxfd.B ",
101 "            3206Bwxxszx%et.eaAp77m77mmmf3&eeeg* ",
102 "             @26MvzxNzvlbwfpdettttttttttt.c,n&  ",
103 "             *;16=lsNwwNwgsvslbwwvccc3pcfu<o    ",
104 "              p;<69BvwwsszslllbBlllllllu<5+     ",
105 "              OS0y6FBlvvvzvzss,u=Blllj=54       ",
106 "               c1-699Blvllllllu7k96MMMg4        ",
107 "               *10y8n6FjvllllB<166668           ",
108 "                S-kg+>666<M<996-y6n<8*          ",
109 "                p71=4 m69996kD8Z-66698&&        ",
110 "                &i0ycm6n4 ogk17,0<6666g         ",
111 "                 N-k-<>     >=01-kuu666>        ",
112 "                 ,6ky&      &46-10ul,66,        ",
113 "                 Ou0<>       o66y<ulw<66&       ",
114 "                  *kk5       >66By7=xu664       ",
115 "                   <<M4      466lj<Mxu66o       ",
116 "                   *>>       +66uv,zN666*       ",
117 "                              566,xxj669        ",
118 "                              4666FF666>        ",
119 "                               >966666M         ",
120 "                                oM6668+         ",
121 "                                  *4            ",
122 "                                                ",
123 "                                                "
124 ]
125
126 class WheelbarrowExample:
127     # Cuando se invoca (con la señal delete_event), finaliza la aplicación
128     def close_application(self, widget, event, data=None):
129         gtk.main_quit()
130         return gtk.FALSE
131
```

```
132     def __init__(self):
133         # Crea la ventana principal y conecta la señal delete_event para finalizar
134         # la aplicación.  Obsérvese que la ventana principal no tendrá título
135         # ya que vamos a hacer una ventana emergente (popup).
136         window = gtk.Window(gtk.WINDOW_POPUP)
137         window.connect("delete_event", self.close_application)
138         window.set_events(window.get_events() | gtk.gdk.BUTTON_PRESS_MASK)
139         window.connect("button_press_event", self.close_application)
140         window.show()
141
142         # ahora para el pixmap y el control de imagen
143         pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(
144             window.window, None, WheelbarrowFull_xpm)
145         image = gtk.Image()
146         image.set_from_pixmap(pixmap, mask)
147         image.show()
148
149         # Para mostrar la imagen usamos un control fijo para situarla
150         fixed = gtk.Fixed()
151         fixed.set_size_request(200, 200)
152         fixed.put(image, 0, 0)
153         window.add(fixed)
154         fixed.show()
155
156         # Esto enmascara todo salvo la imagen misma
157         window.shape_combine_mask(mask, 0, 0)
158
159         # mostramos la ventana
160         window.set_position(gtk.WIN_POS_CENTER_ALWAYS)
161         window.show()
162
163 def main():
164     gtk.main()
165     return 0
166
167 if __name__ == "__main__":
168     WheelbarrowExample()
169     main()
```

Para hacer la imagen sensible, conectamos la señal "button_press_event" para que el programa finalice. Las lineas 138-139 hacen el dibujo sensible a una pulsación de un botón del ratón y lo conectan al método close_application() .

9.7. Reglas

Los controles `Ruler` (Regla) se usan para indicar la posición del puntero del ratón en una ventana determinada. Una ventana puede tener una regla vertical a lo largo del ancho y una regla horizontal a lo largo del alto. Un pequeño triángulo indicador en la regla muestra la posición exacta del puntero respecto a la regla.

Antes de nada es necesario crear una regla. Las reglas horizontales y verticales se crean usando las siguientes funciones:

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

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

Una vez que se crea una regla podemos definir la unidad de medida. Las unidades de medida para las reglas pueden ser `PIXELS` (píxeles), `INCHES` (pulgadas) o `CENTIMETERS` (centímetros). Esto se fija con el método:

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

La medida predeterminada es `PIXELS`.

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

Otra característica importante de una regla es cómo marca las unidades de escala y donde se coloca el indicador de posición inicialmente. Esto se fija usando el método:

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

Los argumentos *`lower`* (bajo) y *`upper`* (alto) definen la extensión de la regla, y *`max_size`* (tamaño máximo) es el mayor número posible que se visualizará. La *`Position`* (Posición) define la posición inicial del indicador del puntero dentro de la regla.

Una regla vertical puede medir una ventana de 800 píxeles de ancho así:

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

Las marcas mostradas en la regla irán desde 0 a 800, con un número cada 100 píxeles. Si en lugar de eso quisieramos una regla de 7 a 16, escribiríamos:

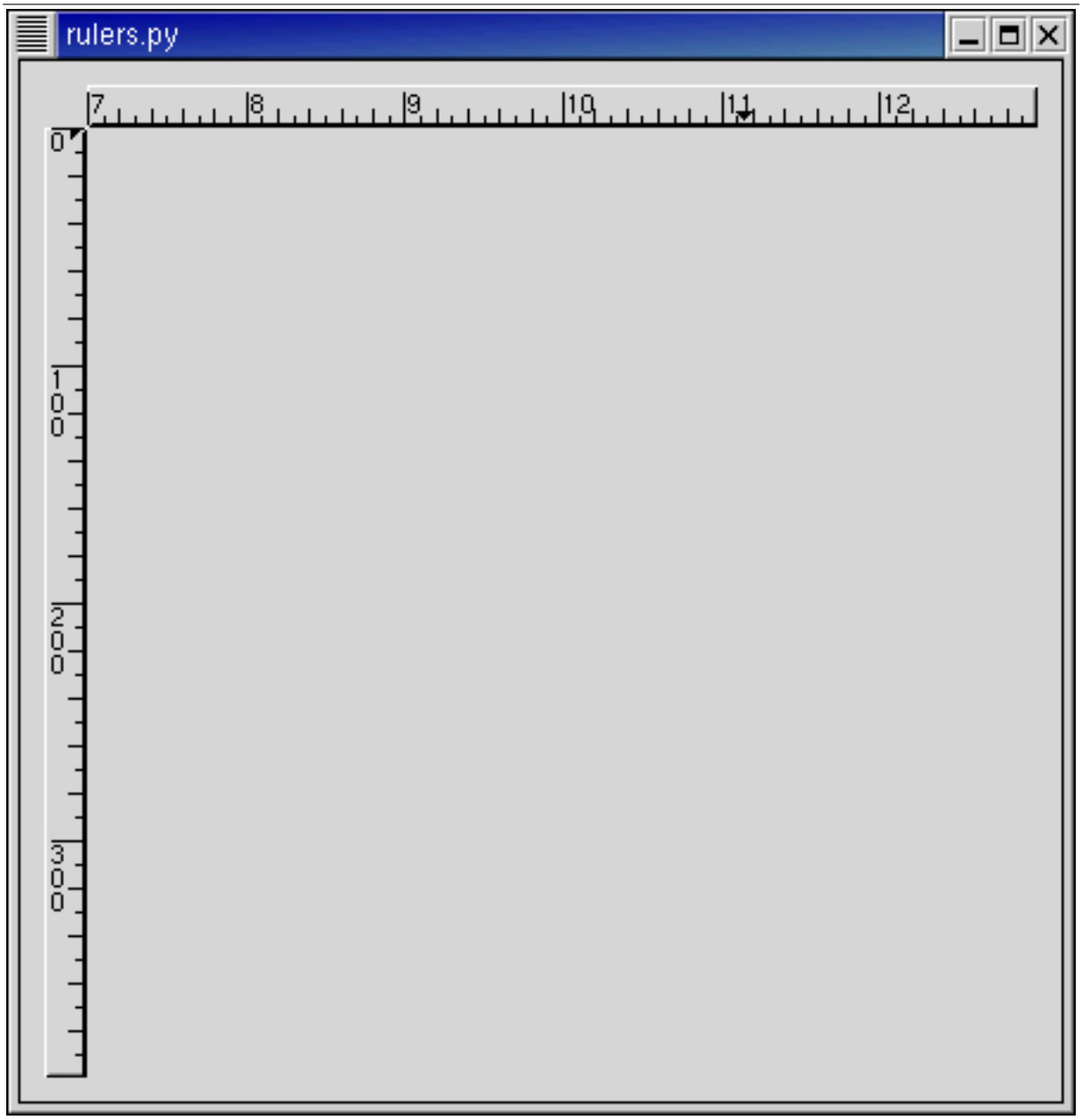
```
vruler.set_range(7, 16, 0, 20)
```

El indicador de la regla es una pequeña marca triangular que indica la posición del puntero relativa a la regla. Si la regla se usa para seguir el puntero del ratón, la señal "motion_notify_event" debe conectarse al método "motion_notify_event" de la regla. Hay que configurar una retrollamada para "motion_notify_event" para el área y usar connect_object() para que la regla emita una señal "motion_notify_signal":

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

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

El programa de ejemplo rulers.py crea un área de dibujo con una regla horizontal en la parte de arriba y una regla vertical a su izquierda. El tamaño del área de dibujo es de 600 píxeles de ancho por 400 píxeles de alto. La regla horizontal va desde 7 hasta 13 con una marca cada 100 píxeles, mientras que la regla vertical va de 0 a 400 con una marca cada 100 píxeles. La colocación del área de dibujo y las reglas se hace con una tabla. La figura Figura 9.8 ilustra el resultado:

Figura 9.8 Ejemplo de Reglas



El código fuente es rulers.py:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo rulers.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class RulersExample:
10     XSIZE = 400
11     YSIZE = 400
12
13     # Esta rutina toma el control cuando se pulsa el botón de cerrar
14     def close_application(self, widget, event, data=None):
15         gtk.main_quit()
16         return gtk.FALSE
17
```

```
18      def __init__(self):
19          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
20          window.connect("delete_event", self.close_application)
21          window.set_border_width(10)
22
23          # Crea una tabla para colocar la regla y el área de dibujo
24          table = gtk.Table(3, 2, gtk.FALSE)
25          window.add(table)
26
27          area = gtk.DrawingArea()
28          area.set_size_request(self.XSIZE, self.YSIZE)
29          table.attach(area, 1, 2, 1, 2,
30                       gtk.EXPAND|gtk.FILL, gtk.FILL, 0, 0 )
31          area.set_events(gtk.gdk.POINTER_MOTION_MASK |
32                          gtk.gdk.POINTER_MOTION_HINT_MASK )
33
34          # La regla horizontal está arriba. Cuando el ratón se mueve por el
35          # área de dibujo se pasa un evento motion_notify_event al manejador
36          # adecuado para la regla
37          hrule = gtk.HRuler()
38          hrule.set_metric(gtk.PIXELS)
39          hrule.set_range(7, 13, 0, 20)
40          def motion_notify(ruler, event):
41              return ruler.emit("motion_notify_event", event)
42          area.connect_object("motion_notify_event", motion_notify, hrule)
43          table.attach(hrule, 1, 2, 0, 1,
44                       gtk.EXPAND|gtk.SHRINK|gtk.FILL, gtk.FILL, 0, 0 )
45
46          # La regla vertical está a la izquierda. Cuando el ratón se mueve ↩
    por el
47          # área de dibujo se pasa un evento motion_notify_event al manejador
48          # adecuado para la regla
49          vrule = gtk.VRuler()
50          vrule.set_metric(gtk.PIXELS)
51          vrule.set_range(0, self.YSIZE, 10, self.YSIZE)
52          area.connect_object("motion_notify_event", motion_notify, vrule)
53          table.attach(vrule, 0, 1, 1, 2,
54                       gtk.FILL, gtk.EXPAND|gtk.SHRINK|gtk.FILL, 0, 0 )
55
56          # Ahora mostramos todo
57          area.show()
58          hrule.show()
59          vrule.show()
60          table.show()
61          window.show()
62
63  def main():
64      gtk.main()
65      return 0
66
67  if __name__ == "__main__":
68      RulersExample()
69      main()
```

Las lineas 42 y 52 conectan la retrollamada motion_notify() al área pasandole *hrule* en la linea 42 y *vrule* en la línea 52 como datos de usuario. La retrollamada motion_notify() se llamará dos veces cada vez que el ratón se mueva - una vez con *hrule* y otra vez con *vrule*.

9.8. Barras de Estado

Las `Statusbar` (Barras de Estado) son unos controles simples que se usan para visualizar un mensaje de texto. Mantienen una pila de los mensajes que se les han enviado, para que al quitar el mensaje actual se visualice el mensaje anterior.

Para que distintas partes de la aplicación puedan usar la misma barra de estado para visualizar mensajes, el control de barra de estado mantiene Identificadores de Contexto que se usan para identificar diferentes "usuarios". El mensaje en el tope de la pila es el que se visualiza, no importa el contexto al que pertenezca. Los mensajes se apilan en orden último en llegar primero en salir, no en orden de identificadores de contexto.

Una barra de estado se crea con una llamada a:

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

Se puede solicitar un nuevo Identificador de Contexto usando una llamada al siguiente método con una pequeña descripción textual del contexto:

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

Hay tres métodos adicionales para utilizar las barras de estado:

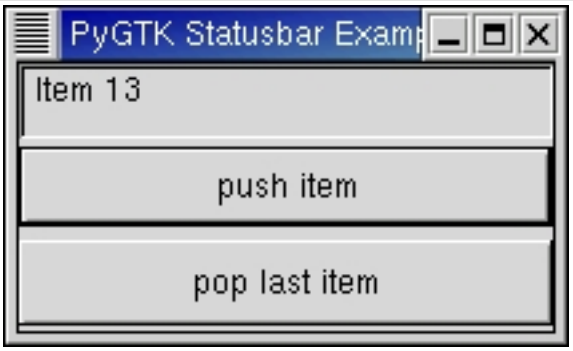
```
message_id = statusbar.push(context_id, text)

statusbar.pop(context_id)

statusbar.remove(context_id, message_id)
```

El primero, push(), se usa para añadir un nuevo mensaje a la *statusbar* (barra de estado). Devuelve un *message_id* (identificador de mensaje), que puede usarse con el método remove() para borrar el mensaje que cumpla la combinación de *message_id* y *context_id* en la pila de la *statusbar* (barra de estado).

El método pop() elimina el mensaje que esté en la posición más alta de la pila con el identificador de contexto *context_id*.

El programa de ejemplo statusbar.py crea una barra de estado y dos botones, uno para insertar elementos en la barra de estado, y otro para sacar el último elemento fuera. La figura Figura 9.9 muestra el resultado:

Figura 9.9 Ejemplo de Barra de Estado



El código fuente es:

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

```
17          self.status_bar.pop(data)
18          return
19
20      def __init__(self):
21          self.count = 1
22          # crea una ventana nueva
23          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
24          window.set_size_request(200, 100)
25          window.set_title("PyGTK Statusbar Example")
26          window.connect("delete_event", lambda w,e: gtk.main_quit())
27
28          vbox = gtk.VBox(gtk.FALSE, 1)
29          window.add(vbox)
30          vbox.show()
31
32          self.status_bar = gtk.Statusbar()
33          vbox.pack_start(self.status_bar, gtk.TRUE, gtk.TRUE, 0)
34          self.status_bar.show()
35
36          context_id = self.status_bar.get_context_id("Statusbar example")
37
38          button = gtk.Button("push item")
39          button.connect("clicked", self.push_item, context_id)
40          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
41          button.show()
42
43          button = gtk.Button("pop last item")
44          button.connect("clicked", self.pop_item, context_id)
45          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 2)
46          button.show()
47
48          # siempre mostramos la ventana al final para que se muestre
49          # de una vez en pantalla.
50          window.show()
51
52  def main():
53      gtk.main()
54      return 0
55
56  if __name__ == "__main__":
57      StatusbarExample()
58      main()
```

9.9. Entradas de Texto (Entry)

El control `Entry` (Entrada) permite escribir texto y mostrarlo en una caja de texto con una única línea. El texto puede fijarse con llamadas a métodos que permiten que nuevo texto reemplace, se inserte antes o se añada después del contenido actual del control `Entry` .

La función para crear un control `Entry` es:

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

Si el argumento *`max`* se especifica se establece un límite de longitud del texto dentro de la `Entry`. Si *`max`* es 0 no hay límite.

La longitud máxima de una entrada puede cambiarse usando el método:

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

El siguiente método altera el texto que hay actualmente en el control `Entry` .

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

El método set_text() fija el contenido del control `Entry` a *`text`*, remplazando el contenido actual. Obsérvese que la clase `Entry` implementa la interfaz `Editable` (sí, `gobject` soporta interfaces al estilo de Java) que contiene algunas funciones más para manipular los contenidos. Por ejemplo, el método:

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

inserta *`text`* en la posición indicada dentro de la *`entry`*.

El contenido de la `Entry` puede recuperarse usando una llamada al siguiente método. Esto es útil en las retrollamadas que se describen más abajo.

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

Si no queremos que el contenido de la `Entry` sea modificada por alguien escribiendo en ella, podemos cambiar su estado de edición.

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

El método anterior permite intercambiar el estado de edición del control `Entry` pasándole un valor `TRUE` o `FALSE` en el argumento *`is_editable`*.

Si estamos usando el control `Entry` y no queremos que el texto que se introduzca sea visible, por ejemplo cuando una contraseña se escribe, podemos usar el siguiente método, que además acepta una bandera booleana.

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

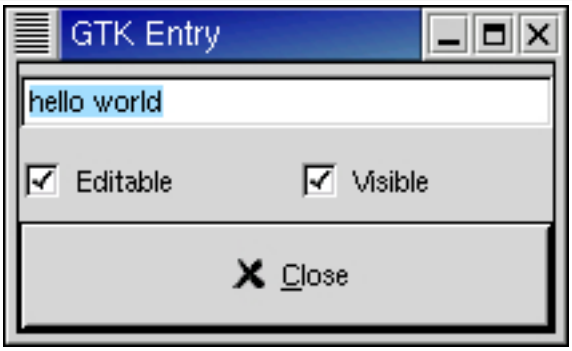
Una región del texto se puede poner como seleccionada usando el siguiente método. Esto se usaría sobre todo cuando se ponga algún valor predeterminado en un `Entry`, haciendo fácil para el usuario el borrado.

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

Si queremos recibir notificación cuando el usuario introduzca el texto, podemos conectar a las señales "activate" o "changed". La primera se produce cuando el usuario pulsa la tecla enter dentro del control `Entry`. La segunda se produce cuando ocurre cualquier cambio en el texto, por ejemplo, para cualquier inserción o borrado de un caracter.

El programa de ejemplo entry.py muestra el uso de un control `Entry` . La figura Figura 9.10 muestra el resultado de ejecutar el programa:

Figura 9.10 Ejemplo de Entrada



El código fuente entry.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo entry.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class EntryExample:
10     def enter_callback(self, widget, entry):
```

```
11          entry_text = entry.get_text()
12          print "Entry contents: %s\n" % entry_text
13
14      def entry_toggle_editable(self, checkbutton, entry):
15          entry.set_editable(checkbutton.get_active())
16
17      def entry_toggle_visibility(self, checkbutton, entry):
18          entry.set_visibility(checkbutton.get_active())
19
20      def __init__(self):
21          # create a new window
22          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23          window.set_size_request(200, 100)
24          window.set_title("GTK Entry")
25          window.connect("delete_event", lambda w,e: gtk.main_quit())
26
27          vbox = gtk.VBox(gtk.FALSE, 0)
28          window.add(vbox)
29          vbox.show()
30
31          entry = gtk.Entry()
32          entry.set_max_length(50)
33          entry.connect("activate", self.enter_callback, entry)
34          entry.set_text("hello")
35          entry.insert_text(" world", len(entry.get_text()))
36          entry.select_region(0, len(entry.get_text()))
37          vbox.pack_start(entry, gtk.TRUE, gtk.TRUE, 0)
38          entry.show()
39
40          hbox = gtk.HBox(gtk.FALSE, 0)
41          vbox.add(hbox)
42          hbox.show()
43
44          check = gtk.CheckButton("Editable")
45          hbox.pack_start(check, gtk.TRUE, gtk.TRUE, 0)
46          check.connect("toggled", self.entry_toggle_editable, entry)
47          check.set_active(gtk.TRUE)
48          check.show()
49
50          check = gtk.CheckButton("Visible")
51          hbox.pack_start(check, gtk.TRUE, gtk.TRUE, 0)
52          check.connect("toggled", self.entry_toggle_visibility, entry)
53          check.set_active(gtk.TRUE)
54          check.show()
55
56          button = gtk.Button(stock=gtk.STOCK_CLOSE)
57          button.connect("clicked", lambda w: gtk.main_quit())
58          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
59          button.set_flags(gtk.CAN_DEFAULT)
60          button.grab_default()
61          button.show()
62          window.show()
63
64 def main():
65      gtk.main()
66      return 0
67
68 if __name__ == "__main__":
69      EntryExample()
70      main()
```

9.10. Botones Aumentar/Disminuir

El control `SpinButton` (Botón Aumentar/ Disminuir) se usa generalmente para permitir al usuario seleccionar un valor dentro de un rango de valores numéricos. Consiste en una caja de entrada de texto con botones de flecha arriba y abajo a un lado. Seleccionando uno de los botones causa que el valor aumente o disminuya en el rango de valores posibles. La caja de entrada también puede editarse directamente para introducir un valor específico.

El `SpinButton` permite que el valor tenga cero o más cifras decimales y puede incrementarse/decrementarse en pasos configurables. La acción de mantener pulsado uno de los botones opcionalmente provoca una aceleración en el cambio del valor correspondiente al tiempo que se mantenga presionado.

El `SpinButton` usa un objeto Adjustment (Ajuste) para almacenar la información del rango de valores que el botón aumentar/disminuir puede tomar. Esto le hace un control muy útil.

Recordemos que un control Adjustment (Ajuste) se crea con la siguiente función, que muestra la información que almacena:

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

Estos atributos de un `Adjustment` (Ajuste) se usan en el `SpinButton` de la siguiente manera:

value	valor inicial para el Botón Aumentar/Disminuir
lower	el valor más bajo del rango
upper	el valor más alto del rango
step_increment	valor que se incrementa/decrementa cuando se pulsa el botón-1 del ratón en un botón
page_increment	valor que se incrementa/decrementa cuando se pulsa el botón-2 del ratón en un botón
page_size	no se usa

Adicionalmente, el botón del ratón botón-3 se puede usar para saltar directamente a los valores *upper* (superior) y *lower* (inferior) cuando se usa para seleccionar uno de los botones. Veamos como crear un `SpinButton` (Botón Aumentar/Disminuir):

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

El argumento *climb_rate* (razón de escalada) puede tomar un valor entre 0.0 y 1.0 e indica la cantidad de aceleración que el `SpinButton` tiene. El argumento *digits* especifica el número de cifras decimales que se mostrarán.

Un `SpinButton` se puede reconfigurar después de su creación usando el siguiente método:

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

La variable *spin_button* especifica el botón aumentar/disminuir que se va a reconfigurar. Los otros argumentos son los mismos que antes.

El *adjustment* (ajuste) se puede fijar y recupar independientemente usando los siguientes dos métodos:

```
spin_button.set_adjustment(adjustment)

adjustment = spin_button.get_adjustment()
```

El número de cifras decimales también se puede cambiar usando:

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

El valor que un `SpinButton` está mostrando actualmente se puede cambiar usando el siguiente método:

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

El valor actual de un `SpinButton` se puede recuperar como un valor real o como un valor entero usando los siguientes métodos:

```
float_value = spin_button.get_value()

int_value = spin_button.get_value_as_int()
```

Si quieres alterar el valor de un SpinButton relativo a su valor actual, entonces usa el siguiente método:

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

El parámetro *direction* (dirección) puede tomar uno de los siguientes valores:

```
SPIN_STEP_FORWARD  # paso hacia adelante
SPIN_STEP_BACKWARD # paso hacia atrás
SPIN_PAGE_FORWARD  # página hacia adelante
SPIN_PAGE_BACKWARD # página hacia atrás
SPIN_HOME          # inicio
SPIN_END           # fin
SPIN_USER_DEFINED  # definido por el usuario
```

Este método comprime bastante funcionalidad, que explicaremos ahora. Muchos de estos parámetros usan valores del objeto Adjustment (Ajuste) que está asociado con un SpinButton (Botón Aumentar/Disminuir).

SPIN_STEP_FORWARD (paso adelante) y SPIN_STEP_BACKWARD (paso atrás) cambian el valor del SpinButton con una cantidad especificada por el *increment* (incremento), a menos que *increment* (incremento) sea igual a 0, en cuyo caso el valor se modifica con el *step_increment* (incremento de paso) del Adjustment.

SPIN_PAGE_FORWARD (página adelante) y SPIN_PAGE_BACKWARD (página atrás) simplemente alteran el valor del SpinButton por *increment* (incremento).

SPIN_HOME (inicio) pone el valor del SpinButton a la parte de abajo del rango del Adjustment .

SPIN_END (fin) fija el valor del SpinButton a la parte de arriba del rango del Adjustment .

SPIN_USER_DEFINED (definido por el usuario) simplemente modifica el valor del SpinButton con la cantidad especificada.

Ahora nos alejamos de los métodos para fijar y recuperar los atributos del rango de un SpinButton, y nos centramos en los métodos que modifican la apariencia y comportamiento del propio control SpinButton.

El primero de estos métodos se usa para limitar la caja de texto del SpinButton para que solo contenga un valor numérico. Esto evita que el usuario escriba cualquier otra cosa que no sea un valor numérico dentro de la caja de texto de un SpinButton:

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

El argumento *numeric* es TRUE para limitar la caja de texto a valores numéricos o FALSE para quitar esta limitación.

Puedes fijar si quieres que el valor del SpinButton se quede en los valores inferior y superior del rango con el siguiente método:

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

El SpinButton limitará los valores dentro del rango si *wrap* es TRUE.

Puedes hacer que el SpinButton redondee el valor al *step_increment* (incremento) más cercano, lo cual se fija dentro del objeto Adjustment usado en el SpinButton. Esto se consigue con el siguiente método cuando el argumento *snap_to_ticks* es TRUE :

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

La política de actualización de un SpinButton se cambia con el siguiente método:

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

Los valores posibles de esta política son:

```
UPDATE_ALWAYS   # actualizar siempre

UPDATE_IF_VALID # actualizar si es válido
```

Estas políticas afectan el comportamiento de un `SpinButton` cuando analiza el texto insertado y sincroniza su valor con los valores del Adjustment.

En el caso de `UPDATE_IF_VALID` (actualizar si es válido) el valor del `SpinButton` sólo cambia si el texto es un valor numérico que está dentro del rango especificado por el Adjustment. En cualquier otro caso el texto se resetea al valor actual.

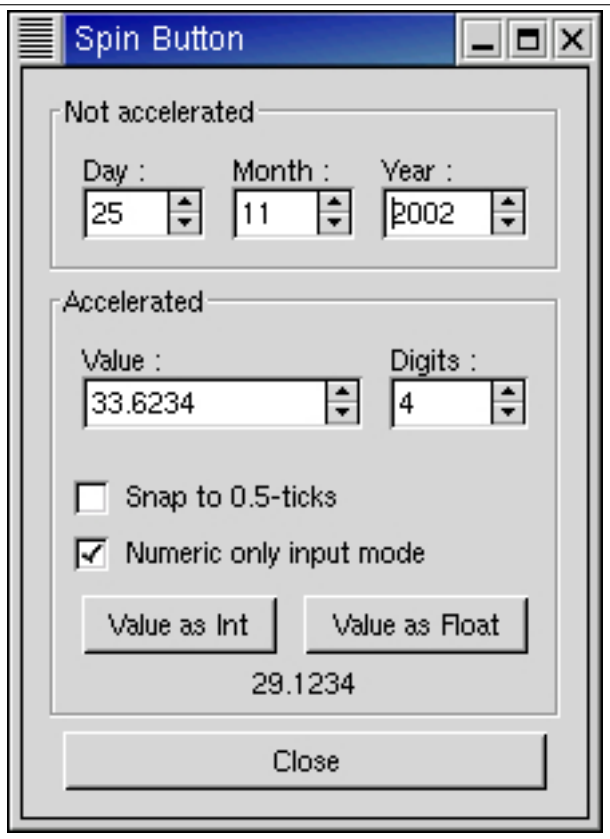
En el caso de `UPDATE_ALWAYS` (actualizar siempre) se ignoran los errores de conversión al pasar el texto a un valor numérico.

Finalmente, se puede solicitar una actualización del `SpinButton` :

```
spin_button.update()
```

El programa de ejemplo spinbutton.py muestra el uso de botones de aumentar/disminuir incluyendo el uso de varias características. La figura Figura 9.11 muestra el resultado de ejecutar el programa de ejemplo:

Figura 9.11 Ejemplo de Botón Aumentar/Disminuir



El código fuente spinbutton.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo spinbutton.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class SpinButtonExample:
10     def toggle_snap(self, widget, spin):
11         spin.set_snap_to_ticks(widget.get_active())
12
13     def toggle_numeric(self, widget, spin):
14         spin.set_numeric(widget.get_active())
15
16     def change_digits(self, widget, spin, spin1):
```

```
17          spin1.set_digits(spin.get_value_as_int())
18
19      def get_value(self, widget, data, spin, spin2, label):
20          if data == 1:
21              buf = "%d" % spin.get_value_as_int()
22          else:
23              buf = "%0.*f" % (spin2.get_value_as_int(),
24                               spin.get_value())
25          label.set_text(buf)
26
27      def __init__(self):
28          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
29          window.connect("destroy", lambda w: gtk.main_quit())
30          window.set_title("Spin Button")
31
32          main_vbox = gtk.VBox(gtk.FALSE, 5)
33          main_vbox.set_border_width(10)
34          window.add(main_vbox)
35
36          frame = gtk.Frame("Not accelerated")
37          main_vbox.pack_start(frame, gtk.TRUE, gtk.TRUE, 0)
38
39          vbox = gtk.VBox(gtk.FALSE, 0)
40          vbox.set_border_width(5)
41          frame.add(vbox)
42
43          # Botones de aumentar/disminuir día, mes y año
44          hbox = gtk.HBox(gtk.FALSE, 0)
45          vbox.pack_start(hbox, gtk.TRUE, gtk.TRUE, 5)
46
47          vbox2 = gtk.VBox(gtk.FALSE, 0)
48          hbox.pack_start(vbox2, gtk.TRUE, gtk.TRUE, 5)
49
50          label = gtk.Label("Day :")
51          label.set_alignment(0, 0.5)
52          vbox2.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
53
54          adj = gtk.Adjustment(1.0, 1.0, 31.0, 1.0, 5.0, 0.0)
55          spinner = gtk.SpinButton(adj, 0, 0)
56          spinner.set_wrap(gtk.TRUE)
57          vbox2.pack_start(spinner, gtk.FALSE, gtk.TRUE, 0)
58
59          vbox2 = gtk.VBox(gtk.FALSE, 0)
60          hbox.pack_start(vbox2, gtk.TRUE, gtk.TRUE, 5)
61
62          label = gtk.Label("Month :")
63          label.set_alignment(0, 0.5)
64          vbox2.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
65
66          adj = gtk.Adjustment(1.0, 1.0, 12.0, 1.0, 5.0, 0.0)
67          spinner = gtk.SpinButton(adj, 0, 0)
68          spinner.set_wrap(gtk.TRUE)
69          vbox2.pack_start(spinner, gtk.FALSE, gtk.TRUE, 0)
70
71          vbox2 = gtk.VBox(gtk.FALSE, 0)
72          hbox.pack_start(vbox2, gtk.TRUE, gtk.TRUE, 5)
73
74          label = gtk.Label("Year :")
75          label.set_alignment(0, 0.5)
76          vbox2.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
77
78          adj = gtk.Adjustment(1998.0, 0.0, 2100.0, 1.0, 100.0, 0.0)
79          spinner = gtk.SpinButton(adj, 0, 0)
80          spinner.set_wrap(gtk.FALSE)
```

```
 81          spinner.set_size_request(55, -1)
 82          vbox2.pack_start(spinner, gtk.FALSE, gtk.TRUE, 0)
 83
 84          frame = gtk.Frame("Accelerated")
 85          main_vbox.pack_start(frame, gtk.TRUE, gtk.TRUE, 0)
 86
 87          vbox = gtk.VBox(gtk.FALSE, 0)
 88          vbox.set_border_width(5)
 89          frame.add(vbox)
 90
 91          hbox = gtk.HBox(gtk.FALSE, 0)
 92          vbox.pack_start(hbox, gtk.FALSE, gtk.TRUE, 5)
 93
 94          vbox2 = gtk.VBox(gtk.FALSE, 0)
 95          hbox.pack_start(vbox2, gtk.TRUE, gtk.TRUE, 5)
 96
 97          label = gtk.Label("Value :")
 98          label.set_alignment(0, 0.5)
 99          vbox2.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
100
101          adj = gtk.Adjustment(0.0, -10000.0, 10000.0, 0.5, 100.0, 0.0)
102          spinner1 = gtk.SpinButton(adj, 1.0, 2)
103          spinner1.set_wrap(gtk.TRUE)
104          spinner1.set_size_request(100, -1)
105          vbox2.pack_start(spinner1, gtk.FALSE, gtk.TRUE, 0)
106
107          vbox2 = gtk.VBox(gtk.FALSE, 0)
108          hbox.pack_start(vbox2, gtk.TRUE, gtk.TRUE, 5)
109
110          label = gtk.Label("Digits :")
111          label.set_alignment(0, 0.5)
112          vbox2.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
113
114          adj = gtk.Adjustment(2, 1, 5, 1, 1, 0)
115          spinner2 = gtk.SpinButton(adj, 0.0, 0)
116          spinner2.set_wrap(gtk.TRUE)
117          adj.connect("value_changed", self.change_digits, spinner2, spinner1 ↩
    )
118          vbox2.pack_start(spinner2, gtk.FALSE, gtk.TRUE, 0)
119
120          hbox = gtk.HBox(gtk.FALSE, 0)
121          vbox.pack_start(hbox, gtk.FALSE, gtk.TRUE, 5)
122
123          button = gtk.CheckButton("Snap to 0.5-ticks")
124          button.connect("clicked", self.toggle_snap, spinner1)
125          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
126          button.set_active(gtk.TRUE)
127
128          button = gtk.CheckButton("Numeric only input mode")
129          button.connect("clicked", self.toggle_numeric, spinner1)
130          vbox.pack_start(button, gtk.TRUE, gtk.TRUE, 0)
131          button.set_active(gtk.TRUE)
132
133          val_label = gtk.Label("")
134
135          hbox = gtk.HBox(gtk.FALSE, 0)
136          vbox.pack_start(hbox, gtk.FALSE, gtk.TRUE, 5)
137          button = gtk.Button("Value as Int")
138          button.connect("clicked", self.get_value, 1, spinner1, spinner2,
139                         val_label)
140          hbox.pack_start(button, gtk.TRUE, gtk.TRUE, 5)
141
142          button = gtk.Button("Value as Float")
143          button.connect("clicked", self.get_value, 2, spinner1, spinner2,
```

```
144                         val_label)
145         hbox.pack_start(button, gtk.TRUE, gtk.TRUE, 5)
146
147         vbox.pack_start(val_label, gtk.TRUE, gtk.TRUE, 0)
148         val_label.set_text("0")
149
150         hbox = gtk.HBox(gtk.FALSE, 0)
151         main_vbox.pack_start(hbox, gtk.FALSE, gtk.TRUE, 0)
152
153         button = gtk.Button("Close")
154         button.connect("clicked", lambda w: gtk.main_quit())
155         hbox.pack_start(button, gtk.TRUE, gtk.TRUE, 5)
156         window.show_all()
157
158 def main():
159     gtk.main()
160     return 0
161
162 if __name__ == "__main__":
163     SpinButtonExample()
164     main()
```

9.11. Lista Desplegable (Combo)

NOTA

El control de Lista Desplegable `Combo` está obsoleto a partir de la versión 2.4 de PyGTK.

La lista desplegable `Combo` es otro control bastante simple que es realmente una colección de otros controles. Desde el punto de vista del usuario, el control consiste en una caja de entrada de texto y un menú desplegable desde el que se puede seleccionar una entrada a partir de un conjunto predefinido. Alternativamente, el usuario puede escribir una opción diferente directamente en la caja de texto.

El `Combo` tiene dos partes principales de las que preocuparse: una *`entry`* (entrada) y una *`list`* (lista). Se puede acceder a ellas usando los atributos:

```
combo.entry

combo.list
```

Lo primero, para crear una lista desplegable, se usa:

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

Ahora, si se quiere fijar la cadena en la sección de la entrada de la lista desplegable, esto se hace manipulando el control entrada directamente:

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

Para fijar los valores de la lista desplegable, se usa el método:

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

Antes de que se pueda hacer esto, hay que componer una lista con las opciones que se deseen.

Aqui tenemos el típico código para crear un conjunto de opciones:

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

combo.set_popdown_strings(slist)
```

En este punto ya se tiene una lista desplegable funcionando. Hay unos cuantos aspectos de su comportamiento que se pueden cambiar. Esto se consigue con los métodos:

```
combo.set_use_arrows(val)

combo.set_use_arrows_always(val)

combo.set_case_sensitive(val)
```

El método set_use_arrows() permite al usuario cambiar el valor de la entrada usando las teclas de flecha arriba/abajo cuando *val* se pone a TRUE. Esto no despliega la lista, si no que sustituye el texto actual de la entrada con la siguiente entrada de la lista (arriba o abajo, según la combinación de tecla indique). Esto se hace buscando en la lista el elemento correspondiente al valor actual de la entrada y seleccionando el elemento anterior/siguiente correspondiente. Normalmente en una entrada las teclas de flecha se usan para cambiar el foco (también puedes hacer esto usando el tabulador). Ten en cuenta que cuando el elemento actual es el último de la lista y pulsas la tecla flecha abajo se cambia el foco (lo mismo ocurre cuando estas en el primer elemento y pulsas la tecla flecha arriba).

Si el valor actual de la entrada no está en la lista, el método set_use_arrows() se desactiva.

El método set_use_arrows_always() , cuando *val* es TRUE, también permite al usuario el uso de las teclas de flecha arriba/abajo para ciclar por las opciones de la lista desplegable, excepto que da la vuelta a los valores de la lista, desactivando por completo el uso de las flechas arriba y abajo para cambiar el foco.

El método set_case_sensitive() dice si GTK busca o no las entradas de una forma sensible a mayúsculas. Esto se usa cuando se le pide al control Combo que busque un valor de la lista usando la entrada actual de la caja de texto. Este completado puede producirse de forma sensible o insensible a mayúsculas, dependiendo de lo que le pasemos a este método. El control Combo también puede simplemente completar la entrada actual si el usuario pulsa la combinación de teclas MOD-1-Tab. MOD-1 normalmente corresponde a la tecla **Alt**, gracias a la utilidad **xmodmap**. Ten en cuenta, sin embargo, que algunos manejadores de ventana también usan esta combinación de teclas, lo que inutilizará su uso en GTK.

Ahora que tenemos una lista desplegable, y que tiene la apariencia y el comportamiento que queremos, lo único que nos falta es la capacidad de obtener los datos de la lista desplegable. Esto es relativamente directo. La mayoría del tiempo, de lo único que en necesario preocuparse es de obtener los datos de la entrada. La entrada es accesible simplemente como *combo.entry*. Las dos cosas fundamentales que se querrán hacer con ella es conectarle la señal "activate", que indica que el usuario ha pulsado la tecla **Return** o la tecla **Enter**, y leer el texto. Lo primero se consigue usando algo como:

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

Obtener el texto en cualquier momento se consigue simplemente usando el siguiente método:

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

Eso es todo lo importante. Hay un método:

```
combo.disable_activate()
```

que desactivará la señal "activate" en el control de entrada de la lista desplegable. Personalmente, no se me ocurre ninguna situación en la que se quiera usar, pero existe.

9.12. Calendario

El control Calendar (Calendario) es una forma efectiva para visualizar y obtener información relativa a fechas mensuales. Es un control muy fácil de usar y trabajar con él.

Crear un control GtkCalendar es tan simple como:

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

El calendario mostrará el mes y el año actual de manera predeterminada.

Puede haber ocasiones en las que se necesite cambiar mucha información dentro de este control y los siguientes métodos permiten realizar múltiples cambios al control Calendar sin que el usuario vea muchos cambios en pantalla.

```
calendar.freeze() # congelar

calendar.thaw()   # reanudar
```

Funcionan exactamente igual que los métodos freeze/thaw de cualquier otro control (desactivando los cambios y reanudándolos).

El control `Calendar` tiene unas cuantas opciones que permiten cambiar la manera en la que el control se visualiza y se comporta usando el método:

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

El argumento *`flags`* (banderas) se puede formar combinando cualquiera de las siguientes cinco opciones usando el operador lógico (`|`):

CALENDAR_SHOW_HEADING	esta opción especifica que el mes y el año deben mostrarse cuando se dibuje el calendario.
CALENDAR_SHOW_DAY_NAMES	esta opción especifica que la descripción de tres letras para cada día (Lun, Mar, etc.) debe mostrarse.
CALENDAR_NO_MONTH_CHANGE	esta opción dice que el usuario no podrá cambiar el mes que se muestra. Esto puede ser bueno si sólo se necesita mostrar un mes en particular como cuando se muestran 12 controles de calendario uno para cada mes dentro de un mes en particular.
CALENDAR_SHOW_WEEK_NUMBERS	esta opción especifica que se muestre el número de cada semana en la parte de abajo izquierda del calendario (ejemplo: Enero 1 = Semana 1, Diciembre 31 = Semana 52).
CALENDAR_WEEK_START_MONDAY	esta opción dice que la semana empezará en Lunes en lugar de en Domingo, que es el valor predeterminado. Esto solo afecta al orden en el que se muestran los días de izquierda a derecha. A partir de PyGTK 2.4 esta opción está obsoleta.

Los siguientes métodos se usan para fijar la fecha que se muestra:

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

calendar.select_day(day)
```

El valor que devuelve el método select_month() es un valor booleano que indica si la selección tuvo éxito.

Con el método select_day() el día especificado se selecciona dentro del mes actual, si eso es posible. Un valor para el día de 0 limpiará la selección actual.

Además de tener un día seleccionado, un número arbitrario de días se pueden "marcar". Un día marcado se destaca en el calendario. Los siguientes métodos se proporcionan para manipular días marcados:

```
result = calendar.mark_day(day)

result = calendar.unmark_day(day)

calendar.clear_marks()
```

mark_day() y unmark_day() devuelven un valor booleano que indica si el método tuvo éxito. Ha de advertirse que las marcas son persistentes entre cambios de mes y año.

El último método del control `Calendar` se usa para obtener la fecha seleccionada, mes y/o año.

```
año, mes, día = calendar.get_date()
```

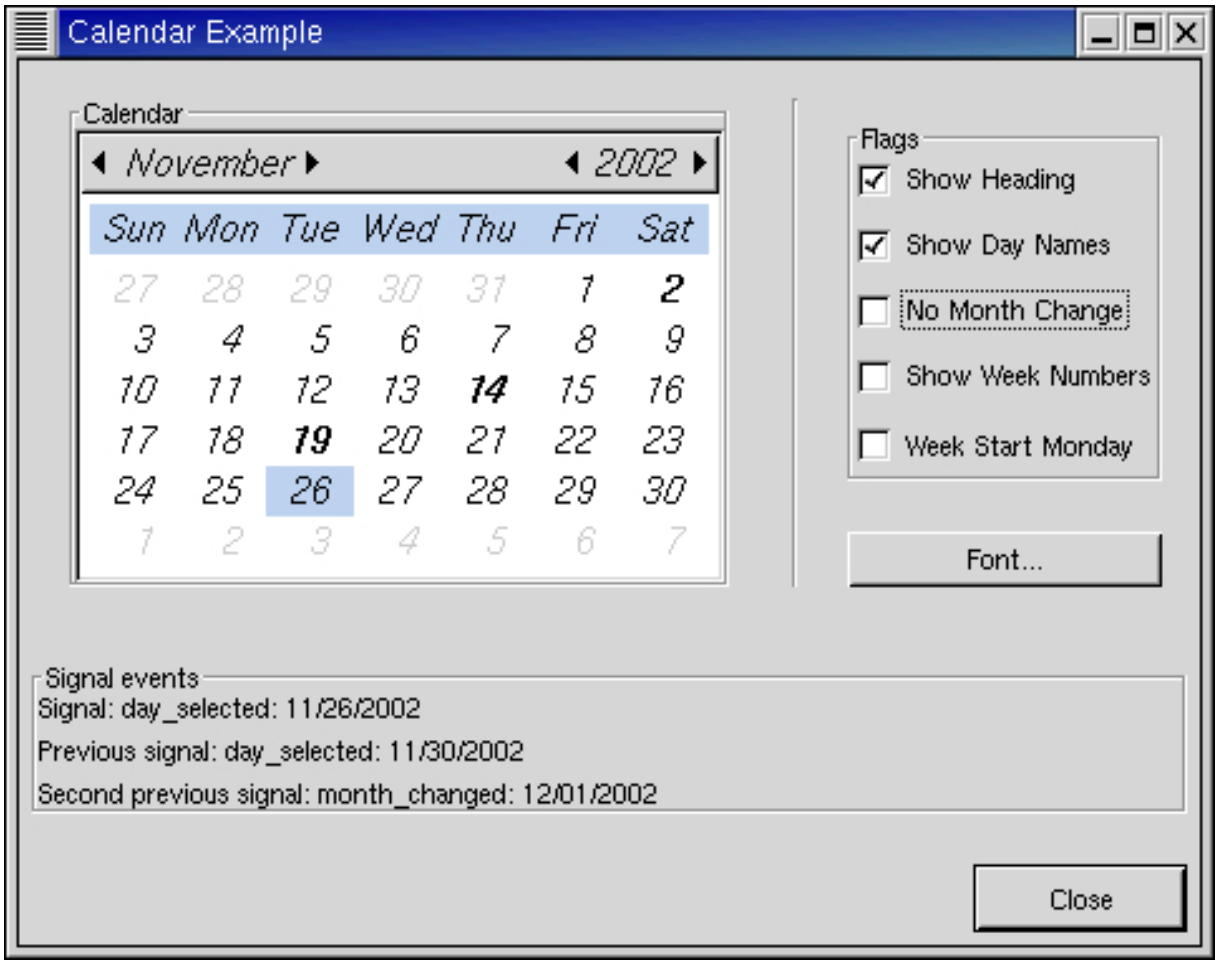
El control `Calendar` puede generar varias señales que indican la selección y cambio de la fecha. Los nombres de estas señales son autoexplicativos, y son:

```
month_changed # cambio de mes

day_selected  # día seleccionado
```

```
day_selected_double_click # doble clic en día seleccionado

prev_month    # mes anterior

next_month    # mes siguiente

prev_year     # año anterior

next_year     # año siguiente
```

Esto nos deja con la necesidad de poner todo ello junto en el programa de ejemplo calendar.py. La figura Figura 9.12 muestra el resultado del programa:

Figura 9.12 Ejemplo de Calendario



El código fuente es calendar.py:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo calendar.py
 4 #
 5 # Copyright (C) 1998 Cesar Miquel, Shawn T. Amundson, Mattias Gronlund
 6 # Copyright (C) 2000 Tony Gale
 7 # Copyright (C) 2001-2004 John Finlay
 8 #
 9 # This program is free software; you can redistribute it and/or modify
10 # it under the terms of the GNU General Public License as published by
11 # the Free Software Foundation; either version 2 of the License, or
12 # (at your option) any later version.
13 #
```

```
14 # This program is distributed in the hope that it will be useful,
15 # but WITHOUT ANY WARRANTY; without even the implied warranty of
16 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 # GNU General Public License for more details.
18 #
19 # You should have received a copy of the GNU General Public License
20 # along with this program; if not, write to the Free Software
21 # Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
22
23 import pygtk
24 pygtk.require('2.0')
25 import gtk, pango
26 import time
27
28 class CalendarExample:
29     DEF_PAD = 10
30     DEF_PAD_SMALL = 5
31     TM_YEAR_BASE = 1900
32
33     calendar_show_header = 0
34     calendar_show_days = 1
35     calendar_month_change = 2
36     calendar_show_week = 3
37
38     def calendar_date_to_string(self):
39         year, month, day = self.window.get_date()
40         mytime = time.mktime((year, month+1, day, 0, 0, 0, 0, 0, -1))
41         return time.strftime("%x", time.localtime(mytime))
42
43     def calendar_set_signal_strings(self, sig_str):
44         prev_sig = self.prev_sig.get()
45         self.prev2_sig.set_text(prev_sig)
46
47         prev_sig = self.last_sig.get()
48         self.prev_sig.set_text(prev_sig)
49         self.last_sig.set_text(sig_str)
50
51     def calendar_month_changed(self, widget):
52         buffer = "month_changed: %s" % self.calendar_date_to_string()
53         self.calendar_set_signal_strings(buffer)
54
55     def calendar_day_selected(self, widget):
56         buffer = "day_selected: %s" % self.calendar_date_to_string()
57         self.calendar_set_signal_strings(buffer)
58
59     def calendar_day_selected_double_click(self, widget):
60         buffer = "day_selected_double_click: %s"
61         buffer = buffer % self.calendar_date_to_string()
62         self.calendar_set_signal_strings(buffer)
63
64         year, month, day = self.window.get_date()
65
66         if self.marked_date[day-1] == 0:
67             self.window.mark_day(day)
68             self.marked_date[day-1] = 1
69         else:
70             self.window.unmark_day(day)
71             self.marked_date[day-1] = 0
72
73     def calendar_prev_month(self, widget):
74         buffer = "prev_month: %s" % self.calendar_date_to_string()
75         self.calendar_set_signal_strings(buffer)
76
77     def calendar_next_month(self, widget):
```

```
 78         buffer = "next_month: %s" % self.calendar_date_to_string()
 79         self.calendar_set_signal_strings(buffer)
 80 
 81     def calendar_prev_year(self, widget):
 82         buffer = "prev_year: %s" % self.calendar_date_to_string()
 83         self.calendar_set_signal_strings(buffer)
 84 
 85     def calendar_next_year(self, widget):
 86         buffer = "next_year: %s" % self.calendar_date_to_string()
 87         self.calendar_set_signal_strings(buffer)
 88 
 89     def calendar_set_flags(self):
 90         options = 0
 91         for i in range(5):
 92             if self.settings[i]:
 93                 options = options + (1<<i)
 94         if self.window:
 95             self.window.display_options(options)
 96 
 97     def calendar_toggle_flag(self, toggle):
 98         j = 0
 99         for i in range(5):
100             if self.flag_checkboxes[i] == toggle:
101                 j = i
102 
103         self.settings[j] = not self.settings[j]
104         self.calendar_set_flags()
105 
106     def calendar_font_selection_ok(self, button):
107         self.font = self.font_dialog.get_font_name()
108         if self.window:
109             font_desc = pango.FontDescription(self.font)
110             if font_desc:
111                 self.window.modify_font(font_desc)
112 
113     def calendar_select_font(self, button):
114         if not self.font_dialog:
115             window = gtk.FontSelectionDialog("Font Selection Dialog")
116             self.font_dialog = window
117 
118             window.set_position(gtk.WIN_POS_MOUSE)
119 
120             window.connect("destroy", self.font_dialog_destroyed)
121 
122             window.ok_button.connect("clicked",
123                                      self.calendar_font_selection_ok)
124             window.cancel_button.connect_object("clicked",
125                                                 lambda wid: wid.destroy(),
126                                                 self.font_dialog)
127         window = self.font_dialog
128         if not (window.flags() & gtk.VISIBLE):
129             window.show()
130         else:
131             window.destroy()
132             self.font_dialog = None
133 
134     def font_dialog_destroyed(self, data=None):
135         self.font_dialog = None
136 
137     def __init__(self):
138         flags = [
139             "Show Heading",
140             "Show Day Names",
141             "No Month Change",
```

```
142             "Show Week Numbers",
143             ]
144         self.window = None
145         self.font = None
146         self.font_dialog = None
147         self.flag_checkboxes = 5*[None]
148         self.settings = 5*[0]
149         self.marked_date = 31*[0]
150 
151         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
152         window.set_title("Calendar Example")
153         window.set_border_width(5)
154         window.connect("destroy", lambda x: gtk.main_quit())
155 
156         window.set_resizable(gtk.FALSE)
157 
158         vbox = gtk.VBox(gtk.FALSE, self.DEF_PAD)
159         window.add(vbox)
160 
161         # La parte superior de la ventana, el Calendario, las opcines y  ↩
    fuente.
162         hbox = gtk.HBox(gtk.FALSE, self.DEF_PAD)
163         vbox.pack_start(hbox, gtk.TRUE, gtk.TRUE, self.DEF_PAD)
164         hbbox = gtk.HButtonBox()
165         hbox.pack_start(hbbox, gtk.FALSE, gtk.FALSE, self.DEF_PAD)
166         hbbox.set_layout(gtk.BUTTONBOX_SPREAD)
167         hbbox.set_spacing(5)
168 
169         # Control calendario
170         frame = gtk.Frame("Calendar")
171         hbbox.pack_start(frame, gtk.FALSE, gtk.TRUE, self.DEF_PAD)
172         calendar = gtk.Calendar()
173         self.window = calendar
174         self.calendar_set_flags()
175         calendar.mark_day(19)
176         self.marked_date[19-1] = 1
177         frame.add(calendar)
178         calendar.connect("month_changed", self.calendar_month_changed)
179         calendar.connect("day_selected", self.calendar_day_selected)
180         calendar.connect("day_selected_double_click",
181                          self.calendar_day_selected_double_click)
182         calendar.connect("prev_month", self.calendar_prev_month)
183         calendar.connect("next_month", self.calendar_next_month)
184         calendar.connect("prev_year", self.calendar_prev_year)
185         calendar.connect("next_year", self.calendar_next_year)
186 
187         separator = gtk.VSeparator()
188         hbox.pack_start(separator, gtk.FALSE, gtk.TRUE, 0)
189 
190         vbox2 = gtk.VBox(gtk.FALSE, self.DEF_PAD)
191         hbox.pack_start(vbox2, gtk.FALSE, gtk.FALSE, self.DEF_PAD)
192 
193         # Crear el frame derecho con sus opciones
194         frame = gtk.Frame("Flags")
195         vbox2.pack_start(frame, gtk.TRUE, gtk.TRUE, self.DEF_PAD)
196         vbox3 = gtk.VBox(gtk.TRUE, self.DEF_PAD_SMALL)
197         frame.add(vbox3)
198 
199         for i in range(len(flags)):
200             toggle = gtk.CheckButton(flags[i])
201             toggle.connect("toggled", self.calendar_toggle_flag)
202             vbox3.pack_start(toggle, gtk.TRUE, gtk.TRUE, 0)
203             self.flag_checkboxes[i] = toggle
204 
```

```
205         # Crear el botón de fuentes derecho
206         button = gtk.Button("Font...")
207         button.connect("clicked", self.calendar_select_font)
208         vbox2.pack_start(button, gtk.FALSE, gtk.FALSE, 0)
209
210         #  Crear la parte relativo a señales
211         frame = gtk.Frame("Signal events")
212         vbox.pack_start(frame, gtk.TRUE, gtk.TRUE, self.DEF_PAD)
213
214         vbox2 = gtk.VBox(gtk.TRUE, self.DEF_PAD_SMALL)
215         frame.add(vbox2)
216
217         hbox = gtk.HBox (gtk.FALSE, 3)
218         vbox2.pack_start(hbox, gtk.FALSE, gtk.TRUE, 0)
219         label = gtk.Label("Signal:")
220         hbox.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
221         self.last_sig = gtk.Label("")
222         hbox.pack_start(self.last_sig, gtk.FALSE, gtk.TRUE, 0)
223
224         hbox = gtk.HBox (gtk.FALSE, 3)
225         vbox2.pack_start(hbox, gtk.FALSE, gtk.TRUE, 0)
226         label = gtk.Label("Previous signal:")
227         hbox.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
228         self.prev_sig = gtk.Label("")
229         hbox.pack_start(self.prev_sig, gtk.FALSE, gtk.TRUE, 0)
230
231         hbox = gtk.HBox (gtk.FALSE, 3)
232         vbox2.pack_start(hbox, gtk.FALSE, gtk.TRUE, 0)
233         label = gtk.Label("Second previous signal:")
234         hbox.pack_start(label, gtk.FALSE, gtk.TRUE, 0)
235         self.prev2_sig = gtk.Label("")
236         hbox.pack_start(self.prev2_sig, gtk.FALSE, gtk.TRUE, 0)
237
238         bbox = gtk.HButtonBox ()
239         vbox.pack_start(bbox, gtk.FALSE, gtk.FALSE, 0)
240         bbox.set_layout(gtk.BUTTONBOX_END)
241
242         button = gtk.Button("Close")
243         button.connect("clicked", lambda w: gtk.main_quit())
244         bbox.add(button)
245         button.set_flags(gtk.CAN_DEFAULT)
246         button.grab_default()
247
248         window.show_all()
249
250 def main():
251     gtk.main()
252     return 0
253
254 if __name__ == "__main__":
255     CalendarExample()
256     main()
```

9.13. Selección de Color

El control de selección de color es, como cabe de esperar, un control para seleccionar colores interactivamente. Este control compuesto permite al usuario seleccionar un color manipulando triples RGB (Rojo, Verde, Azul) y HSV (Tono, Saturación, Valor). Esto se consigue ajustando valores simples con deslizadores o entradas, o haciendo clic en el color deseado en una rueda de tono-saturación y una barra de valor. Opcionalmente, la opacidad del color también se puede especificar.

El control de selección de color solo emite una señal por ahora, "color_changed", que se emite siempre que el color actual del control cambie, bien porque el usuario lo cambia o porque se especifique

explícitamente a través del método set_color() .

Veamos lo que nos ofrece el control de selección de color. El control viene en dos sabores: `gtk.ColorSelection` y `gtk.ColorSelectionDialog`.

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

Probablemente no se use este constructor directamente. Se crea un control `ColorSelection` huerfano que habría que emparentar uno mismo. El control `ColorSelection` hereda del control `VBox`.

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

donde *`title`* (título) es una cadena usada para la barra de título del diálogo.

Este es el constructor más común del selector de color. Crea un `ColorSelectionDialog`. Éste consiste en un `Frame` que contiene un control `ColorSelection`, un `HSeparator` y un `HBox` con tres botones, Ok, Cancelar y Ayuda. Se pueden obtener estos botones accediendo a los atributos *`ok_button`*, *`cancel_button`* y *`help_button`* del `ColorSelectionDialog`, (por ejemplo, *`colorseldlg.ok_button`*). El control `ColorSelection` es accesible usando la variable *`colorsel`*:

```
colorsel = colorseldlg.colorsel
```

El control `ColorSelection` tiene unos cuantos métodos que cambian sus características o proporcionan acceso a la selección de color.

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

El control de selección de color permite ajustar la opacidad de un color (también conocida como el canal alfa). Esto esta desactivado por defecto. Llamando a este método con *`has_opacity`* igual `TRUE` activa la opacidad. De la misma forma, *`has_opacity`* igual a `FALSE` desactivará la opacidad.

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

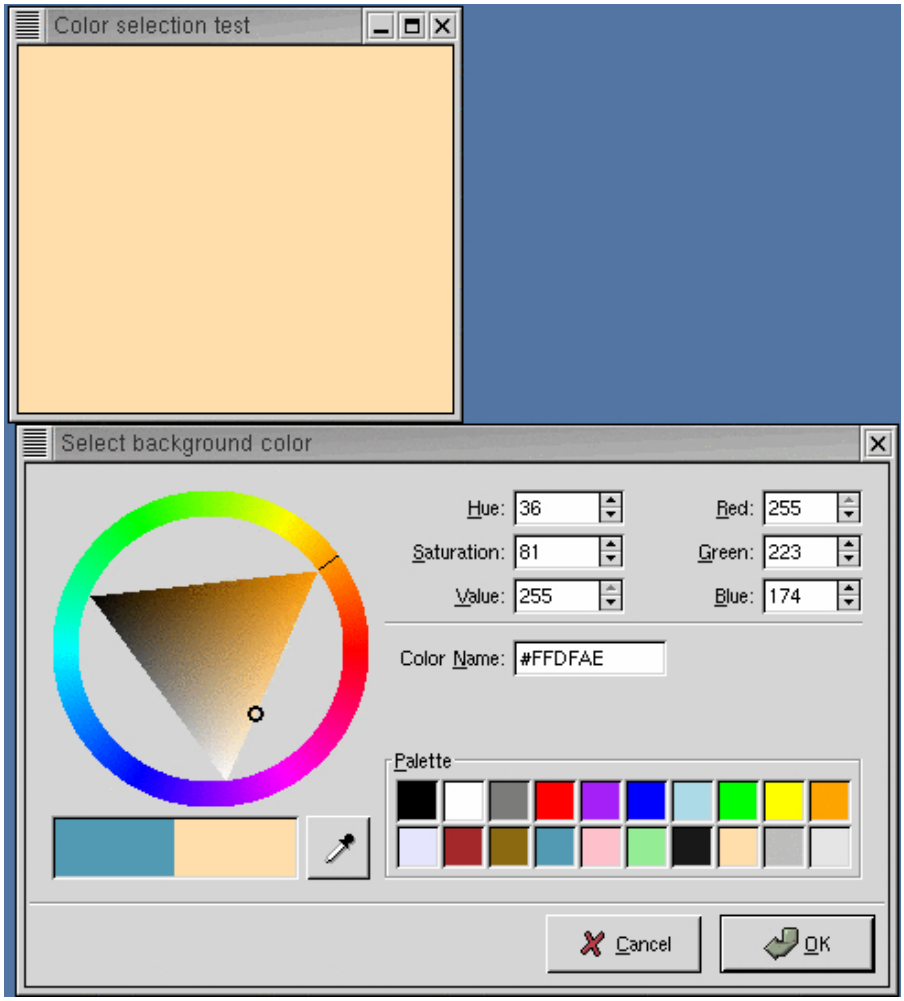
Puedes poner el color actual explícitamente llamando al método set_current_color() con un `GdkColor`. La opacidad (canal alfa) se pone con el método set_current_alpha().El valor del canal alfa *`alpha`* debe estar entre 0 (completamente transparente) y 65636 (completamente opaco).

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

Cuando se tenga que mirar el color actual, típicamente al recibir la señal "color_changed", se pueden usar esos métodos.

El programa de ejemplo colorsel.py demuestra el uso del `ColorSelectionDialog`. Este programa muestra una ventana que contiene un área de dibujo. Al hacer clic en ella se abre un diálogo de selección de color, y cambiando el color en dicho diálogo se cambia el color de fondo. La figura Figura 9.13 muestra el programa en acción:

Figura 9.13 Ejemplo de Diálogo de Selección de Color



El código fuente de colorsel.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo colorsel.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ColorSelectionExample:
10     # manejador de cambio de Color
11     def color_changed_cb(self, widget):
12         # Obtenemos el mapa de color del área de dibujo
13         colormap = self.drawingarea.get_colormap()
14
15         # Obtenemos el color actual
16         color = self.colorseldlg.colorsel.get_current_color()
17
18         # Fijamos el color de fondo de la ventana
19         self.drawingarea.modify_bg(gtk.STATE_NORMAL, color)
20
21     # manejador de eventos del área de dibujo
22     def area_event(self, widget, event):
23         handled = gtk.FALSE
24
25         # Comprobamos si se ha recibido un evento de pulsación de botón
```

```
26          if event.type == gtk.gdk.BUTTON_PRESS:
27              handled = gtk.TRUE
28
29              # Creamos el diálogo de selección de color
30              if self.colorseldlg == None:
31                  self.colorseldlg = gtk.ColorSelectionDialog(
32                      "Select background color")
33
34              # Obtenemos el control ColorSelection
35              colorsel = self.colorseldlg.colorsel
36
37              colorsel.set_previous_color(self.color)
38              colorsel.set_current_color(self.color)
39              colorsel.set_has_palette(gtk.TRUE)
40
41              # Lo conectamos a la señal "color_changed"
42              colorsel.connect("color_changed", self.color_changed_cb)
43              # Mostramos el diálogo
44              response = self.colorseldlg.run()
45
46              if response -- gtk.RESPONSE_OK:
47                  self.color = colorsel.get_current_color()
48              else:
49                  self.drawingarea.modify_bg(gtk.STATE_NORMAL, self.color)
50
51              self.colorseldlg.hide()
52
53          return handled
54
55      # manejador de cierre y salida
56      def destroy_window(self, widget, event):
57          gtk.main_quit()
58          return gtk.TRUE
59
60      def __init__(self):
61          self.colorseldlg = None
62          # creación de la ventana principal, título y políticas
63          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
64          window.set_title("Color selection test")
65          window.set_resizable(gtk.TRUE)
66
67          # La conectamos a los eventos "delete" y "destroy" para poder salir
68          window.connect("delete_event", self.destroy_window)
69
70          # Creamos un área de dibujo, fijamos su tamaño y capturamos eventos ↩
     de botón
71          self.drawingarea = gtk.DrawingArea()
72
73          self.color = self.drawingarea.get_colormap().alloc_color(0, 65535,  ↩
    0)
74
75          self.drawingarea.set_size_request(200, 200)
76          self.drawingarea.set_events(gtk.gdk.BUTTON_PRESS_MASK)
77          self.drawingarea.connect("event",  self.area_event)
78
79          # Añadimos el área de dibujo a la ventana y mostramos ambos  ↩
    controles
80          window.add(self.drawingarea)
81          self.drawingarea.show()
82          window.show()
83
84  def main():
85      gtk.main()
86      return 0
```

```
87
88 if __name__ == "__main__":
89     ColorSelectionExample()
90     main()
```

9.14. Selectores de Fichero

El control de selección de fichero es una forma rápida y fácil de mostrar una caja de diálogo de Fichero. Viene con botones Ok, Cancelar y Ayuda, por lo que es estupendo para ahorrase tiempo de programación.

Para crear una nueva caja de selección de fichero se usa:

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

Para fijar el nombre de fichero, por ejemplo para mostrar un directorio específico, o establecer un fichero predeterminado, usa este método:

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

Para obtener el nombre de fichero que el usuario ha escrito o seleccionado, se usa este método:

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

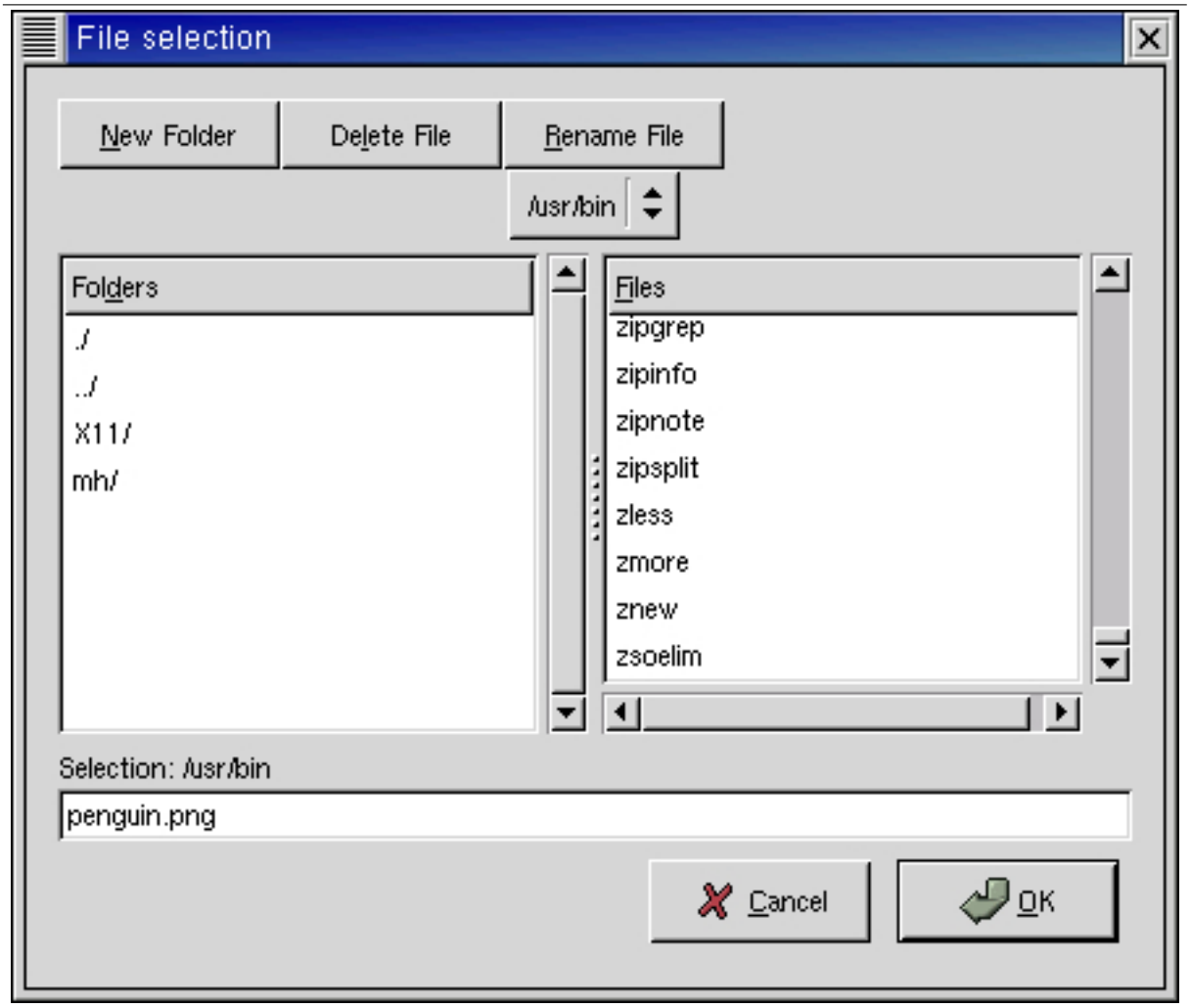
También hay referencias a los controles contenidos en el control de selección de ficheros. Estos son los atributos:

```
filesel.dir_list         # lista de directorios
filesel.file_list        # lista de ficheros
filesel.selection_entry  # entrada de selección
filesel.selection_text   # texto de selección
filesel.main_vbox        # caja vertical principal
filesel.ok_button        # botón ok
filesel.cancel_button    # botón cancelar
filesel.help_button      # botón ayuda
filesel.history_pulldown # lista de historia
filesel.history_menu     # menú de historia
filesel.fileop_dialog # diálogo
filesel.fileop_entry # entrada
filesel.fileop_file # fichero
filesel.fileop_c_dir # cambio de directorio
filesel.fileop_del_file # borrar fichero
filesel.fileop_ren_file # renombrar fichero
filesel.button_area      # área de botones
filesel.action_area      # área de acción
```

Lo más probable es que se quieran usar los atributos *ok_button*, *cancel_button*, y *help_button* para conectar sus señales a las retrollamadas.

El programa de ejemplo filesel.py ilustra el uso del control de selección de ficheros. Como se puede ver, no hay mucho más que decir para crear un control de selección de ficheros. Aunque en este ejemplo el botón Ayuda aparece en pantalla, no hace nada porque no hay ninguna señal conectada a él. La figura Figura 9.14 muestra la pantalla resultante:

Figura 9.14 Ejemplo de Selección de Ficheros



El código fuente de filesel.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo filesel.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class FileSelectionExample:
10     # Obtener el nombre del fichero seleccionado e imprimirlo a la consola
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         # Creamos un nuevo control de selección de fichero
19         self.filew = gtk.FileSelection("File selection")
20
21         self.filew.connect("destroy", self.destroy)
22         # Conectar ok_button al método file_ok_sel
23         self.filew.ok_button.connect("clicked", self.file_ok_sel)
24
25         # Conectar cancel_button para destruir el control
```

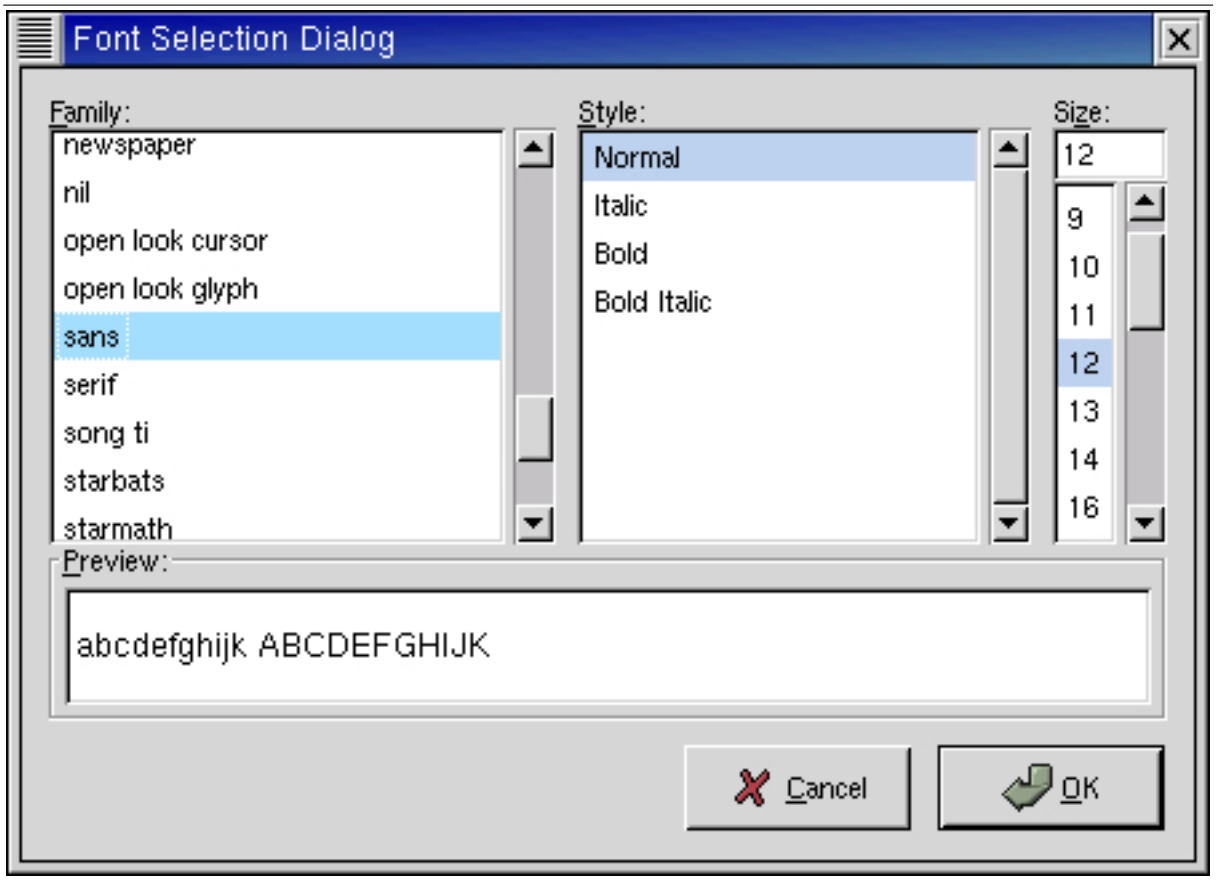
```
26          self.filew.cancel_button.connect("clicked",
27                                           lambda w: self.filew.destroy())
28
29          # Fijamos el nombre de fichero, como si fuese un diálogo de ↩
    guardado,
30          # y damos un nombre por defecto
31          self.filew.set_filename("penguin.png")
32
33          self.filew.show()
34
35 def main():
36     gtk.main()
37     return 0
38
39 if __name__ == "__main__":
40     FileSelectionExample()
41     main()
```

9.15. Diálogo de Selección de Fuentes

El Diálogo de Selección de Fuentes permite al usuario seleccionar una fuente de forma interactiva. El diálogo contiene un control `FontSelection` y botones de OK y Cancelar. Un botón de Aplicar también está disponible en el diálogo, pero inicialmente está oculto. El Diálogo de Selección de Fuentes permite al usuario seleccionar una fuente de las fuentes de sistema disponibles (las mismas que se obtienen al usar **xlsfonts**).

La figura Figura 9.15 ilustra un `FontSelectionDialog`:

Figura 9.15 Diálogo de Selección de Fuentes



El diálogo contiene un conjunto de tres fichas que proporcionan:

- una interfaz para seleccionar la fuente, el estilo y el tamaño
- información detallada sobre la fuente seleccionada
- una interfaz para el mecanismo de filtrado de fuente que restringe las fuentes disponibles para seleccionar

La función para crear un `FontSelectionDialog` es:

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

El *`title`* (título) es una cadena que se usará en el texto de la barra de título.
Una instancia de un Diálogo de Selección de Fuentes tiene varios atributos:

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

El atributo *`fontsel`* es una referencia al control de selección de fuente. *`main_vbox`* es una referencia a la `gtk.VBox` que contiene el *`fontsel`* y el *`action_area`* en el diálogo. El atributo *`action_area`* es una referencia a la `gtk.HButtonBox` que contiene los botones OK, Aplicar y Cancelar. Los atributos *`ok_button`*, *`cancel_button`* y *`apply_button`* son referencias a los botones OK, Cancelar y Aplicar que se pueden usar para realizar las conexiones a las señales de los botones. La referencia *`apply_button`* también se puede usar para mostrar el botón Aplicar mediante el método `show()` .
Se puede fijar la fuente inicial que se mostrará en el diálogo usando el método:

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

El argumento *`fontname`* es el nombre de una fuente de sistema completo o parcialmente especificado. Por ejemplo:

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

especifica una fuente inicial parcialmente.
El nombre de la fuente seleccionada se puede obtener con el método:

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

El Diálogo de Selección de Fuentes tiene un área de previsualización que muestra texto usando la fuente seleccionada. El texto que se usa en el área de previsualización se puede establecer con el método:

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

El texto de previsualización se puede obtener con el método:

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

El programa de ejemplo calendar.py usa un diálogo de selección de fuentes para seleccionar la fuente que se usa para mostrar la información del calendario. Las lineas 105-110 definen una retrollamada para obtener el nombre de la fuente a partir del Diálogo de Selección de Fuentes y lo usa para fijar la fuente para el control del calendario. Las lineas 112-131 definen el método que crea un Diálogo de Selección de Fuentes, configura las retrollamadas para los botones OK y Cancelar y muestra el diálogo.

Capítulo 10

Controles Contenedores

10.1. La Caja de Eventos (EventBox)

Algunos controles GTK no tienen ventanas X asociadas, por lo que simplemente se dibujan encima de sus padres. A causa de esto, no pueden recibir eventos y si se dimensionan incorrectamente, no pueden recortarse por lo que puedes ver partes mal, etc. Si se necesita más de estos controles, la `EventBox` (Caja de Eventos) es lo que se necesita.

A primera vista, el control `EventBox` puede aparecer completamente inútil. No dibuja nada en la pantalla y no responde a ningún evento. Sin embargo, tiene una función - proporciona una ventana X a sus controles hijos. Esto es importante ya que muchos controles GTK no tienen una ventana X asociada. No tener una ventana X ahorra memoria y mejora el rendimiento, pero también tiene inconvenientes. Un control sin ventana X no puede recibir eventos, no realiza ningun recortado sobre sus contenidos y no puede establecer su color de fondo. Aunque el nombre `EventBox` enfatiza la función de manejador de eventos, el control también se puede usar para recorte. (y más, mira el ejemplo más abajo).

Para crear un nuevo control `EventBox`, se usa:

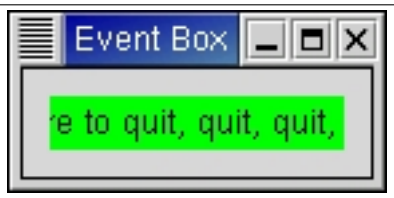
```
event_box = gtk.EventBox()
```

Un control hijo se puede añadir a esta *`event_box`*:

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

El programa de ejemplo eventbox.py muestra dos usos de un `EventBox` - se crea una etiqueta y se recorta en una caja pequeña, tiene un fondo verde y se ha configurado para que un clic de ratón en la etiqueta haga que el programa termine. Al redimensionar la ventana se cambian cantidades en la etiqueta. La figura Figura 10.1 muestra la ventana del programa:

Figura 10.1 Ejemplo de Caja de Eventos



El código fuente de eventbox.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo 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          # Creamos una EventBox y la añadimos a la ventana principal
17          event_box = gtk.EventBox()
18          window.add(event_box)
19          event_box.show()
20
21          # Creamos una etiqueta larga
22          label = gtk.Label("Click here to quit, quit, quit, quit, quit")
23          event_box.add(label)
24          label.show()
25
26          # La recortamos
27          label.set_size_request(110, 20)
28
29          # Y conectamos una acción a la misma
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          # Más cosas para las que se necesita una ventana de X ...
34          event_box.realize()
35          event_box.window.set_cursor(gtk.gdk.Cursor(gtk.gdk.HAND1))
36
37          # Poner el fondo en verde
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. El control Alineador

El control `Alignment` (Alineador) permite colocar un control dentro de su ventana con una posición y un tamaño relativos al tamaño del propio control `Alignment`. Por ejemplo, puede ser útil para centrar un control dentro de la ventana.

Sólo hay dos llamadas asociadas al control `Alignment`:

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

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

La función `gtk.Alignment()` crea un nuevo control `Alignment` con los parámetros especificados. El método set() permite alterar los parámetros de alineación de un control `Alignment` existente.

Los cuatro parámetros son números en coma flotante que pueden estar entre 0.0 y 1.0. Los argumentos *xalign* y *yalign* afectan a la posición del control dentro del Alignment. Las propiedades de alineación especifican la fracción de espacio *libre* que se colocará por encima o a la izquierda del control hijo. Sus valoren van de 0.0 (sin espacio *libre* por encima o a la izquierda del hijo) a 1.0 (todo espacio *libre* o a la izquierda del hijo). Naturalmente, si las dos propiedades de escala están puestas a 1.0, entonces las propiedades de alineación no tienen efecto, puesto que el control hijo se expandirá para llenar el espacio disponible.

Los argumentos `xscale` e `yscale` especifican la fracción de espacio *libre* absorbido por el control hijo. Los valores pueden variar desde 0.0 (el hijo no absorbe nada) hasta 1.0 (el hijo toma todo el espacio *libre*.

Un control hijo puede añadirse a este `Alignment` usando:

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

Para un ejemplo del uso de un control `Alignment`, consulte el ejemplo del control de Barra de Progreso progressbar.py

10.3. Contenedor Fijo (Fixed)

El contenedor `Fixed` (Fijo) permite situar controles en una posición fija dentro de su ventana, relativa a su esquina superior izquierda. La posición de los controles se puede cambiar dinámicamente.

Sólo hay dos llamadas asociadas al control fijo:

```
fixed = gtk.Fixed()

fixed.put(widget, x, y)

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

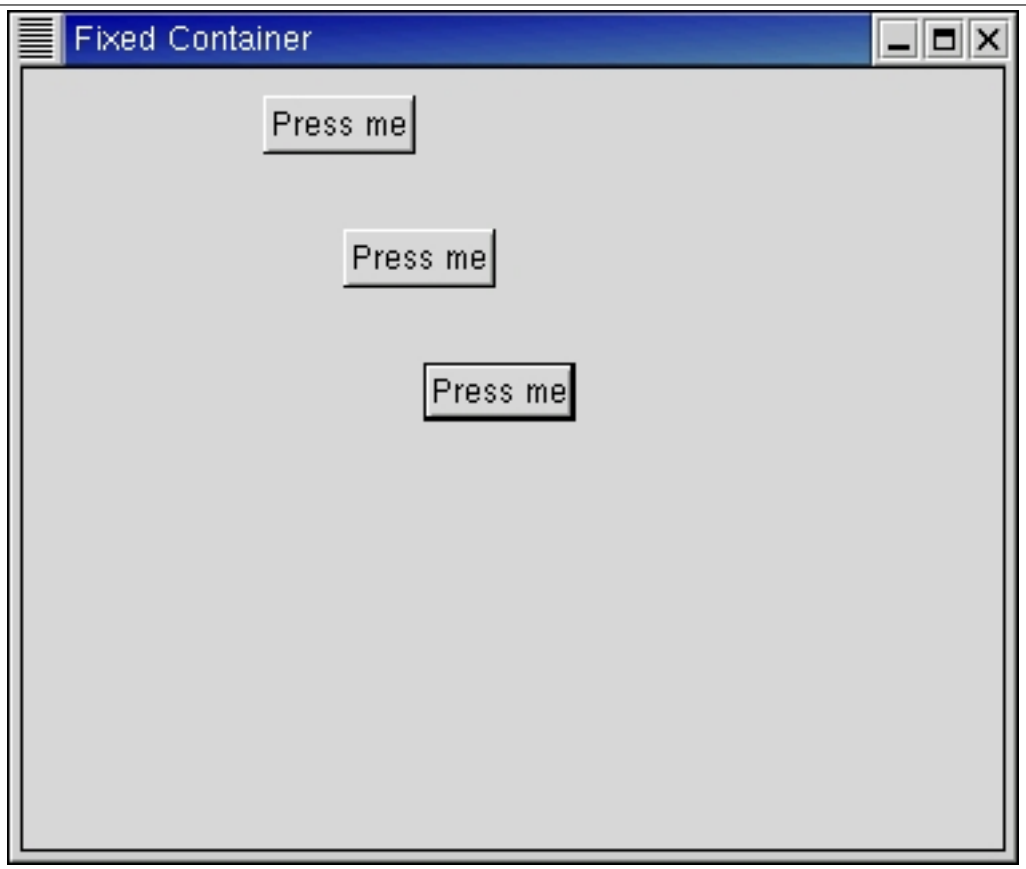
La función `gtk.Fixed()` permite crear un nuevo contenedor `Fixed`.

El método put() coloca al control en el contenedor fijo en la posición especificada por `x` e `y`.

El método move() te permite mover el control especificado a una nueva posición.

El ejemplo fixed.py ilustra cómo usar el contenedor `Fixed` . La figura Figura 10.2 muestra el resultado:

Figura 10.2 Ejemplo de Fijo



El código fuente de fixed.py es:

```
1 #!/usr/bin/env python
2
```

```
 3 # ejemplo fixed.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class FixedExample:
10     # Esta retrollamada mueve el botón a una nueva posición
11     # en el contenedor Fixed.
12     def move_button(self, widget):
13         self.x = (self.x+30)%300
14         self.y = (self.y+50)%300
15         self.fixed.move(widget, self.x, self.y)
16
17     def __init__(self):
18         self.x = 50
19         self.y = 50
20
21         # Creamos una nueva ventana
22         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
23         window.set_title("Fixed Container")
24
25         # Conectamos el evento "destroy" a un manejador de señales
26         window.connect("destroy", lambda w: gtk.main_quit())
27
28         # Fija el grosor del borde de ventana
29         window.set_border_width(10)
30
31         # Creamos un contenedor Fixed
32         self.fixed = gtk.Fixed()
33         window.add(self.fixed)
34         self.fixed.show()
35
36         for i in range(1, 4):
37             # Crea un nuevo toón con la etiqueta "Press me"
38             button = gtk.Button("Press me")
39
40             # Cuando el botón recibe la señal "clicked" llamará al
41             # método move_button().
42             button.connect("clicked", self.move_button)
43
44             # Empaquetamos el botón en la ventana del contenedor fijo
45             self.fixed.put(button, i*50, i*50)
46
47             # El paso final es mostrar el control recién creado
48             button.show()
49
50         # Mostramos la ventana
51         window.show()
52
53 def main():
54     # Entramos en el bucle de eventos
55     gtk.main()
56     return 0
57
58 if __name__ == "__main__":
59     FixedExample()
60     main()
```

10.4. Contenedor de Disposición (Layout)

El contenedor `Layout` (Disposición) es similar al contenedor `Fixed` (Fijo) excepto en que implementa un área de desplazamiento infinita (donde infinito es menor que 2^32). El sistema de ventanas X tiene una limitación que hace uqe las ventanas sólo puedan tener 32767 píxeles de ancho o alto. El contenedor `Layout` soluciona esta limitación haciendo algunos trucos exóticos mediante el uso de gravedad de ventana y de bit, para que se tenga un desplazamiento suave incluso cuando se tienen muchos controles en el área de desplazamiento.

Un contenedor `Layout` se crea con:

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

Como se puede ver, se puede especificar opcionalmente los objetos Adjustment que el control `Layout` usará para su desplazamiento. Si no se especifican los objetos Adjustment, se crearán unos nuevos.

Se pueden añadir y mover controles en el contenedor `Layout` usando los dos métodos siguientes:

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

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

El tamaño del contenedor `Layout` se puede establecer y consultar usando los siguientes métodos:

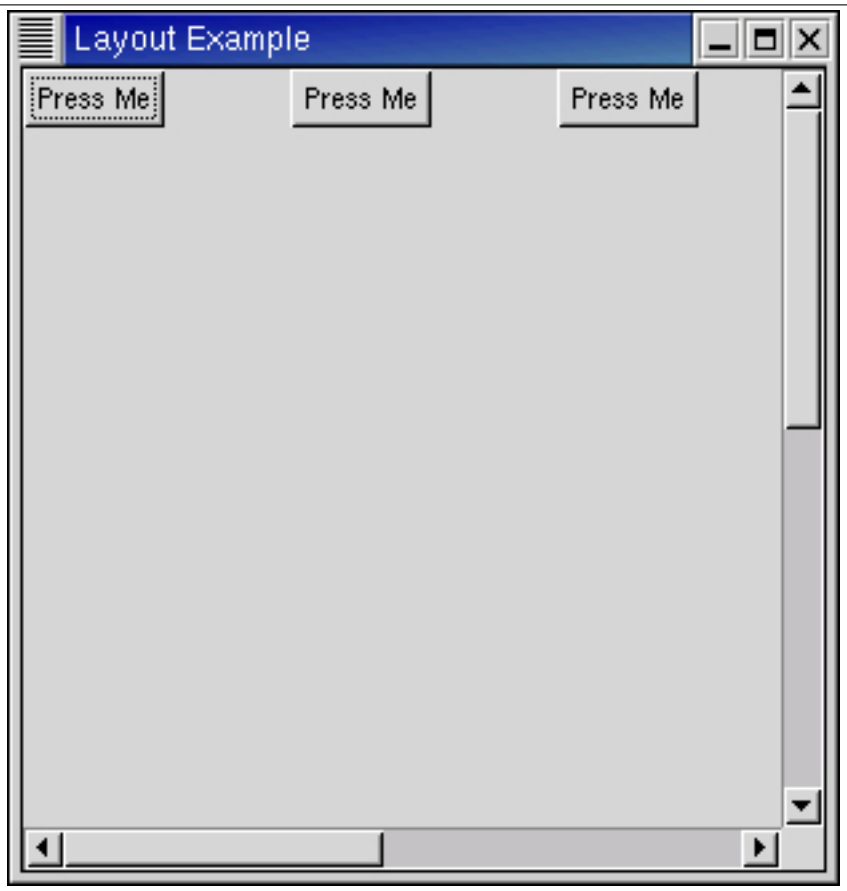
```
layout.set_size(width, height)

size = layout.get_size()
```

Los últimos cuatro métodos para los controles `Layout` widgets son para manipular los controles de ajuste horizontal y vertical:

```
hadj = layout.get_hadjustment()

vadj = layout.get_vadjustment()

layout.set_hadjustment(adjustment)

layout.set_vadjustment(adjustment)
```

El programa de ejemplo layout.py crea tres botones y los pone en un control disposición. Cuando se hace clic en un botón, se mueve a una posición aleatoria en el control disposición. La figura Figura 10.3 ilustra la ventana inicial del programa:

Figura 10.3 Ejemplo de Disposición



El código fuente de layout.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo layout.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8 import random
 9
10 class LayoutExample:
11     def WindowDeleteEvent(self, widget, event):
12         # devolvemos false para que se destruya la ventana
13         return gtk.FALSE
14
15     def WindowDestroy(self, widget, *data):
16         # salimos del bucle principal
17         gtk.main_quit()
18
19     def ButtonClicked(self, button):
20         # movemos el botón
21         self.layout.move(button, random.randint(0,500),
22                          random.randint(0,500))
23
24     def __init__(self):
25         # creamos la ventana principal
26         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
27         window.set_title("Layout Example")
28         window.set_default_size(300, 300)
29         window.connect("delete-event", self.WindowDeleteEvent)
```

```
30          window.connect("destroy", self.WindowDestroy)
31          # creamos la tabla y la empaquetamos en la ventana
32          table = gtk.Table(2, 2, gtk.FALSE)
33          window.add(table)
34          # creamos el control disposición y lo empaquetamos en la tabla
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          # creamos las barras de desplazamiento y las empaquetamos también
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          # indicamos a las barras de desp. que usen el ajuste del control ↩
    disposición
47          vAdjust = self.layout.get_vadjustment()
48          vScrollbar.set_adjustment(vAdjust)
49          hAdjust = self.layout.get_hadjustment()
50          hScrollbar.set_adjustment(hAdjust)
51          # creamos 3 botones ylos ponemos en el control disposición
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          # mostramos todos los controles
62          window.show_all()
63
64 def main():
65      # entramos en el bucle principal
66      gtk.main()
67      return 0
68
69 if __name__ == "__main__":
70      LayoutExample()
71      main()
```

10.5. Marcos (Frame)

Los Marcos se pueden usar para encerrar un widget o un grupo de ellos dentro de una caja que, opcionalmente, puede llevar un título. La posición del título y el estilo de la caja se puede alterar a gusto.

Un `Frame` (Marco) se puede crear con la siguiente función

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

El *label* (título) se coloca en la esquina superior izquierda del marco de manera predeterminada. Especificando un valor de None para el argumento *label* o sin especificar el argumento *label* hará que no se visualice ningún título. El texto del título se puede cambiar usando el método:

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

La posición del título se puede cambiar usando el método:

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

xalign y *yalign* toman valores entre 0.0 y 1.0. *xalign* indica la posición del título en la horizontal superior del marco. *yalign* no se usa por ahora. El valor por defecto de *xalign* es 0.0 lo que coloca al título en la esquina izquierda del marco.

El siguiente método modifica el estilo de la caja que se usa para rodear el marco.

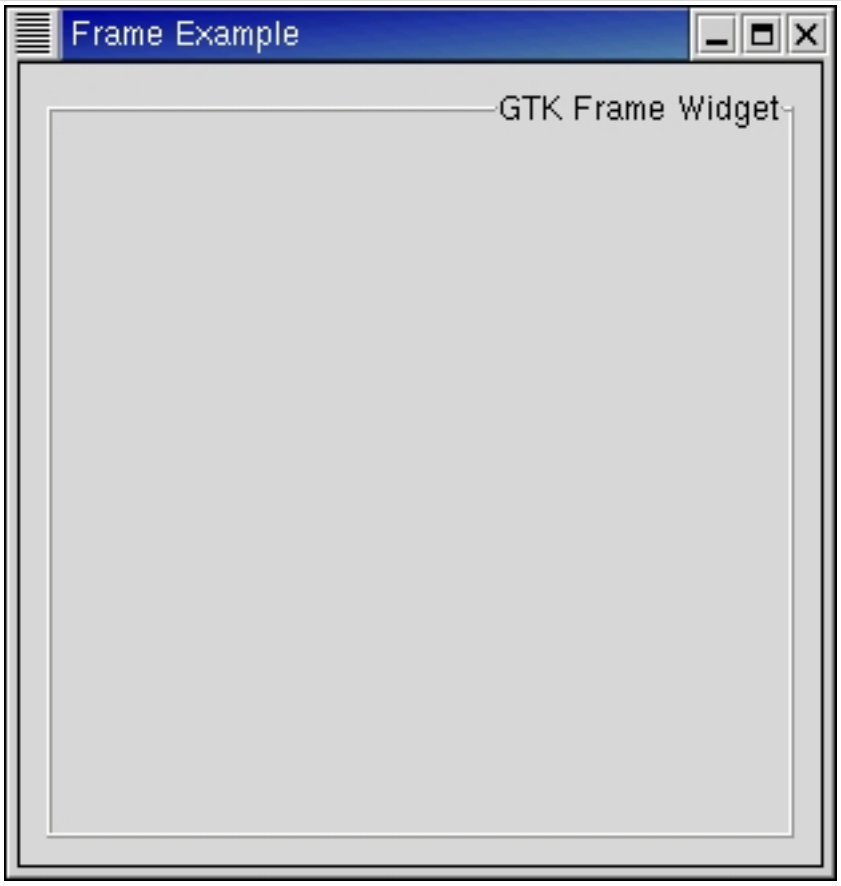
```
frame.set_shadow_type(type)
```

El argumento *type* puede tomar uno de los siguientes valores:

```
SHADOW_NONE           # sin sombra
SHADOW_IN             # sombra hacia dentro
SHADOW_OUT            # sombra hacia fuera
SHADOW_ETCHED_IN      # sombra marcada hacia dentro (valor predeterminado)
SHADOW_ETCHED_OUT     # sombra marcada hacia fuera
```

El ejemplo frame.py muestra el uso del control Marco. La figura Figura 10.4 muestra la ventana resultante:

Figura 10.4 Ejemplo de Marco



EL código fuente de frame.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo frame.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class FrameExample:
10     def __init__(self):
11         # Creamos una ventana nueva
12         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

```
13          window.set_title("Frame Example")
14
15          # Conectamos el evento "destroy" al manejador de señal
16          window.connect("destroy", lambda w: gtk.main_quit())
17          window.set_size_request(300, 300)
18
19          # Fijamos el grosor del borde de ventana
20          window.set_border_width(10)
21
22          # Creamos un Marco
23          frame = gtk.Frame()
24          window.add(frame)
25
26          # Fijamos la etiqueta del marco
27          frame.set_label("GTK Frame Widget")
28
29          # Alineamos la etiqueta a la derecha del marco
30          frame.set_label_align(1.0, 0.0)
31
32          # Fijamos el estilo del marco
33          frame.set_shadow_type(gtk.SHADOW_ETCHED_OUT)
34          frame.show()
35
36          # Mostramos la ventana
37          window.show()
38
39 def main():
40      # Entramos en el bucle de eventos
41      gtk.main()
42      return 0
43
44 if __name__ == "__main__":
45      FrameExample()
46      main()
```

Los programas calendar.py, label.py y spinbutton.py también usan Marcos.

10.6. Marcos Proporcionales (AspectFrame)

El control de marco proporcional es como un control de marco, excepto en que también mantiene el cociente de proporcionalidad (esto es, el cociente del ancho entre el alto) del control hijo en un determinado valor, añadiendo espacio extra si es necesario. Esto es útil, por ejemplo, si se quiere previsualizar una imagen más grande. El tamaño de la previsualización debería variar cuando el usuario redimensione la ventana, pero el cociente de proporcionalidad debe siempre corresponderse al de la imagen original.

Para crear un nuevo marco proporcional se utiliza:

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

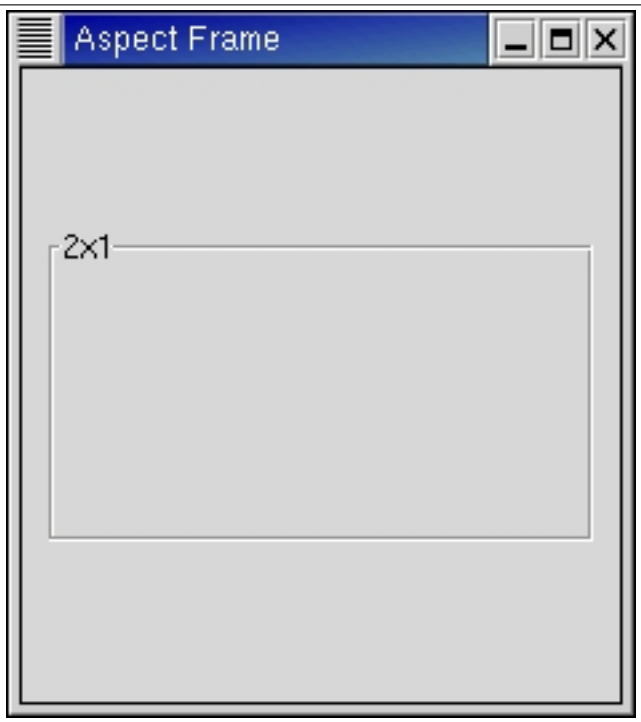
label especifica el texto a mostrar en el título. *xalign* e *yalign* especifican la alineación como en el control Alignment. Si *obey_child* es TRUE, el cociente de proporcionalidad de un control hijo se corresponderá con el cociente de proporcionalidad del tamaño ideal que solicite. En caso contrario, se especifica dicho cociente con el argumento *ratio*.

Para cambiar las opciones de un marco proporcional existente, se puede utilizar:

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

Como ejemplo, el programa aspectframe.py usa un `AspectFrame` para presentar un área de dibujo cuyo cociente de proporcionalidad será siempre 2:1, aunque el usuario redimensione la ventana contenedora. La figura Figura 10.5 muestra la ventana del programa:

Figura 10.5 Ejemplo de Marco Proporcional



El código fuente del programa aspectframe.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo aspectframe.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class AspectFrameExample:
10     def __init__(self):
11         window = gtk.Window(gtk.WINDOW_TOPLEVEL);
12         window.set_title("Aspect Frame")
13         window.connect("destroy", lambda x: gtk.main_quit())
14         window.set_border_width(10)
15
16         # Creamos un marco proporcional aspect_frame y lo añadimos a la ventana
17         aspect_frame = gtk.AspectFrame("2x1", # label
18                                        0.5, # center x
19                                        0.5, # center y
20                                        2, # xsize/ysize = 2
21                                        gtk.FALSE) # ignore child's aspect
22         window.add(aspect_frame)
23         aspect_frame.show()
24
25         # Añadimos un control hijo al marco proporcional
26         drawing_area = gtk.DrawingArea()
27
28         # Solicitamos una ventana 200x200, pero AspectFrame nos devolverá una 200x100
29         # puesto que forzamos una proporción 2x1
30         drawing_area.set_size_request(200, 200)
31         aspect_frame.add(drawing_area)
32         drawing_area.show()
33         window.show()
```

```
34
35 def main():
36     gtk.main()
37     return 0
38
39 if __name__ == "__main__":
40     AspectFrameExample()
41     main()
```

10.7. Controles de Panel (HPaned y VPaned)

Los controles de paneles son útiles cuando se quiere dividir un área en dos partes, con el tamaño relativo de las dos partes controlado por el usuario. Se dibuja una barra entre las dos partes con un mango que el usuario puede arrastrar para cambiar la relación. La división puede ser horizontal (`HPaned`) o vertical (`VPaned`).

Para crear una nueva ventana con panel, se hace una llamada a:

```
hpane = gtk.HPaned()
o
vpane = gtk.VPaned()
```

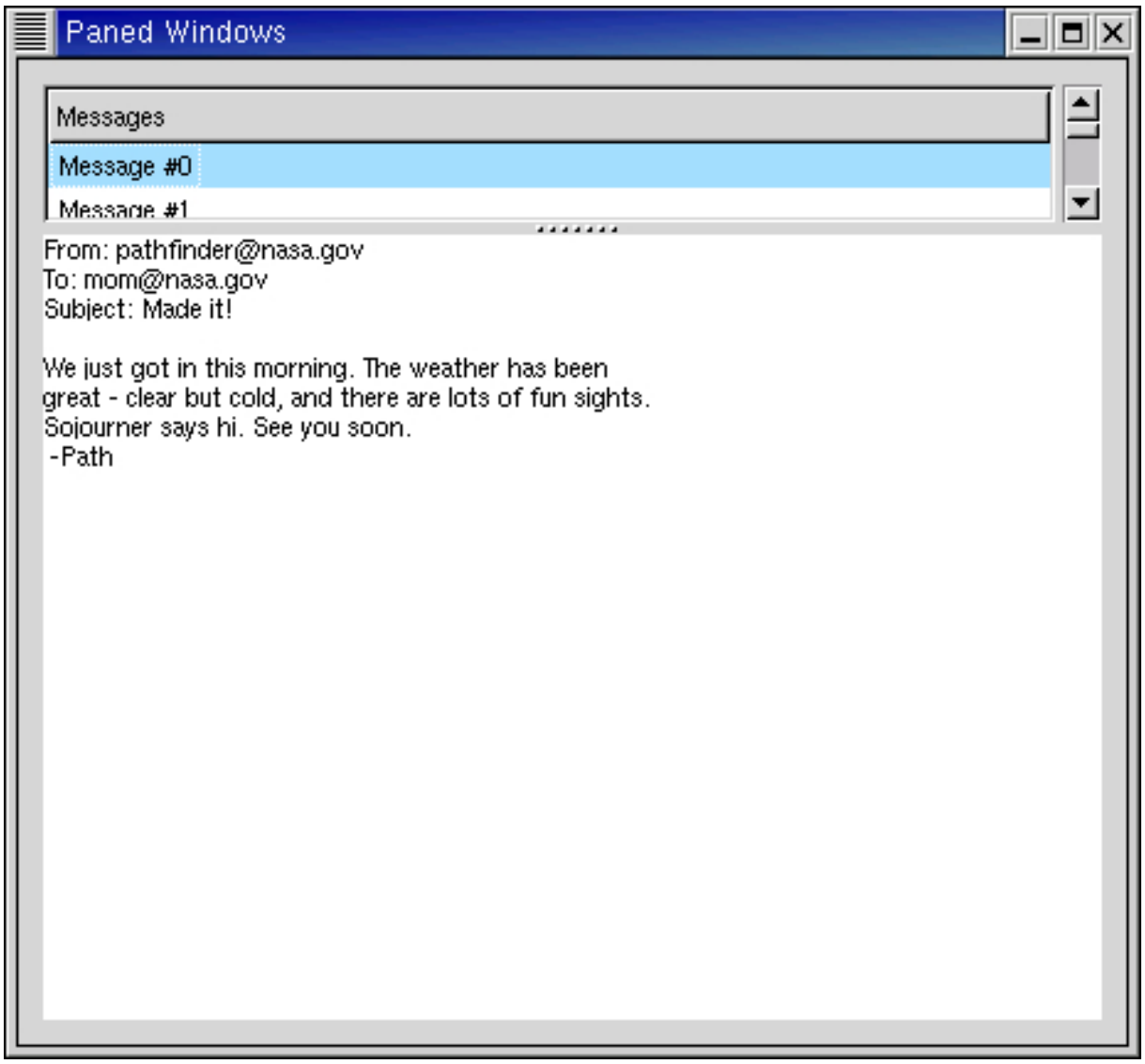
Después de crear el control de panel, hay que añadir hijos a sus dos mitades. Para hacer eso, se usan los métodos:

```
paned.add1(child)

paned.add2(child)
```

El método add1() añade el control hijo *child* a la izquierda o arriba del control de panel. El método add2() añade un control hijo *child* a la derecha o abajo del panel.

El programa de ejemplo paned.py crea parte de la interfaz de usuario de un programa de correo electrónico imaginario. Una ventana se divide en dos partes verticalmente, donde la parte de arriba es una lista de correos electrónicos y la parte de abajo es el texto del mensaje electrónico. La mayoría del programa es bastante sencillo. Hay un par de puntos en los que hacer hincapie: no se puede añadir texto a un control Text hasta que se realiza. Esto podría conseguirse llamando al método realize() , pero como demostración de una técnica alternativa, conectamos un manejador para la señal "realize" para añadir el texto. Además, necesitamos añadir la opción `SHRINK` (encoger) a algunos de los elementos de la tabla que contiene la ventana de texto y sus barras de desplazamiento, para que cuando la parte de abajo se haga más pequeña, las secciones correctas se encojan en vez de que desaparezcan por la parte de abajo de la ventana. La figura Figura 10.6 muestra el resultado de ejecutar el programa:

Figura 10.6 Ejemplo de Panel



El código fuente del programa paned.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo paned.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk, gobject
 8
 9 class PanedExample:
10     # Creamos la lista de "mensajes"
11     def create_list(self):
12         # Creamos una ventana, con desplazamiento si es necesario
13         scrolled_window = gtk.ScrolledWindow()
14         scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk. ↩
   POLICY_AUTOMATIC)
15
16         model = gtk.ListStore(gobject.TYPE_STRING)
17         tree_view = gtk.TreeView(model)
18         scrolled_window.add_with_viewport (tree_view)
19         tree_view.show()
20
```

```
21          # Añadimos algunos mensajes a la ventana
22          for i in range(10):
23              msg = "Message #%d" % i
24              iter = model.append()
25              model.set(iter, 0, msg)
26
27          cell = gtk.CellRendererText()
28          column = gtk.TreeViewColumn("Messages", cell, text=0)
29          tree_view.append_column(column)
30
31          return scrolled_window
32
33      # Añadimos texto a nuestro control de texto - esta retrollamada se ↩
    invoca
34      # cuando se realiza nuestra ventana. Podríamos forzar también la ↩
    realización
35      # mediante gtk.Widget.realize(), pero primero tendría que ser parte
36      # de una jerarquía
37      def insert_text(self, buffer):
38          iter = buffer.get_iter_at_offset(0)
39          buffer.insert(iter,
40                        "From: pathfinder@nasa.gov\n"
41                        "To: mom@nasa.gov\n"
42                        "Subject: Made it!\n"
43                        "\n"
44                        "We just got in this morning. The weather has been\n"
45                        "great - clear but cold, and there are lots of fun ↩
    sights.\n"
46                        "Sojourner says hi. See you soon.\n"
47                        " -Path\n")
48
49      # Creamos un área de texto desplazable que muestra un "mensaje"
50      def create_text(self):
51          view = gtk.TextView()
52          buffer = view.get_buffer()
53          scrolled_window = gtk.ScrolledWindow()
54          scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk. ↩
    POLICY_AUTOMATIC)
55          scrolled_window.add(view)
56          self.insert_text(buffer)
57          scrolled_window.show_all()
58          return scrolled_window
59
60      def __init__(self):
61          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
62          window.set_title("Paned Windows")
63          window.connect("destroy", lambda w: gtk.main_quit())
64          window.set_border_width(10)
65          window.set_size_request(450, 400)
66
67          # creamos un control vpaned y lo añadimos a la ventana principal
68          vpaned = gtk.VPaned()
69          window.add(vpaned)
70          vpaned.show()
71
72          # Ahora creamos los contenidos de las dos partes de la ventana
73          list = self.create_list()
74          vpaned.add1(list)
75          list.show()
76
77          text = self.create_text()
78          vpaned.add2(text)
79          text.show()
80          window.show()
```

```
81
82 def main():
83     gtk.main()
84     return 0
85
86 if __name__ == "__main__":
87     PanedExample()
88     main()
```

10.8. Vistas (Viewport)

Es poco probable que necesites usar el control `Viewport` (Vista) directamente. Es mucho más probable que uses el control ScrolledWindow, el cual usa un `Viewport`.

Un control de vista permite colocar un control más grande dentro de él de tal forma que se pueda ver una parte de él de una vez. Usa Adjustments para definir el área que se ve.

Un `Viewport` se crea con la función:

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

Como se puede ver se pueden especificar los Adjustments horizontal y vertical que el control usa cuando se crea. Creará sus propios ajustes si se le pasa `None` como valor de los argumentos o simplemente no se le pasan argumentos.

Se pueden consultar y fijar los ajustes después de que el control se haya creado usando los siguientes cuatro métodos:

```
viewport.get_hadjustment()

viewport.get_vadjustment()

viewport.set_hadjustment(adjustment)

viewport.set_vadjustment(adjustment)
```

El otro método que se usa para modificar la apariencia es:

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

Los valores posibles para el parámetro *`type`* son:

```
SHADOW_NONE         # sin sombra
SHADOW_IN           # sombra hacia adentro
SHADOW_OUT          # sombra hacia afuera
SHADOW_ETCHED_IN    # sombra marcada hacia adentro
SHADOW_ETCHED_OUT   # sombra marcada hacia fuera
```

10.9. Ventanas de Desplazamiento (ScrolledWindow)

Las ventanas de desplazamiento se usan para crear un área de desplazamiento con otro control dentro de ella. Se puede insertar cualquier tipo de control dentro de una ventana de desplazamiento, y será accesible, independientemente del tamaño, usando barras de desplazamiento.

La siguiente función se usa para crear una nueva ventana de desplazamiento.

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

Donde el primer argumento es el ajuste para la dirección horizontal, y el segundo, el ajuste para la dirección vertical. Casi siempre se ponen a `None` o no se especifican.

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

Este método especifica la política a usar con respecto a las barras de desplazamiento. El primer argumento le fija la política a la barra de desplazamiento horizontal, y el segundo, la política de la barra de desplazamiento vertical.

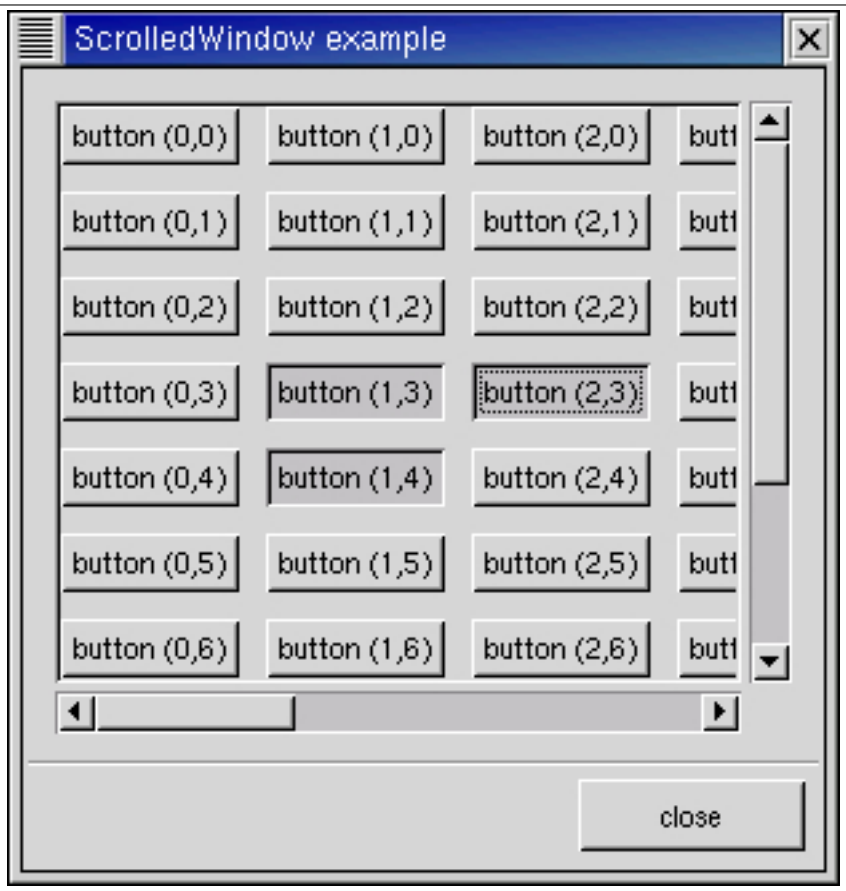
La política puede tomar los valores `POLICY_AUTOMATIC` o `POLICY_ALWAYS`. `POLICY_AUTOMATIC` decidirá automáticamente si son necesarias las barras de desplazamiento, mientras que `POLICY_ALWAYS` siempre dejará visibles las barras de desplazamiento.

Tras esto se puede colocar el objeto dentro de la ventana de desplazamiento usando el siguiente método.

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

El programa de ejemplo scrolledwin.py coloca una tabla con 100 botones biestado dentro de una ventana de desplazamiento. Sólo se comentan las partes que pueden ser nuevas. La figura Figura 10.7 muestra la ventana del programa:

Figura 10.7 Ejemplo de Ventana de Desplazamiento



El código fuente de scrolledwin.py es:

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

# ejemplo scrolledwin.py

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

class ScrolledWindowExample:
    def destroy(self, widget):
        gtk.main_quit()

    def __init__(self):
        # Creamos una nueva ventana de diálogo en la que empaquetar
        # la ventana de desplazamiento
        window = gtk.Dialog()
        window.connect("destroy", self.destroy)
```

```
18          window.set_title("ScrolledWindow example")
19          window.set_border_width(0)
20          window.set_size_request(300, 300)
21
22          # creamos una nueva ventana de desplazamiento
23          scrolled_window = gtk.ScrolledWindow()
24          scrolled_window.set_border_width(10)
25
26          # la política es bien POLICY AUTOMATIC, o bien POLICY_ALWAYS.
27          # POLICY_AUTOMATIC decidirá automáticamente la necesidad de barras ↩
    de
28          # desplazamiento, mientras que POLICY_ALWAYS dejará permanentemente ↩
     las
29          # barras. La primera es la barra de desplazamiento horizontal, la ↩
    segunda la
30          # vertical.
31          scrolled_window.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_ALWAYS)
32
33          # Se crea la ventana de diálogo con una vbox en ella
34          window.vbox.pack_start(scrolled_window, gtk.TRUE, gtk.TRUE, 0)
35          scrolled_window.show()
36
37          # creamos una tabla de 10 por 10 casillas
38          table = gtk.Table(10, 10, gtk.FALSE)
39
40          # fijamos el espaciado a 10 en x y 10 en y
41          table.set_row_spacings(10)
42          table.set_col_spacings(10)
43
44          # empaquetamos la tabla en la ventana de desplazamiento
45          scrolled_window.add_with_viewport(table)
46          table.show()
47
48          # esto simplemente crea una trama de botones biestado en la tabla
49          # para demostrar el uso de la ventana de desplazamiento
50          for i in range(10):
51              for j in range(10):
52                  buffer = "button (%d,%d)" % (i, j)
53                  button = gtk.ToggleButton(buffer)
54                  table.attach(button, i, i+1, j, j+1)
55                  button.show()
56
57          # Añadimos un botón "close" en la parte inferior del diálogo
58          button = gtk.Button("close")
59          button.connect_object("clicked", self.destroy, window)
60
61          # ponemos el botón como posible predeterminado
62          button.set_flags(gtk.CAN_DEFAULT)
63          window.action_area.pack_start( button, gtk.TRUE, gtk.TRUE, 0)
64
65          # Esto lo establece como predeterminado. Pulsando
66          # "Enter" hará que se active este botón
67          button.grab_default()
68          button.show()
69          window.show()
70
71 def main():
72      gtk.main()
73      return 0
74
75 if __name__ == "__main__":
76      ScrolledWindowExample()
77      main()
```

Si se prueba a cambiar el tamaño a la ventana se verá cómo reaccionan las barras de desplazamiento. Puede que también se quiera usar el método set_size_request() para fijar el tamaño por defecto de la ventana o de otros controles.

10.10. Cajas de Botones (ButtonBoxes)

Las `ButtonBoxes` (Cajas de Botones) proporcionan un sistema fácil de agrupar botones rápidamente. Vienen en variedades horizontales y verticales. Se puede crear una nueva `ButtonBox` con una de las siguiente llamadas, que crean una caja horizontal o vertical, respectivamente:

```
hbutton_box = gtk.HButtonBox()

vbutton_box = gtk.VButtonBox()
```

Los únicos métodos de las cajas de botones afectan a la disposición de los botones.

La disposición de los botones dentro de la caja se establece usando:

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

El argumento *`layout_style`* puede tomar uno de los siguientes valores:

```
BUTTONBOX_DEFAULT_STYLE # estilo predeterminado
BUTTONBOX_SPREAD        # esparcidos
BUTTONBOX_EDGE          # al borde
BUTTONBOX_START         # al principio
BUTTONBOX_END           # al final
```

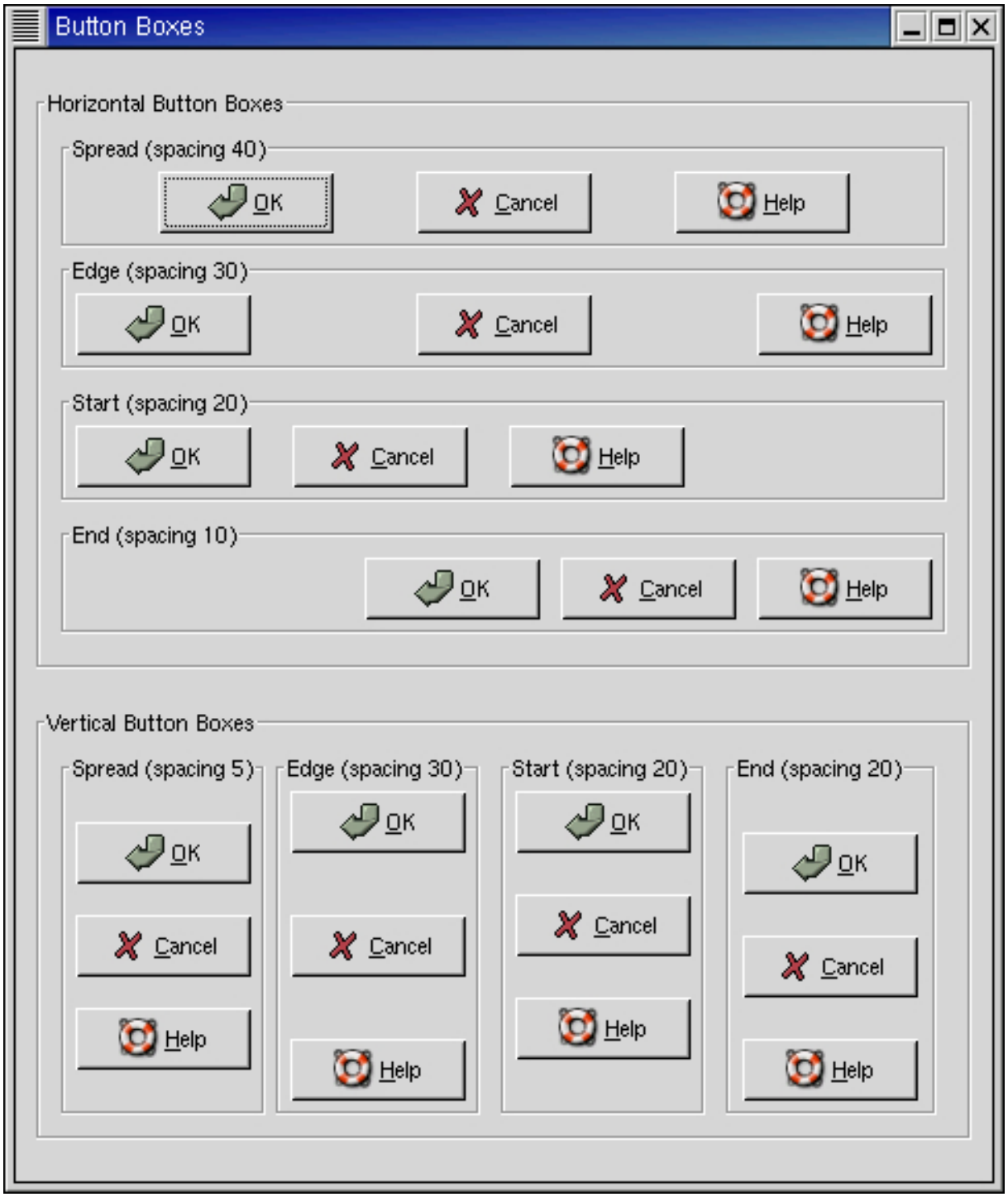
El valor actual de *`layout_style`* se puede consultar usando:

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

Los botones se añaden a la `ButtonBox` usando el típico método de `Container` (Contenedor):

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

El programa de ejemplo buttonbox.py ilustra todos los tipos de disposición de las `ButtonBoxes`. La ventana resultante es:



El código fuente del programa buttonbox.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo buttonbox.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ButtonBoxExample:
10     # Creamos una Button Box con los parámetros especificados
11     def create_bbox(self, horizontal, title, spacing, layout):
12         frame = gtk.Frame(title)
13
```

```
14          if horizontal:
15              bbox = gtk.HButtonBox()
16          else:
17              bbox = gtk.VButtonBox()
18
19          bbox.set_border_width(5)
20          frame.add(bbox)
21
22          # Fijamos el aspecto de Button Box
23          bbox.set_layout(layout)
24          bbox.set_spacing(spacing)
25
26          button = gtk.Button(stock=gtk.STOCK_OK)
27          bbox.add(button)
28
29          button = gtk.Button(stock=gtk.STOCK_CANCEL)
30          bbox.add(button)
31
32          button = gtk.Button(stock=gtk.STOCK_HELP)
33          bbox.add(button)
34
35          return frame
36
37      def __init__(self):
38          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
39          window.set_title("Button Boxes")
40
41          window.connect("destroy", lambda x: gtk.main_quit())
42
43          window.set_border_width(10)
44
45          main_vbox = gtk.VBox(gtk.FALSE, 0)
46          window.add(main_vbox)
47
48          frame_horz = gtk.Frame("Horizontal Button Boxes")
49          main_vbox.pack_start(frame_horz, gtk.TRUE, gtk.TRUE, 10)
50
51          vbox = gtk.VBox(gtk.FALSE, 0)
52          vbox.set_border_width(10)
53          frame_horz.add(vbox)
54
55          vbox.pack_start(self.create_bbox(gtk.TRUE, "Spread (spacing 40)",
56                                           40, gtk.BUTTONBOX_SPREAD),
57                          gtk.TRUE, gtk.TRUE, 0)
58
59          vbox.pack_start(self.create_bbox(gtk.TRUE, "Edge (spacing 30)",
60                                           30, gtk.BUTTONBOX_EDGE),
61                          gtk.TRUE, gtk.TRUE, 5)
62
63          vbox.pack_start(self.create_bbox(gtk.TRUE, "Start (spacing 20)",
64                                           20, gtk.BUTTONBOX_START),
65                          gtk.TRUE, gtk.TRUE, 5)
66
67          vbox.pack_start(self.create_bbox(gtk.TRUE, "End (spacing 10)",
68                                           10, gtk.BUTTONBOX_END),
69                          gtk.TRUE, gtk.TRUE, 5)
70
71          frame_vert = gtk.Frame("Vertical Button Boxes")
72          main_vbox.pack_start(frame_vert, gtk.TRUE, gtk.TRUE, 10)
73
74          hbox = gtk.HBox(gtk.FALSE, 0)
75          hbox.set_border_width(10)
76          frame_vert.add(hbox)
77
```

```
 78         hbox.pack_start(self.create_bbox(gtk.FALSE, "Spread (spacing 5)",
 79                                          5, gtk.BUTTONBOX_SPREAD),
 80                         gtk.TRUE, gtk.TRUE, 0)
 81
 82         hbox.pack_start(self.create_bbox(gtk.FALSE, "Edge (spacing 30)",
 83                                          30, gtk.BUTTONBOX_EDGE),
 84                         gtk.TRUE, gtk.TRUE, 5)
 85
 86         hbox.pack_start(self.create_bbox(gtk.FALSE, "Start (spacing 20)",
 87                                          20, gtk.BUTTONBOX_START),
 88                         gtk.TRUE, gtk.TRUE, 5)
 89
 90         hbox.pack_start(self.create_bbox(gtk.FALSE, "End (spacing 20)",
 91                                          20, gtk.BUTTONBOX_END),
 92                         gtk.TRUE, gtk.TRUE, 5)
 93
 94         window.show_all()
 95
 96 def main():
 97     # Entramos en el bucle de eventos
 98     gtk.main()
 99     return 0
100
101 if __name__ == "__main__":
102     ButtonBoxExample()
103     main()
```

10.11. Barra de Herramientas (Toolbar)

Las `Toolbars` (Barras de Herramientas) se usan normalmente para agrupar un número de controles y simplificar la personalización de su apariencia y disposición. Típicamente una barra de herramientas consiste en botones con iconos, etiquetas y pistas, pero se puede poner cualquier otro control en una barra de herramientas. Finalmente, los elementos se pueden colocar horizontal o verticalmente y se pueden ver los botones con iconos, texto, o ambos.

La creación de una barra de herramientas se consigue (como uno podría sospechar) con la siguiente función:

```
toolbar = gtk.Toolbar()
```

Después de crear una barra de herramientas uno puede añadir al principio, al final o en otro posición, items (textos simples) o elementos (cualquier tipo de control) en la barra de herramientas. Para describir un item necesitamos un texto, un texto para la pista, un texto privado para la pista, un icono para el botón y una retrollamada. Por ejemplo, para añadir al principio o al final un item se pueden usar los siguientes métodos:

```
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)
```

Si se quiere usar el método insert_item(), el único parámetro adicional que debe especificarse, es la posición en la cual el elemento se debe insertar. Así:

```
toolbar.insert_item(text, tooltip_text, tooltip_private_text, icon, callback,
                    user_data, position)
```

Para añadir de forma sencilla espacios entre items de la barra de herramientas se pueden usar los siguientes métodos:

```
toolbar.append_space()

toolbar.prepend_space()
```

```
toolbar.insert_space(position)
```

Si es necesario, la orientación de una barra de herramientas, su estilo y el hecho de que las pistas estén disponibles, se puede cambiar sobre la marcha usando los siguientes métodos:

```
toolbar.set_orientation(orientation)

toolbar.set_style(style)

toolbar.set_tooltips(enable)
```

Donde la `orientation` puede ser `ORIENTATION_HORIZONTAL` u `ORIENTATION_VERTICAL`. El *`style`* se usa para especificar la apariencia de la barra de herramientas y puede ser `TOOLBAR_ICONS` (iconos), `TOOLBAR_TEXT` (texto), o `TOOLBAR_BOTH` (ambos). El argumento *`enable`* puede ser o `TRUE` o `FALSE`.

Para mostrar algunas otras cosas que se pueden hacer con una barra de herramientas veamos el siguiente programa de ejemplo toolbar.py (interrumpiremos el listado con explicaciones adicionales):

```
 1  #!/usr/bin/env python
 2
 3  # ejemplo toolbar.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class ToolbarExample:
10      # Este método se conecta al botón Close o
11      # al cierre de la ventana desde el Gestor de Ventanas
12      def delete_event(self, widget, event=None):
13          gtk.main_quit()
14          return gtk.FALSE
15
```

Este principio debería resultar familiar si no es el primer programa PyGTK. No obstante hay una cosa más, vamos a importar un bonito dibujo XPM (gtk.xpm) para que nos sirva de icono para todos los botones. La linea 10 empieza la clase de ejemplo `ToolbarExample` y las lineas 12-14 definen el método de retrollamada que finalizará el programa.

```
16      # es fácil... cuando se conmuta uno de los botones, simplemente
17      # comprobamos cuál está activo y cambiamos el estilo de la barra de ↩
    herramientas
18      # de acuerdo con ello
19        def radio_event(self, widget, toolbar):
20            if self.text_button.get_active():
21                toolbar.set_style(gtk.TOOLBAR_TEXT)
22            elif self.icon_button.get_active():
23                toolbar.set_style(gtk.TOOLBAR_ICONS)
24            elif self.both_button.get_active():
25                toolbar.set_style(gtk.TOOLBAR_BOTH)
26
27      # incluso más fácil, comprobamos la conmutación del botón
28      # y activamos/desactivamos las pistas
29      def toggle_event(self, widget, toolbar):
30          toolbar.set_tooltips(widget.get_active())
31
```

Las lineas 19-30 son dos métodos de retrollamada que se llamarán cuando uno de los botones de la barra de herramientas se active. Estas cosas resultan familiares si ya se han usado botones biestado (y botones de exclusión mútua).

```
32        def __init__(self):
33            # Aquí está la ventana principal (un diálogo) y el asa
34            # necesitamos una barra de herramientas, un icono con máscara ( ↩
    uno para todos
```

```
35          # los botones) y un control de icono en el que ponerlo (aunque ↩
    crearemos
36          # un control aparte para cada botón)
37          # creamos una nueva ventana con su título y un tamaño adecuado
38          dialog = gtk.Dialog()
39          dialog.set_title("GTKToolbar Tutorial")
40          dialog.set_size_request(450, 250)
41          dialog.set_resizable(gtk.TRUE)
42
43          # generalmente querremos salir si cierran la ventana
44          dialog.connect("delete_event", self.delete_event)
45
46          # para que tenga mejor aspecto ponemos la barra en la caja con ↩
    asa
47          # de manera que se pueda separar de la ventana principal
48          handlebox = gtk.HandleBox()
49          dialog.vbox.pack_start(handlebox, gtk.FALSE, gtk.FALSE, 5)
50
```

Lo anterior debería ser común a cualquier otra aplicación PyGTK. Inicialización de una instancia de `ToolbarExample`, creación de la ventana, etc. Sólo hay una cosa que probablemente necesite más explicación: una caja con mango (asa) (HandleBox). Una caja con mango es simplemente otra caja cualquiera que puede usarse para meter controles dentro. La diferencia entre ella y las cajas típicas es que puede despegarse de la ventana padre (o, de hecho, la caja con mango permanece en el padre, pero se reduce a un rectángulo muy pequeño, mientras que todos sus contenidos cambian de padre a una nueva ventana flotante). Normalmente es bueno tener una barra de herramientas separable, por lo que estos dos controles suelen ir juntos normalmente.

```
51          # la barra de herramientas será horizontal, con iconos y texto, y
52          # con espacios de 5pxl entre elementos y, finalmente,
53          # la insertaremos en la caja con asa
54          toolbar = gtk.Toolbar()
55          toolbar.set_orientation(gtk.ORIENTATION_HORIZONTAL)
56          toolbar.set_style(gtk.TOOLBAR_BOTH)
57          toolbar.set_border_width(5)
58          handlebox.add(toolbar)
59
```

Bien, lo que hacemos arriba es una simple inicialización del control de la barra de herramientas.

```
60          # nuestro primer elemento es el botón <close>
61          iconw = gtk.Image() # control de icono
62          iconw.set_from_file("gtk.xpm")
63          close_button = toolbar.append_item(
64              "Close",           # etiqueta de botón
65              "Closes this app", # la pista del botón
66              "Private",         # información privada de la pista
67              iconw,             # control icono
68              self.delete_event) # una señal
69          toolbar.append_space() # espacio tras el elemento
70
```

En el código anterior se puede ver el caso más simple: añadir un botón a la barra de herramientas. Justo antes de añadir el nuevo elemento, tenemos que construir un control de imágen que servirá como icono para este elemento; este paso tiene que repetirse para cada nuevo elemento. Justo después del elemento añadimos también espacio, para que los elementos siguientes no se toquen unos a otros. Como se puede ver, el método append_item() devuelve una referencia al control de botón recién creado, para que se pueda trabajar con él de la forma habitual.

```
71          # ahora, hagamos el grupo de botones de exclusión...
72          iconw = gtk.Image() # control de icono
73          iconw.set_from_file("gtk.xpm")
74          icon_button = toolbar.append_element(
75              gtk.TOOLBAR_CHILD_RADIOBUTTON, # type of element
76              None,                          # control
```

```
77                "Icon",                           # etiqueta
78                "Only icons in toolbar",          # pista
79                "Private",                        # cadena privada de pista
80                iconw,                            # icono
81                self.radio_event,                 # señal
82                toolbar)                          # datos para la señal
83            toolbar.append_space()
84            self.icon_button = icon_button
85
```

Aqui empezamos a crear un grupo de botones de exclusión mútua. Para hacer esto usamos el método append_element() . De hecho, usando este método, uno puede también puede añadir items normales o incluso espacios (*type* = `gtk.TOOLBAR_CHILD_SPACE` (espacio) o `gtk.TOOLBAR_CHILD_BUTTON` (botón)). En el ejemplo anterior hemos empezado a crear un grupo de botones de exclusión mútua. Cuando se crean otros botones de exclusión mútua se necesita una referencia al botón anterior en el grupo, para que se pueda construir una lista de botones fácilmente (mira la sección RadioButtons de este tutorial). También tenemos una referencia al botón en la instancia de `ToolbarExample` para acceder a él más tare.

```
86            # los siguientes botones de exclusión se refieren a los ↩
    anteriores
87            iconw = gtk.Image() # control de icono
88            iconw.set_from_file("gtk.xpm")
89            text_button = toolbar.append_element(
90                gtk.TOOLBAR_CHILD_RADIOBUTTON,
91                icon_button,
92                "Text",
93                "Only texts in toolbar",
94                "Private",
95                iconw,
96                self.radio_event,
97                toolbar)
98            toolbar.append_space()
99            self.text_button = text_button
100
101           iconw = gtk.Image() # control de icono
102           iconw.set_from_file("gtk.xpm")
103           both_button = toolbar.append_element(
104               gtk.TOOLBAR_CHILD_RADIOBUTTON,
105               text_button,
106               "Both",
107               "Icons and text in toolbar",
108               "Private",
109               iconw,
110               self.radio_event,
111               toolbar)
112           toolbar.append_space()
113           self.both_button = both_button
114           both_button.set_active(gtk.TRUE)
115
```

Creamos otros botones de exclusión mútua de la misma forma excepto que le pasamos uno de los botones creados al método append_element() para especificar el grupo.

Al final tenemos que establecer el estado de uno de los botones manualmente (si no todos tienen el estado activo, impidiéndonos que los vayamos intercambiando).

```
116           # aquí ponemos simplemente un botón biestado
117           iconw = gtk.Image() # control de icono
118           iconw.set_from_file("gtk.xpm")
119           tooltips_button = toolbar.append_element(
120               gtk.TOOLBAR_CHILD_TOGGLEBUTTON,
121               None,
122               "Tooltips",
123               "Toolbar with or without tips",
```

```
124                 "Private",
125                 iconw,
126                 self.toggle_event,
127                 toolbar)
128             toolbar.append_space()
129             tooltips_button.set_active(gtk.TRUE)
130
```

Un botón biestado se puede crear de la forma obvia (si ya se han creado botones de exclusión mútua alguna vez).

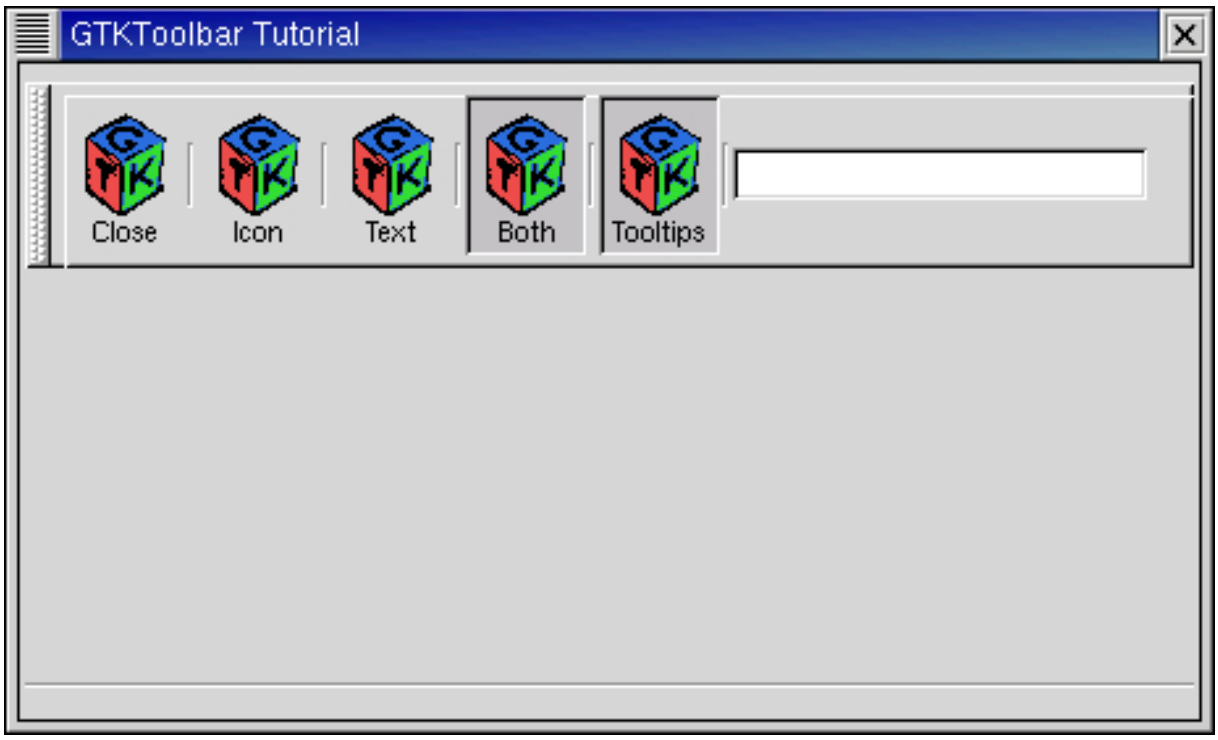
```
131             # para empaquetar un control en la barra, solamente tenemos
132             # crearlo y añadirlo con una pista apropiada
133             entry = gtk.Entry()
134             toolbar.append_widget(entry,  "This is just an entry", "Private")
135
136             # bien, no está creada dentro de la barra, así que tenemos que  ↩
    mostrarla
137             entry.show()
138
```

Como se puede ver, añadir cualquier tipo de control a una barra de herramientas es fácil. La única cosa que se tiene que recordar es que este control debe mostrarse manualmente (al contrario que los elementos, que se mostrarán de golpe con la barra de herramientas).

```
139             # ¡ya está! Mostremos todo.
140             toolbar.show()
141             handlebox.show()
142             dialog.show()
143
144     def main():
145         # nos quedamos en gtk_main y ¡esperamos que empiece la diversión!
146         gtk.main()
147         return 0
148
149     if __name__ == "__main__":
150         ToolbarExample()
151         main()
```

La linea 142 termina la definición de la clase `ToolbarExample`. Las lineas 144-147 definen la función `main`() que simplemente llama a la función `gtk.main`() para iniciar el bucle de procesamiento de eventos. Las lineas 149-151 crean una instancia de `ToolbarExample` y luego entran en el bucle de procesamiento de eventos. Así termina el tutorial de la barra de herramientas. Por supuesto, para apreciarla en su máximo esplendor se necesita también el icono XPM, gtk.xpm. La figura Figura 10.8 muestar la ventana resultante:

Figura 10.8 Ejemplo de Barra de Herramientas



10.12. Fichas (Notebook)

El control `NoteBook` (Fichas) es una colección de "páginas" que se solapan unas con otras. Cada página contiene información diferente y sólo una es visible a un tiempo. Este control se ha hecho muy popular últimamente en la programación de interfaces gráficas de usuario, y es una buena manera de mostrar bloques de información similar que guardan una separación en la pantalla.

La primera función que se necesita saber, como probablemente se adivine, se usa para crear un nuevo control de fichas.

```
notebook = gtk.Notebook()
```

Una vez que se han creado las fichas, hay unos cuantos métodos que manipulan el control. Veámoslos uno a uno.

El primero que trataremos especifica cómo colocar los indicadores de página. Estos indicadores o "pestañas" como se las conoce, se pueden colocar de cuatro maneras: arriba, abajo, izquierda o derecha.

```
notebook.set_tab_pos(pos)
```

pos será uno de las siguientes, que son bastante descriptivas:

```
POS_LEFT    # izquierda
POS_RIGHT   # derecha
POS_TOP     # arriba
POS_BOTTOM  # abajo
```

`POS_TOP` es el valor predeterminado.

Lo siguiente que trataremos será cómo añadir páginas a las fichas. Hay tres formas de añadir páginas a un `NoteBook`. Veamos las dos primeras, ya que son bastante similares.

```
notebook.append_page(child, tab_label)

notebook.prepend_page(child, tab_label)
```

Estos métodos añaden páginas a las fichas insertándolas al final (append), o al principio (prepend). *child* es el control que se colocará dentro de la página en cuestión, y *tab_label* es el título para la página que se está añadiendo. El control *child* debe crearse por separado, y normalmente es un conjunto de opciones de configuración dentro de otro control contenedor, tal como una tabla.

El último método para añadir una página a las fichas contiene todas las propiedades de los otros dos, y además permite especificar en qué posición se insertará la página en las fichas.

```
notebook.insert_page(child, tab_label, position)
```

Los parámetros son los mismos que en append() y prepend() excepto en que contiene un parámetro extra, *position*. Este parámetro se usa para especificar en qué lugar la página se insertará; la primera página está en la posición cero.

Ahora que sabemos cómo añadir una página, veamos cómo podemos borrar una página de las fichas.

```
notebook.remove_page(page_num)
```

Este método borra la página especificada por *page_num* de las fichas contenidas en la variable *notebook*.

Para saber cuál es la página actual de las fichas utiliza el método:

```
page = notebook.get_current_page()
```

Los próximos dos métodos son simples llamadas para mover la página hacia adelante o hacia atrás. Simplemente se utilizan en el control de fichas sobre el que se quiera operar.

```
notebook.next_page()

notebook.prev_page()
```

NOTA

Cuando el *notebook* tiene como página actual la última página, y se llama a next_page() , no ocurre nada. De forma análoga, si el *notebook* está en la primera página, y se llama a prev_page(), no pasa nada.

Este método fija la página "activa". Si se quiere que las fichas comiencen con la página 5 activa, se usará este método. Si no se usa, el comportamiento predeterminado de las fichas es mostrar la primera página.

```
notebook.set_current_page(page_num)
```

Los siguientes dos métodos añaden o borran las pestañas y el borde respectivamente.

```
notebook.set_show_tabs(show_tabs)

notebook.set_show_border(show_border)
```

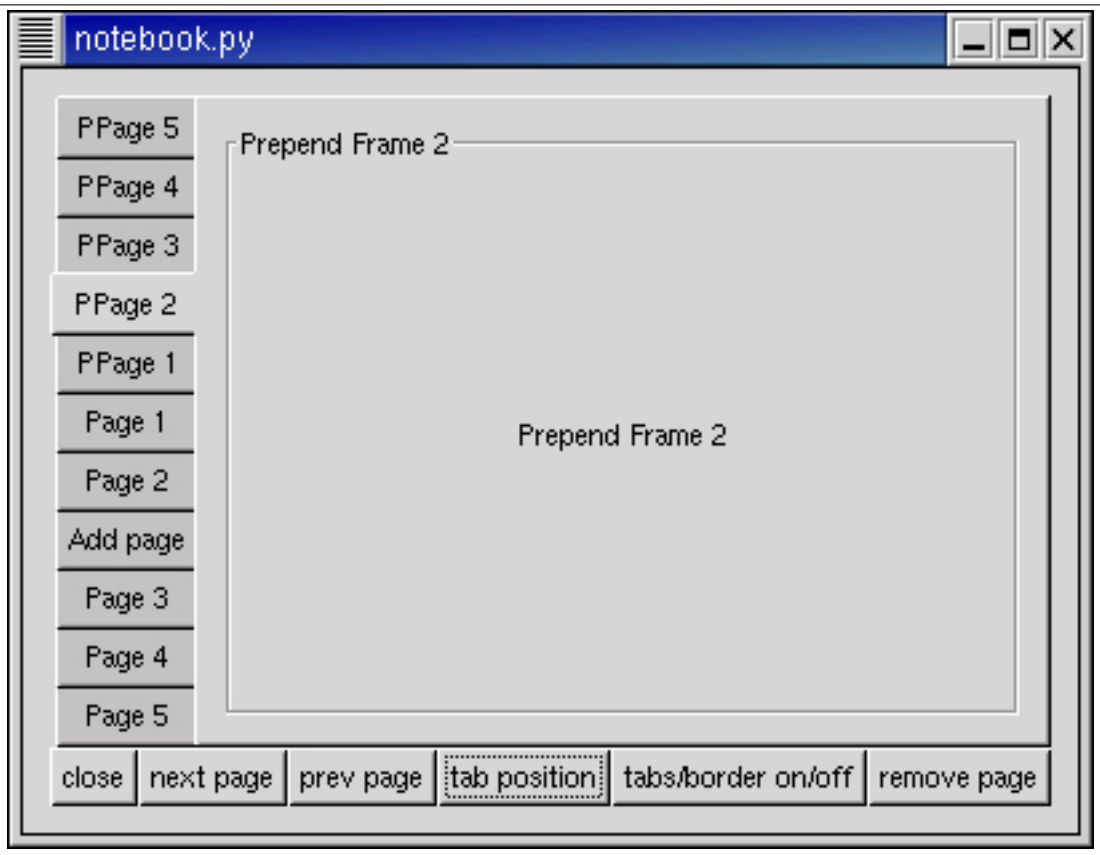
El siguiente método es útil cuando se tiene un gran número de páginas, y las pestañas no caben en la página. Permite desplazarse por las pestañas usando dos botones de flechas.

```
notebook.set_scrollable(scrollable)
```

show_tabs (mostrar pestañas), *show_border* (mostrar border) y *scrollable* (desplazable) pueden ser TRUE o FALSE.

Ahora veamos un ejemplo. El programa notebook.py crea una ventana con unas fichas y seis botones. Las fichas contienen 11 páginas, añadidas de tres formas diferentes, al final, en medio y al principio. Los botones permiten rotar la posición de las pestañas, añadir o borrar las pestañas y el borde, borrar una página, cambiar las páginas hacia delante y hacia atrás, y salir del programa. La figura Figura 10.9 muestra la ventana del programa:

Figura 10.9 Ejemplo de Fichas



El código fuente de notebook.py es:

```
 1 #!/usr/bin/env python
 2
 3 # ejemplo notebook.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class NotebookExample:
10     # Este método rota la posición de las pestañas
11     def rotate_book(self, button, notebook):
12         notebook.set_tab_pos((notebook.get_tab_pos()+1) %4)
13
14     # Añadir/Eliminar las pestañas de página y los bordes
15     def tabsborder_book(self, button, notebook):
16         tval = gtk.FALSE
17         bval = gtk.FALSE
18         if self.show_tabs == gtk.FALSE:
19       tval = gtk.TRUE
20         if self.show_border == gtk.FALSE:
21       bval = gtk.TRUE
22
23         notebook.set_show_tabs(tval)
24         self.show_tabs = tval
25         notebook.set_show_border(bval)
26         self.show_border = bval
27
28     # Eliminar una página de las fichas
29     def remove_book(self, button, notebook):
30         page = notebook.get_current_page()
31         notebook.remove_page(page)
```

```
32          # Need to refresh the widget --
33          # This forces the widget to redraw itself.
34          notebook.queue_draw_area(0,0,-1,-1)
35
36      def delete(self, widget, event=None):
37          gtk.main_quit()
38          return gtk.FALSE
39
40      def __init__(self):
41          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
42          window.connect("delete_event", self.delete)
43          window.set_border_width(10)
44
45          table = gtk.Table(3,6,gtk.FALSE)
46          window.add(table)
47
48          # Crear unas nuevas fichas, definir la posición de las pestañas
49          notebook = gtk.Notebook()
50          notebook.set_tab_pos(gtk.POS_TOP)
51          table.attach(notebook, 0,6,0,1)
52          notebook.show()
53          self.show_tabs = gtk.TRUE
54          self.show_border = gtk.TRUE
55
56          # Añadimos unas cuantas páginas a las fichas
57          for i in range(5):
58              bufferf = "Append Frame %d" % (i+1)
59              bufferl = "Page %d" % (i+1)
60
61              frame = gtk.Frame(bufferf)
62              frame.set_border_width(10)
63              frame.set_size_request(100, 75)
64              frame.show()
65
66              label = gtk.Label(bufferf)
67              frame.add(label)
68              label.show()
69
70              label = gtk.Label(bufferl)
71              notebook.append_page(frame, label)
72
73          # Ahora añadimos una página en un punto determinado
74          checkbutton = gtk.CheckButton("Check me please!")
75          checkbutton.set_size_request(100, 75)
76          checkbutton.show ()
77
78          label = gtk.Label("Add page")
79          notebook.insert_page(checkbutton, label, 2)
80
81          # Finalmente añadimos una página delante
82          for i in range(5):
83              bufferf = "Prepend Frame %d" % (i+1)
84              bufferl = "PPage %d" % (i+1)
85
86              frame = gtk.Frame(bufferf)
87              frame.set_border_width(10)
88              frame.set_size_request(100, 75)
89              frame.show()
90
91              label = gtk.Label(bufferf)
92              frame.add(label)
93              label.show()
94
95              label = gtk.Label(bufferl)
```

```
 96             notebook.prepend_page(frame, label)
 97
 98         # Establecer en qué página empezamos (página 4)
 99         notebook.set_current_page(3)
100
101         # Creamos unos cuantos botones
102         button = gtk.Button("close")
103         button.connect("clicked", self.delete)
104         table.attach(button, 0,1,1,2)
105         button.show()
106
107         button = gtk.Button("next page")
108         button.connect("clicked", lambda w: notebook.next_page())
109         table.attach(button, 1,2,1,2)
110         button.show()
111
112         button = gtk.Button("prev page")
113         button.connect("clicked", lambda w: notebook.prev_page())
114         table.attach(button, 2,3,1,2)
115         button.show()
116
117         button = gtk.Button("tab position")
118         button.connect("clicked", self.rotate_book, notebook)
119         table.attach(button, 3,4,1,2)
120         button.show()
121
122         button = gtk.Button("tabs/border on/off")
123         button.connect("clicked", self.tabsborder_book, notebook)
124         table.attach(button, 4,5,1,2)
125         button.show()
126
127         button = gtk.Button("remove page")
128         button.connect("clicked", self.remove_book, notebook)
129         table.attach(button, 5,6,1,2)
130         button.show()
131
132         table.show()
133         window.show()
134
135 def main():
136     gtk.main()
137     return 0
138
139 if __name__ == "__main__":
140     NotebookExample()
141     main()
```

Espero que esto ayude en el camino para crear fichas en los programas PyGTK.

10.13. Elementos incrustables y puntos de conexión (Plugs y Sockets)

Los `Plugs` y los `Sockets` cooperan para embeber la interfaz de un proceso en otro proceso. Esto también se puede lograr con el uso de Bonobo.

10.13.1. Elementos incrustables (Plugs)

Un `Plug` encapsula una interfaz de usuario aportada por una aplicación de forma que se puede embeber en la interfaz de otra aplicación. La señal "embedded" alerta a la aplicación embebida que ha sido embebida en la interfaz de usuario de la otra aplicación.

Un `Plug` se crea con la siguiente función:

```
plug = gtk.Plug(socket_id)
```

que crea un nuevo `Plug` y lo embebe en el `Socket` identificado por *`socket_id`*. Si *`socket_id`* es 0L, el elemento conectable queda "desconectado" y puede ser conectado más tarde a un `Socket` utilizando el método de `Socket` add_id().

El método de `Plug`:

```
id = plug.get_id()
```

devuelve el identificador de ventana (window ID) de un `Plug`, que se puede usar para embeberlo en un `Socket` utilizando el método de `Socket` add_id().

El programa de ejemplo plug.py ilustra el uso de un elemento conectable Plug:

```
 1 #!/usr/bin/python
 2
 3 import pygtk
 4 pygtk.require('2.0')
 5 import gtk,sys
 6
 7 Wid = 0L
 8 if len(sys.argv) == 2:
 9     Wid = long(sys.argv[1])
10
11 plug = gtk.Plug(Wid)
12 print "Plug ID=", plug.get_id()
13
14 def embed_event(widget):
15     print "I (",widget,") have just been embedded!"
16
17 plug.connect("embedded", embed_event)
18
19 entry = gtk.Entry()
20 entry.set_text("hello")
21 def entry_point(widget):
22     print "You've changed my text to '%s'" % widget.get_text()
23
24 entry.connect("changed", entry_point)
25 plug.connect("destroy", gtk.mainquit)
26
27 plug.add(entry)
28 plug.show_all()
29
30
31 gtk.mainloop()
```

El programa se invoca así:

```
plug.py [windowID]
```

donde *`windowID`* es el ID (identificador) de un `Socket` al que conectar el `Plug`.

10.13.2. Puntos de Conexión (Sockets)

Un `Socket` proporciona el control que permite embeber un control incrustable `Plug` desde otra aplicación a la propia interfaz de forma transparente. Una aplicación crea un control `Socket`, pasa el ID de ventana de ese control a otra aplicación, que luego crea un `Plug` utilizando ese ID de ventana como parámetro. Cualquier control contenido en el elemento `Plug` aparece dentro de la ventana de la primera aplicación.

El identificador (ID) de ventana del `Socket` se obtiene utilizando el método de `Socket` get_id(). Antes de usar este método el `Socket` debe estar realizado y añadido a su control padre.

NOTA

Si se le pasa el ID de ventana del `Socket` a otro proceso que va a crear un elemento `Plug` en el `Socket`, es preciso asegurarse de que el control `Socket` no se va a destruir hasta que se crea el elemento `Plug`.

Cuando se notifica a GTK+ que la ventana embebida ha sido destruída, entonces también se encarga de destruir el `Socket`. Por tanto, se debe prever la posibilidad de que los sockets se destruyan en cualquier momento mientras el bucle de eventos principal está en ejecución. La destrucción de un `Socket` provocará también la destrucción de un `Plug` embebido en él.

La comunicación entre un `Socket` y un `Plug` se hace según el protocolo XEmbed. Este protocolo también se ha implementado en otros toolkits como Qt, lo que permite el mismo nivel de integración al integrar un control de Qt en GTK+ o viceversa.

Creación de un nuevo `Socket` vacío:

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

El `Socket` debe estar contenido en una ventana raíz antes de la invocación del método add_id():

```
socket.add_id(window_id)
```

que añade un cliente XEMBED, tal como un `Plug`, al `Socket`. El cliente puede estar en el mismo proceso o en uno diferente.

Para embeber un `Plug` en un `Socket` se puede, bien crear el `Plug` con:

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

y luego pasar el número devuelto por el método de `Plug` get_id() al método de `Socket` add_id():

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

, o bien se puede invocar el método de `Socket` get_id():

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

para obtener el identificador de ventana (ID) del socket, y luego crear el plug con:

```
plug = gtk.Plug(window_id)
```

El `Socket` debe haber sido añadido ya a una ventana raíz antes de que se pueda hacer esta llamada.

El programa de ejemplo socket.py ilustra el uso de un `Socket`:

```
 1 #!/usr/bin/python
 2
 3 import string
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk,sys
 8
 9 window = gtk.Window()
10 window.show()
11
12 socket = gtk.Socket()
13 socket.show()
14 window.add(socket)
15
16 print "Socket ID=", socket.get_id()
17 window.connect("destroy", gtk.mainquit)
18
19 def plugged_event(widget):
20     print "I (",widget,") have just had a plug inserted!"
21
22 socket.connect("plug-added", plugged_event)
23
```

```
24 if len(sys.argv) == 2:
25     socket.add_id(long(sys.argv[1]))
26
27 gtk.mainloop()
```

Para ejecutar el ejemplo se puede usar plug.py antes:

```
$ python plug.py
Plug ID= 20971522
```

y copiar el identificador ID al primer algumento de socket.py:

```
$ python socket.py 20971522
Socket ID= 48234523
I ( <gtk.Plug object (GtkPlug) at 0x3008dd78> ) have just been embded!
I ( <gtk.Socket object (GtkSocket) at 0x3008ddf0> ) have just had a plug ↩
    inserted!
```

O se puede ejecutar socket.py:

```
$ python socket.py
Socket ID= 20971547
```

y después: plug.py, copiando el identificador ID de ventana:

```
$ python plug.py
20971547
I ( <gtk.Socket object (GtkSocket) at 0x3008ddf0> ) have just had a plug ↩
    inserted!
Plug ID= 48234498
```

Capítulo 11

Control Menú

Hay dos formas de crear menús: la forma fácil y la dificil. Cada una tiene sus usos, pero normalmente se puede usar el `Itemfactory` (Factoria de Elementos) (la forma fácil). La forma "difícil" consiste en crear los menús usando llamadas directamente, mientras que de la forma fácil se hacen llamadas a `GtkItemFactory`. Esto último es mucho más simple, pero cada enfoque tiene sus ventajas y desventajas.

La `Itemfactory` es mucho más fácil de usar, y más fácil de añadir menús, aunque escribir unas cuantas funciones auxiliares para crear menús usando el método manual puede ser más ventajoso de cara a la usabilidad. Por ejemplo, con `Itemfactory`, no es posible añadir imágenes o el carácter '/' a los menús.

11.1. Creación Manual de Menús

Siguiendo la tradición docente, veremos primero la forma difícil. :)

Hay tres controles involucrados en la creación de una barra de menús y de submenús:

- un elemento de menú, que es lo que el usuario va a seleccionar, por ejemplo, "Guardar"
- un menú, que actua como contenedor para elementos de menú, y
- una barra de menú, que sirve de contenedor para cada uno de los menús individuales.

Esto es algo complicado por el hecho de que los controles de elementos de menú se usan para dos cosas distintas. Son tanto los controles que se colocan en el menú, y el control que se coloca en la barra de menú, que, cuando se selecciona, activa el menú.

Veamos las funciones que se usan para crear menús y barras de menús. La primera función se usa para crear una nueva barra de menú:

```
menu_bar = gtk.MenuBar()
```

Esta función bastante autoexplicativa crea una nueva barra de menús. Se puede usar el método `gtk.Container` add() para meter la barra de menús en una ventana, o los métodos pack de `gtk.Box` para meterlo en una caja - igual que los botones.

```
menu = gtk.Menu()
```

Esta función devuelve una referencia a un nuevo menú; nunca se mostrará (con el método show()), es sólo un contenedor para elementos de menús. Esto se aclarará más cuando se vea el ejemplo que aparece más abajo.

La siguiente función se usa para crear elementos de menús que se colocan en el menú (y la barra de menús):

```
menu_item = gtk.MenuItem(label=None)
```

El parámetro *label*, si existe, se analizará buscando caracteres mnemónicos. Esta llamada se usa para crear los elementos de menú que se van a visualizar. Debe recordarse la diferenciar entre un "menú" como el que se crea con la función `gtk.Menu`() y un "elemento de menú" como el que se crea con `gtk.MenuItem`() . El elemento de menú será en realidad un botón con una acción asociada, mientras que un menú será un contenedor que contiene elementos de menú.

Una vez se ha creado un elemento de menú es preciso meterlo en un menú. Esto se consigue con el método append() . Para poder saber cuándo el usuario selecciona un elemento, se necesita conectarlo a la señal "activate" de la forma habitual. Por tanto, si se quiere crear un menú estándar Archivo, con las opciones Abrir, Guardar, y Salir, el código sería algo asi como:

```
file_menu = gtk.Menu()    # No necesitamos mostrar los menús

# Creamos los elementos de menú
open_item = gtk.MenuItem("Abrir")
save_item = gtk.MenuItem("Guardar")
quit_item = gtk.MenuItem("Salir")

# Los añadimos al menú
file_menu.append(open_item)
file_menu.append(save_item)
file_menu.append(quit_item)

# Se conectan las funciones de retrollamada a la señal "activate"
open_item.connect_object("activate", menuitem_response, "file.open")
save_item.connect_object("activate", menuitem_response, "file.save")

# se conecta el elemento quit a nuestra función de salida
quit_item.connect_object ("activate", destroy, "file.quit")

# Mostramos los elementos de menú
open_item.show()
save_item.show()
quit_item.show()
```

En este punto tenemos nuestro menú. Ahora es necesario crear una barra de menú y un elemento de menú para la entrada Archivo, al que añadiremos nuestro menú. El código es el siguiente:

```
menu_bar = gtk.MenuBar()
window.add(menu_bar)
menu_bar.show()

file_item = gtk.MenuItem("Archivo")
file_item.show()
```

Ahora necesitamos asociar el menú con *file_item*. Esto se hace con el método:

```
menu_item.set_submenu(submenu)
```

Por tanto, nuestro ejemplo continuaría así:

```
menu_item.set_submenu(file_menu)
```

Lo único que nos queda es añadir el menú a la barra de menús, lo cual se consigue con el método:

```
menu_bar.append(child)
```

que en nuestro caso es así:

```
menu_bar.append(file_item)
```

Si queremos el menú justificado a la derecha en la barra de menús, como suelen ser los menús de ayuda, podemos usar el siguiente método (de nuevo en *file_item* en nuestro ejemplo) antes de añadirlo a la barra de menús.

```
menu_item.set_right_justified(right_justified)
```

Aqui tenemos un resumen de los pasos necesarios para crear una barra de menús con menús en ella:

- Se crea un nuevo menú con `gtk.Menu()`
- Se hacen varias llamadas a `gtk.MenuItem()`, una para cada elemento que se quiera tener en el menú. Y se usa el método append() para poner cada uno de estos elementos en el menú.

- Se crea un elemento de menú usando `gtk.MenuItem()`. Este será la raiz del menú, el texto que aparezca en él estará en la propia barra de menús.
- Se usa el método set_submenu() para añadir el menú al elemento raiz del menú (el creado en el paso anterior).
- Se crea una nueva barra de menús con `gtk.MenuBar()`. Este paso se hace únicamente una vez al crear una serie de menús en una barra de menús.
- Se usa el método append() para poner el menú raiz en la barra de menús.

La creación de un menú emergente es prácticamente igual. La diferencia es que el menú no se visualiza "automáticamente" por una barra de menús, sino explícitamente llamando al método popup() desde un evento de pulsación de botón, por ejemplo. Se han de seguir estos pasos:

- Se crea una retrollamada que maneje el evento con el siguiente formato:

```
def handler(widget, event):
```

- y usará el evento para saber donde mostrar el menú.
- En el manejador del evento, si el evento es una pulsación de botón, se trata el evento como un evento de botón (realmente lo es) y se usa como en el código de ejemplo para pasar información al método popup() .
- Se enlaza el manejador del evento al control con:

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

- donde widget es el control al que se está conectando, handler es la función manejadora, y menu es un menú creado con `gtk.Menu()`. Esto puede ser un menú que también está en una barra de menús, tal como se muestra en el código de ejemplo.

11.2. Ejemplo de Menú Manual

Esto es más o menos lo que hace falta. Veamos el programa de ejemplo menu.py para ayudarnos a clarificar los conceptos. La figura Figura 11.1 muestra la ventana del programa:

Figura 11.1 Ejemplo de Menú

El código fuente de menu.py es:

```
1 #!/usr/bin/env python
2
3 # ejemplo menu.py
4
5 import pygtk
6 pygtk.require('2.0')
7 import gtk
8
```

```
 9 class MenuExample:
10     def __init__(self):
11         # create a new window
12         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
13         window.set_size_request(200, 100)
14         window.set_title("GTK Menu Test")
15         window.connect("delete_event", lambda w,e: gtk.main_quit())
16 
17         # Se inicia el control menú, y se ha de recordar que ¡¡NUNCA
18         # se debe mostrar con show() este control!!
19         # Este es el menú que contiene los elementos de menú, el que
20         # se desplegará al pulsar en "Root Menu" en la aplicación
21         menu = gtk.Menu()
22 
23         # Después un pequeño bucle construye tres entradas de menú para
24         # "test-menu".  Obsérvese la llamada a gtk_menu_append.  Aquí vamos
25         # añadiendo una lista de elementos de menú a nuestro menú.   ↩
    Normalmente también
26         # interceptaríamos la señal "clicked" para cada uno de los  ↩
    elementos de menú y
27         # se establecería una retrollamada para ellas, pero se omiten aquí  ↩
    para ahorrar espacio
28         for i in range(3):
29             # Copiar los nombres a buf
30             buf = "Test-undermenu - %d" % i
31 
32             # Crear un nuevo elemento de menú con nombre...
33             menu_items = gtk.MenuItem(buf)
34 
35             # ...y lo añadimos al menú
36             menu.append(menu_items)
37 
38       # Hacemos algo interesante cuando se selecciona el elemento de menú
39       menu_items.connect("activate", self.menuitem_response, buf)
40 
41             # Mostramos el control
42             menu_items.show()
43 
44         # Este es el menú raíz, y será la etiqueta mostrada en la barra de  ↩
    menú
45         # No tendrá un manejador de señal asociado, puesto que únicamente
46         # muestra el resto del menú al pulsarlo.
47         root_menu = gtk.MenuItem("Root Menu")
48 
49         root_menu.show()
50 
51         # Ahora especificamos que queremos que nuestro recien creado "menu" ↩
     sea el
52         # menú del menú raíz "root menu"
53         root_menu.set_submenu(menu)
54 
55         # Una vbox para poner un menú y un botón en él:
56         vbox = gtk.VBox(gtk.FALSE, 0)
57         window.add(vbox)
58         vbox.show()
59 
60         # Creamos una barra de menú para los menús, que añadimos a la  ↩
    ventana principal
61         menu_bar = gtk.MenuBar()
62         vbox.pack_start(menu_bar, gtk.FALSE, gtk.FALSE, 2)
63         menu_bar.show()
64 
65         # Creamos un botón al que añadir el menú emergente
66         button = gtk.Button("press me")
```

```
67          button.connect_object("event", self.button_press, menu)
68          vbox.pack_end(button, gtk.TRUE, gtk.TRUE, 2)
69          button.show()
70
71          # Y finalmente añadimos el elemento de menú a la barra de menú.  ↩
    Este es el
72          # elemento de menú raíz del que hemos estado hablando =)
73          menu_bar.append (root_menu)
74
75          # siempres mostramos la ventana como último paso, de manera que se  ↩
    dibuja todo
76          # de una vez.
77          window.show()
78
79      # Respondemos a una pulsación de botón con un menú pasado como control  ↩
    "widget"
80      #
81      # Nótese que el argumento "widget" es el menú pasado, NO
82      # el botón que fue pulsado.
83      def button_press(self, widget, event):
84          if event.type == gtk.gdk.BUTTON_PRESS:
85              widget.popup(None, None, None, event.button, event.time)
86              # Notificamos al código de llamada que hemos manejado este  ↩
    evento.
87              # La cola acaba aquí.
88              return gtk.TRUE
89          # Notificamos al código de llamada que no manejamos este evento. La ↩
     cola continúa.
90          return gtk.FALSE
91
92      # Imprimimos una cadena cuando se selecciona un elemento de menú
93      def menuitem_response(self, widget, string):
94          print "%s" % string
95
96  def main():
97      gtk.main()
98      return 0
99
100 if __name__ == "__main__":
101     MenuExample()
102     main()
```

También se puede hacer un elemento de menú insensible y, usando una tabla de atajos (aceleradores), conectar teclas a retrollamadas de menús.

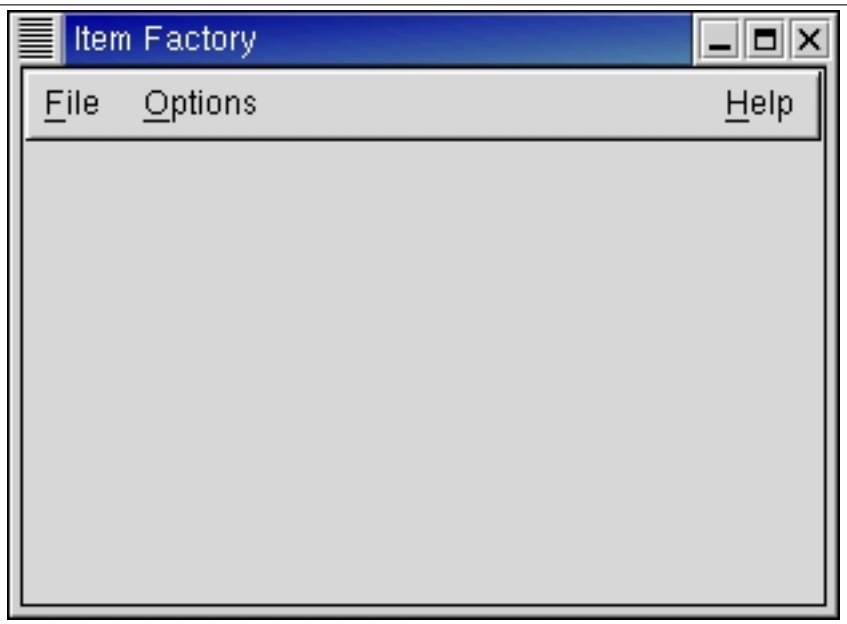
11.3. Uso de la Factoria de Elementos

Ahora que conocemos la forma difícil, así es como se haría usando las llamadas a `gtk.ItemFactory`.

11.4. Ejemplo de Factoria de Elementos - ItemFactory

El programa de ejemplo itemfactory.py usa la `gtk.ItemFactory`. La figura Figura 11.2 muestra la ventana del programa:

Figura 11.2 Ejemplo de Factoria de Elementos



El código fuente de itemfactory.py es:

```
 1   #!/usr/bin/env python
 2
 3   # ejemplo itemfactory.py
 4
 5   import gtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class ItemFactoryExample:
10     # Retrollamada básica obligatoria
11     def print_hello(self, w, data):
12         print "Hello, World!"
13
14     # Esta es la estructura de la Factoria de Elementos usada para generar ↩
    nuevos menús
15     # Item 1: La ruta del menú. La letra después del subrayado indica
16     #         una tecla de atajo cuando el menú se abra
17     # Item 2: La tecla de atajo para el elemento
18     # Item 3: La retrollamada
19     # Item 4: La acción de retrollamada. Esto cambia los parámetros con
20     #         los que la retrollamada es llamada. El valor predeterminado ↩
    es 0
21     # Item 5: El tipo de elemento, usado para definir el tipo al que ↩
    pertenece el elemento
22     #       Aqui están los posibles valores
23
24     #       NULL               -> "<Item>"
25     #       ""                 -> "<Item>"
26     #       "<Title>"          -> crear un elemento de título
27     #       "<Item>"           -> crear un elemento simple
28     #       "<CheckItem>"      -> crear un elemento de activación
29     #       "<ToggleItem>"     -> crear un botón biestado
30     #       "<RadioItem>"      -> crear un botón de exclusión mútua
31     #       <path>             -> ruta de un elemento de exclusión mútua
32     #       "<Separator>"      -> crear un separador
33     #       "<Branch>"         -> crear un contenedor de nuevos elementos
34     #       "<LastBranch>"     -> crear una rama justificada a la derecha
35
36     def get_main_menu(self, window):
```

```
37         accel_group = gtk.AccelGroup()
38
39         # Esta función inicializa la factoría de elementos.
40         # Param 1: El tipo del menú - puede ser MenuBar, Menu,
41         #          u OptionMenu.
42         # Param 2: La ruta o camino del menú.
43         # Param 3: Una referencia a un AccelGroup. La factoria de elementos ↩
    establece la
44         #          tabla de aceleradores (atajos) al generar los menús.
45         item_factory = gtk.ItemFactory(gtk.MenuBar, "<main>", accel_group)
46
47         # Este método genera los elementos de menú. Pasa a la factoría de  ↩
   elementos la lista
48         #  de los elementos de menú
49         item_factory.create_items(self.menu_items)
50
51         # Añadir el nuevo grupo de aceleradores a la ventana.
52         window.add_accel_group(accel_group)
53
54         # es necesario mantener una referencia a item_factory para evitar  ↩
   su destrucción
55         self.item_factory = item_factory
56         # Finalmente, se devuelve la barra de menú creada por la factoría  ↩
   de elementos.
57         return item_factory.get_widget("<main>")
58
59     def __init__(self):
60         self.menu_items = (
61             ( "/_File",         None,         None, 0, "<Branch>" ),
62             ( "/File/_New",     "<control>N", self.print_hello, 0, None ),
63             ( "/File/_Open",    "<control>O", self.print_hello, 0, None ),
64             ( "/File/_Save",    "<control>S", self.print_hello, 0, None ),
65             ( "/File/Save _As", None,         None, 0, None ),
66             ( "/File/sep1",     None,         None, 0, "<Separator>" ),
67             ( "/File/Quit",     "<control>Q", gtk.mainquit, 0, None ),
68             ( "/_Options",      None,         None, 0, "<Branch>" ),
69             ( "/Options/Test",  None,         None, 0, None ),
70             ( "/_Help",         None,         None, 0, "<LastBranch>" ),
71             ( "/_Help/About",   None,         None, 0, None ),
72             )
73         window = gtk.Window(gtk.WINDOW_TOPLEVEL)
74         window.connect("destroy", gtk.mainquit, "WM destroy")
75         window.set_title("Item Factory")
76         window.set_size_request(300, 200)
77
78         main_vbox = gtk.VBox(gtk.FALSE, 1)
79         main_vbox.set_border_width(1)
80         window.add(main_vbox)
81         main_vbox.show()
82
83         menubar = self.get_main_menu(window)
84
85         main_vbox.pack_start(menubar, gtk.FALSE, gtk.TRUE, 0)
86         menubar.show()
87         window.show()
88
89 def main():
90     gtk.main()
91     return 0
92
93 if __name__ == "__main__":
94     ItemFactoryExample()
95     main()
```

Por ahora esto es sólo un ejemplo. Más adelante aparecerá una explicación y muchos comentarios.

Capítulo 12

Área de Dibujo

El control `DrawingArea` (Área de Dibujo) proporciona un lienzo en el que poder dibujar de forma sencilla haciendo uso de los métodos de la clase `gtk.gdk.Drawable` ("dibujable"). Un control `DrawingArea` en realidad encapsula un elemento del tipo `gtk.gdk.Window` (ventana), que es una subclase de `gtk.gdk.Drawable` (al igual que `gtk.gdk.Pixmap`).

Para crear un Área de Dibujo se utiliza la función:

```
drawing_area = gtk.DrawingArea()
```

Los elementos `DrawingArea` (Área de Dibujo) se crean inicialmente con un tamaño de (0,0), por lo que es necesario usar el siguiente método para poder hacer visible el Área de Dibujo (*`drawing_area`*) asignándole un ancho y una altura distintas de cero:

```
drawing_area.set_size_request(width, height)
```

`width` y *`height`* especifican respectivamente el ancho y alto del área de dibujo (`DrawingArea`).

Para dibujar en este control es necesario acceder a su dibujable (`gtk.gdk.Drawable`) asociado, en este caso una ventana (`gtk.gdk.Window`), utilizando su atributo window:

```
drawable = drawing_area.window
```

NOTA

Para poder acceder a la ventana (`gtk.gdk.Window`) asociada al área de dibujo (`DrawingArea`) y poder dibujar en ella el Área de Dibujo debe haber sido instanciada (lo que da lugar a que genere una señal "realize").

12.1. Contexto Gráfico

Hay disponibles varios métodos que nos facilitan el dibujo en un área de dibujo (en su `gtk.gdk.Drawable`). Todos ellos precisan un contexto gráfico (`gtk.gdk.GC`) que alberga la información necesaria para llevar a cabo el dibujado. Para ello, un contexto gráfico (`gtk.gdk.GC`) dispone de los siguientes atributos::

```
background      # fondo
cap_style       # estilo de fin de linea
clip_mask       # máscara de recorte
clip_x_origin   # origen x del rectángulo de recorte
clip_y_origin   # origen y del rectángulo de recorte
fill            # relleno
font            # fuente
foreground      # color de frente (primer plano)
function        # función
graphics_exposures # exposiciones gráficas
```

```
join_style      # estilo de unión de líneas
line_style      # estilo de linea
line_width      # ancho de linea
stipple         # patrón de relleno
sub_window      # subventana
tile            # título
ts_x_origin     # origen x
ts_y_origin     # origen y
```

background (fondo) especifica el `gtk.gdk.Color` utilizado para dibujar el fondo.

foreground (frente) especifica el `gtk.gdk.Color` que se usa para dibujar el color de primer plano.

Para crear un `gtk.gdk.Color` que represente un color dado se emplea el método alloc_color() de la clase `gtk.gdk.Colormap` (Mapa de Color). Previamente a la creación de un color se debe disponer de un mapa de color asociado a un control (widget). Para ello se debe hacer uso del siguiente método:

```
colormap = widget.get_colormap()
```

A la hora de especificar un `gtk.gdk.Color` tenemos varias opciones. Por un lado se puede indicar el color deseado utilizando una cadena de texto (tales como "red" (rojo), "orange" (naranja) o "navajo white" (blanco navajo) de entre las que se definen en el archivo `rgb.txt` de X Window; o, por otro lado, se puede indicar un triplete de números enteros, situados dentro del rango de valores 0 a 65535, que indican la cantidad de rojo (red), verde (green) y azul (blue) cuya mezcla produce el color deseado.

Además se pueden especificar otras propiedades que especifican si el color es modificable (propiedad writeable), de manera que es posible modificar su valor después pero no se puede compartir;o si se debe seleccionar el color más parecido entre los disponibles cuando no sea posible representar el color exacto (propiedad best_match).

El método alloc_color() se define así:

```
color = colormap.alloc_color(red, green, blue, writeable=FALSE, best_match=TRUE ↩
    )

color = colormap.alloc_color(spec, writeable=FALSE, best_match=TRUE)
```

Ejemplos de uso de esta función:

```
navajowhite = colormap.alloc('navajo white')

cyan = colormap.alloc(0, 65535, 65535)
```

cap_style especifica el estilo con el que se dibujan los extremos de las líneas que no están conectadas a otras líneas. Los diferentes estilos disponibles son:

`CAP_NOT_LAST`	igual que `CAP_BUTT` para lineas con ancho distinto de cero. En el caso de líneas de ancho cero no se dibuja el punto final de la linea.
`CAP_BUTT`	los extremos de las lineas se dibujan como cuadrados que se extienden hasta las coordenadas del punto final.
`CAP_ROUND`	los extremos de las lineas se dibujan como semicírculos de diámetro igual al grosor de la linea y cuyo centro se sitúa en el punto final.
`CAP_PROJECTING`	los extremos de las lineas son dibujados como cuadrados que se prolongan medio grosor más allá del punto final.

clip_mask especifica un mapa de píxeles (`gtk.gdk.Pixmap`) que se usará para hacer el recorte del dibujo contenido en el área de dibujo (*drawing_area*).

clip_x_origin y *clip_y_origin* especifican los valores de las coordenadas x e y correspondientes al origen del rectángulo de recorte tomando como referencia la esquina superior izquierda del área de dibujo (*drawing_area*).

fill especifica el estilo de relleno usado en el dibujo. Los estilos de relleno que se encuentran disponibles son:

`SOLID`	dibuja utilizando el color de primer plano (foreground).
`TILED`	dibuja usando un patrón en cuadrícula.

STIPPLED	dibuja usando un patrón de mapa de bits. Los píxeles correspondientes a los bits activados (a uno) del mapa de bits se dibujan usando el color de primer plano (foregraund) y aquellos correspondientes a los bits que no están activados se dejan intactos.
OPAQUE_STIPPLED	dibuja usando un patrón de mapa de bits. Los píxeles correspondientes a los bits del mapa de bits que están activados (a uno) se dibujan con el color de primer plano; aquellos correspondientes a los bits que no están activados se dibujan con el color de fondo.

font es la gtk.gdk.Font (Fuente) que se usa cómo fuente predeterminada para dibujar texto.

NOTA

El uso del atributo *font* está obsoleto.

function especifica una función usada para combinar los valores de los bits de los píxeles de origen con los valores de los bits de los píxeles de destino y producir un pixel transformado. Los 16 posibles valores de este parámetro corresponden a las tablas de verdad posibles de tamaño 2x2. Sin embargo, solamente algunos de estos valores tienen utilidad en la práctica. Así, en el caso de imágenes a color es frecuentel el uso de COPY, XOR e INVERT y en el de mapas de bits también son habituales AND y OR. Los valores posibles de *function* son:

```
COPY        # Copiar
INVERT      # Invertir
XOR         # O exclusivo
CLEAR       # Limpiar
AND         # Y
AND_REVERSE # Y al revés
AND_INVERT  # Y invertida
NOOP        # Nada
OR          # O
EQUIV       # equivalencia
OR_REVERSE  # O al revés
COPY_INVERT # Copiar invertido
OR_INVERT   # O invertido
NAND        # No-Y
SET         # Fijar
```

graphics_exposures especifica si las exposiciones gráficas están activadas (TRUE) o desactivadas (FALSE). Cuando *graphics_exposures* tiene el valor TRUE un fallo al copiar un píxel en una operación de dibujo genera un evento expose y en el caso de que la copia tenga éxito se generará un evento noexpose.

join_style especifica el estilo de unión que se usa en los encuentros de lineas en ángulo. Los estilos disponibles son:

JOIN_MITER	los lados de cada linea se extienden para unirse en ángulo.
JOIN_ROUND	los lados de las dos lineas se unen con un arco de círculo.
JOIN_BEVEL	los lados de las dos lineas se unen en chaflán, una linea recta que forma el mismo ángulo con cada una de las líneas.

line_style especifica el estilo con el que se dibuja una linea. Los estilos disponibles son:

LINE_SOLID	las lineas se dibujan "sólidas" (línea continua).
LINE_ON_OFF_DASH	se dibujan los segmentos impares; los segmentos pares no se dibujan (línea discontínua).
LINE_DOUBLE_DASH	los segmentos impares son normales. Los segmentos pares se dibujan con el color de fondo si el estilo de relleno es SOLID, o con el color de fondo aplicado a la máscara del patrón si el estilo de relleno es STIPPLED.

line_width especifica el ancho con el que se dibujan las líneas.

stipple especifica el `gtk.gdk.Pixmap` que se usará como patrón cuando el relleno esté puesto a `STIPPLED` o a `OPAQUE_STIPPLED`.

sub_window especifica el modo de dibujo de una ventana (`gtk.gdk.Window`) que tiene ventanas hijas (`gtk.gdk.Windows`). Los valores posibles de este parámetro son:

`CLIP_BY_CHILDREN`	sólo se dibuja en la propia ventana pero no en sus ventanas hijas.
`INCLUDE_INFERIORS`	dibuja en la ventana y en sus ventanas hijas.

tile especifica el `gtk.gdk.Pixmap` que se usará para dibujo cuadriculado cuando el relleno (*fill*) esté puesto a `TILED`.

ts_x_origin y *ts_y_origin* especifican las posiciones iniciales (el origen) de los mapas de bits de patrón y de dibujo cuadriculado.

Un Contexto Gráfico nuevo se crea mediante una llamada al método gtk.gdk.Drawable new_gc() :

```
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)
```

Para poder crear un nuevo Contexto Gráfico usando este método es preciso que el control (correspondiente a drawable) sea:

- una ventana (`gtk.gdk.Window`) ya realizada (realized), o bien:
- un mapa de píxeles (`gtk.gdk.Pixmap`) asociado a una ventana (`gtk.gdk.Window`) ya realizada.

Los diversos atributos del Contexto Gráfico tienen los valores por defecto si no se fijan en el método new_gc(). Si se desea establecer el valor de los atributos de los contextos gráficos mediante el método new_gc() resulta muy práctico el uso de argumentos con nombre de Python.

También se pueden establecer los atributos individuales de un `gtk.gdk.GC` asisgnando valores a los atributos correspondientes. Estos son algunos ejemplos de ello:

```
  gc.cap_style = CAP_BUTT
  gc.line_width = 10
  gc.fill = SOLD
  gc.foreground = micolor
```

o usando los siguientes métodos:

```
  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)
```

El patrón de trazos que se usa cuando el parámetro de estilo de linea (*line_style*) es `LINE_ON_OFF_DASH` o `LINE_DOUBLE_DASH` se puede fijar haciendo uso del siguiente método:

```
  gc.set_dashes(offset, dash_list)
```

donde *offset* (desplazamiento) es el índice del valor del trazo inicial en *dash_list* y *dash_list* (lista de trazos) es una lista o tupla que contiene los números de píxeles que dibujar o saltar para formar

los trazos. Éstos se dibujan empezando con el número de píxeles en la posición de desplazamiento(*offset*); después, el siguiente número de píxeles no se dibuja, y, luego, se dibuja el siguiente número de píxeles;; y así se continúa recorriendo todos los números de la lista de trazos y empezando otra vez cuando se llega a su final. Por ejemplo, si la lista de trazos es (2, 4, 8, 16) y el desplazamiento es 1, los trazos se dibujarán así: dibuja 4 píxeles, salta 8 píxeles, dibuja 16 píxeles, salta 2 píxeles, dibuja 4 píxeles, y así sucesivamente.

Es posible hacer una copia de un `gtk.gdk.GC` existente utilizando el método:

```
gc.copy(src_gc)
```

Los atributos de *gc* serán los mismos que los de *src_gc*.

12.2. Métodos de Dibujo

Hay un conjunto general de métodos que se pueden usar para dibujar en el área de dibujo. Estos métodos de dibujo funcionan en cualquier `gtk.gdk.Drawable` (Dibujable), que es una `gtk.gdk.Window` (Ventana) o un `gtk.gdk.Pixmap` (Mapa de Píxeles). Los métodos de dibujo son:

```
drawable.draw_point(gc, x, y) # dibuja_punto
```

gc es el Contexto Gráfico que se usará para hacer el dibujo.

x e *y* son las coordenadas del punto.

```
drawable.draw_line(gc, x1, y1, x2, y2) # dibuja linea
```

gc es el Contexto Gráfico.

x1 e *y1* especifican el punto de inicio de la linea. *x2* e *y2* especifican el punto final de la linea.

```
drawable.draw_rectangle(gc, filled, x, y, width, height) # dibuja rectángulo
```

gc es el Contexto Gráfico.

filled es un valor booleano que indica si el rectángulo debe ser rellenado con el color de primer plano (valor `TRUE`) o no (valor `FALSE`).

x e *y* son las coordenadas de la esquina superior izquierda del rectángulo.

width y *height* son el ancho y el alto del rectángulo.

```
drawable.draw_arc(gc, filled, x, y, width, height, angle1, angle2) # dibuja ↩
    arco
```

gc es el Contexto Gráfico.

filled es un valor booleano que indica si el arco debe ser rellenado con el color de primer plano (valor `TRUE`) o no (valor `FALSE`).

x e *y* son las coordenadas de la esquina superior izquierda del rectángulo que bordea al arco. *width* y *height* son el ancho y el alto del rectángulo que bordea al arco.

angle1 es el ángulo inicial del arco, relativo a la posición de las 3 en punto del reloj, en el sentido contrario de las agujas del reloj, en sesenta y cuatroavos de grado.

angle2 es el ángulo final del arco, relativo a *angle1*, en sesenta y cuatroavos de grado en el sentido de las agujas del reloj.

```
drawable.draw_polygon(gc, filled, points) # dibuja polígono
```

gc es el Contexto Gráfico.

filled es un valor booleano que especifica si el polígono debe ser rellenado con el color de primer plano o(valor `TRUE`) o no (valor `FALSE`).

points es una lista de los puntos que se van a dibujar como un polígono conectado, escrita como una lista de tuplas con las coordenadas, como por ejemplo: [(0,0), (2,5), (3,7), (4,11)].

```
drawable.draw_string(font, gc, x, y, string) # dibuja cadena

drawable.draw_text(font, gc, x, y, string) # dibuja texto
```

font es la `gtk.gdk.Font` (fuente) que se usará para pintar la cadena.
gc es el Contexto Gráfico.
x e *y* son las coordenadas del punto donde se empezará a dibujar la cadena, es decir, la línea base izquierda.
string es la cadena de caracteres a dibujar.

NOTA

Ambos métodos draw_string() y draw_text() están obsoletos - en su lugar se debe usar un `pango.Layout` con el método draw_layout() .

```
drawable.draw_layout(gc, x, y, layout) # dibuja disposición
```

gc es el Contexto Gráfico.
x e *y* son las coordenadas del punto desde el que se empieza a dibujar la disposición.
layout es el `pango.Layout` que se va a dibujar.

```
drawable.draw_drawable(gc, src, xsrc, ysrc, xdest, ydest, width, height) #  ↩
    dibuja dibujable
```

gc es el Contexto Gráfico.
src es el dibujable de orígen.
xsrc e *ysrc* son las coordenadas de la esquina superior izquierda del rectángulo en el dibujable de orígen.
xdest e *ydest* son las coordenadas de la esquina superior izquierda en el área de dibujo.
width y *height* son el ancho y el alto del área del dibujable de origen que será copiada al dibujable (*drawable*). Si *width* o *height* es -1 entonces se usará el ancho o el alto total del dibujable.

```
drawable.draw_image(gc, image, xsrc, ysrc, xdest, ydest, width, height) #  ↩
    dibuja imagen
```

gc es el Contexto Gráfico.
image es la imagen de origen.
xsrc e *ysrc* son las coordenadas de la esquina superior izquierda del rectángulo en el dibujable origen.
xdest e *ydest* son las coordenadas de la esquina superior izquierda del rectángulo en el área de dibujo.
width y *height* son el ancho y el alto del área del dibujable origen que se copiará en el dibujable (*drawable*) destino. Si *width* o *height* es -1 entonces se usará el ancho o el alto total de la imagen.

```
drawable.draw_points(gc, points) # dibuja puntos
```

gc es el Contexto Gráfico.
points es una lista o tupla de pares de coordenadas en tuplas, por ejemplo [(0,0), (2,5), (3,7), (4,11)], que representa la lista de los puntos que se deben dibujar.

```
drawable.draw_segments(gc, segs) # dibuja segmentos
```

gc es el Contexto Gráfico.
segs es una lista o tupla de tuplas que representan las coordenadas de los puntos iniciales y finales de los segmentos que se quiere dibujar, tal como: [(0,0, 1,5), (2,5, 1,7), (3,7, 1,11), (4,11, 1,13)] .

```
drawable.draw_lines(gc, points) # dibuja lineas
```

gc es el Contexto Gráfico.
points es una lista o tupla de pares de coordenadas en tuplas, como por ejemplo [(0,0), (2,5), (3,7), (4,11)], que representa la lista de de los puntos que se van a conectar con líneas.

```
drawable.draw_rgb_image(gc, x, y, width, height, dith, rgb_buf, rowstride) #  ↩
    dibuja imagen rgb
```

```
  drawable.draw_rgb_32_image(gc, x, y, width, height, dith, buf, rowstride) #  ↩
      dibuja imagen rgb 32

  drawable.draw_gray_image(gc, x, y, width, height, dith, buf, rowstride) #  ↩
      dibuja imagen en escala de grises
```

gc es el Contexto Gráfico.

x e *y* son las coordenadas de la esquina superior izquierda del rectángulo que bordea la imagen.

width y *height* son el ancho y el alto del rectángulo que bordea la imagen.

dith es el uno de los métodos de mezclado que se explican a continuación:

Para el método draw_rgb_image() , *rgb_buf* es el conjunto de los datos de la imagen RGB codificado en una cadena como una secuencia de tripletes de píxeles RGB de 8 bits. Para el método draw_rgb_32_image() , *buf* es el conjunto de los datos de la imagen RGB codificado en una cadena como una secuencia de tripletes de píxeles RGB de 8 bits con relleno de 8 bits (4 caracteres por cada píxel RGB). Para el método draw_gray_image() , *buf* es el conjunto de datos de la imagen codificado en una cadena como píxeles de 8 bits.

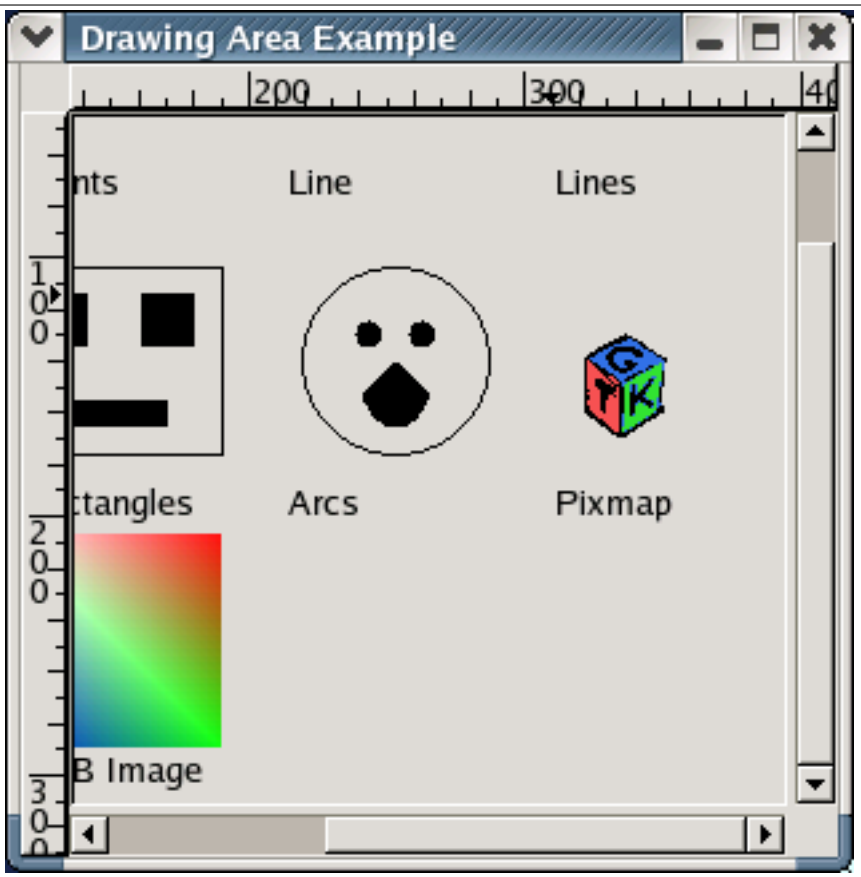
rowstride es el número de caracteres desde el principio de una fila hasta el principio de la siguiente en la imagen. *rowstride* normalmente tiene un valor por defecto de 3 * ancho (*width*) en el método draw_rgb_image() ; 4 * ancho (*width*)para el método draw_rgb_32_image(); y el ancho (*width*) para el método draw_gray_image() . Si *rowstride* es 0 entonces la linea se repetirá un número de veces igual al alto.

Los modos de mezclado (*dither*) son:

```
  RGB_DITHER_NONE    # Nunca se hace mezclado

  RGB_DITHER_NORMAL  # Se hace mezclado cuando se usen 8 bits por píxel (o menos)
sólamente.

  RGB_DITHER_MAX     # Se hace mezclado cuando se usen 16 bits por píxel o menos.
```

El programa de ejemplo drawingarea.py muestra el uso de la mayoría de los métodos de un área de dibujo (`DrawingArea`). También inserta el área de dibujo (`DrawingArea`) en el interior de una ventana con barras de desplazamiento (`ScrolledWindow`) y añade los controles de desplazamiento horizontal y vertical (`Ruler`). La figura Figura 12.1 muestra la ventana resultante:

Figura 12.1 Ejemplo de Área de Dibujo



El código fuente de drawingarea.py está abajo y usa el mapa de píxeles gtk.xpm :

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

# example drawingarea.py

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

class DrawingAreaExample:
    def __init__(self):
        window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        window.set_title("Ejemplo de Área de Dibujo")
        window.connect("destroy", lambda w: gtk.main_quit())
        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.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)
```

```
31          self.area.set_events(gtk.gdk.POINTER_MOTION_MASK |
32                               gtk.gdk.POINTER_MOTION_HINT_MASK )
33          self.area.connect("expose-event", self.area_expose_cb)
34          def motion_notify(ruler, event):
35              return ruler.emit("motion_notify_event", event)
36          self.area.connect_object("motion_notify_event", motion_notify,
37                                   self.hruler)
38          self.area.connect_object("motion_notify_event", motion_notify,
39                                   self.vruler)
40          self.hadj = self.sw.get_hadjustment()
41          self.vadj = self.sw.get_vadjustment()
42          def val_cb(adj, ruler, horiz):
43              if horiz:
44                  span = self.sw.get_allocation()[3]
45              else:
46                  span = self.sw.get_allocation()[2]
47              l,u,p,m = ruler.get_range()
48              v = adj.value
49              ruler.set_range(v, v+span, p, m)
50              while gtk.events_pending():
51                  gtk.main_iteration()
52          self.hadj.connect('value-changed', val_cb, self.hruler, True)
53          self.vadj.connect('value-changed', val_cb, self.vruler, False)
54          def size_allocate_cb(wid, allocation):
55              x, y, w, h = allocation
56              l,u,p,m = self.hruler.get_range()
57              m = max(m, w)
58              self.hruler.set_range(l, l+w, p, m)
59              l,u,p,m = self.vruler.get_range()
60              m = max(m, h)
61              self.vruler.set_range(l, l+h, p, m)
62          self.sw.connect('size-allocate', size_allocate_cb)
63          self.area.show()
64          self.hruler.show()
65          self.vruler.show()
66          self.sw.show()
67          self.table.show()
68          window.show()
69
70      def area_expose_cb(self, area, event):
71          self.style = self.area.get_style()
72          self.gc = self.style.fg_gc[gtk.STATE_NORMAL]
73          self.draw_point(10,10)
74          self.draw_points(110, 10)
75          self.draw_line(210, 10)
76          self.draw_lines(310, 10)
77          self.draw_segments(10, 100)
78          self.draw_rectangles(110, 100)
79          self.draw_arcs(210, 100)
80          self.draw_pixmap(310, 100)
81          self.draw_polygon(10, 200)
82          self.draw_rgb_image(110, 200)
83          return gtk.TRUE
84
85      def draw_point(self, x, y):
86          self.area.window.draw_point(self.gc, x+30, y+30)
87          self.pangolayout.set_text("Punto")
88          self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ↩
    )
89          return
90
91      def draw_points(self, x, y):
92          points = [(x+10,y+10), (x+10,y), (x+40,y+30),
93                    (x+30,y+10), (x+50,y+10)]
```

```
 94            self.area.window.draw_points(self.gc, points)
 95            self.pangolayout.set_text("Puntos")
 96            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ↩
    )
 97            return
 98
 99        def draw_line(self, x, y):
100            self.area.window.draw_line(self.gc, x+10, y+10, x+20, y+30)
101            self.pangolayout.set_text("Línea")
102            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ↩
   )
103            return
104
105        def draw_lines(self, x, y):
106            points = [(x+10,y+10), (x+10,y), (x+40,y+30),
107                      (x+30,y+10), (x+50,y+10)]
108            self.area.window.draw_lines(self.gc, points)
109            self.pangolayout.set_text("Líneas")
110            self.area.window.draw_layout(self.gc, x+5, y+50, self.pangolayout ↩
   )
111            return
112
113        def draw_segments(self, x, y):
114            segments = ((x+20,y+10, x+20,y+70), (x+60,y+10, x+60,y+70),
115                (x+10,y+30 , x+70,y+30), (x+10, y+50 , x+70, y+50))
116            self.area.window.draw_segments(self.gc, segments)
117            self.pangolayout.set_text("Segmentos")
118            self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ↩
   )
119            return
120
121        def draw_rectangles(self, x, y):
122            self.area.window.draw_rectangle(self.gc, gtk.FALSE, x, y, 80, 70)
123            self.area.window.draw_rectangle(self.gc, gtk.TRUE, x+10, y+10,  ↩
   20, 20)
124            self.area.window.draw_rectangle(self.gc, gtk.TRUE, x+50, y+10,  ↩
   20, 20)
125            self.area.window.draw_rectangle(self.gc, gtk.TRUE, x+20, y+50,  ↩
   40, 10)
126            self.pangolayout.set_text("Rectángulos")
127            self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ↩
   )
128            return
129
130        def draw_arcs(self, x, y):
131            self.area.window.draw_arc(self.gc, gtk.FALSE, x+10, y, 70, 70,
132                                      0, 360*64)
133            self.area.window.draw_arc(self.gc, gtk.TRUE, x+30, y+20, 10, 10,
134                                      0, 360*64)
135            self.area.window.draw_arc(self.gc, gtk.TRUE, x+50, y+20, 10, 10,
136                                      0, 360*64)
137            self.area.window.draw_arc(self.gc, gtk.TRUE, x+30, y+10, 30, 50,
138                                      210*64, 120*64)
139            self.pangolayout.set_text("Arcos")
140            self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ↩
   )
141            return
142
143        def draw_pixmap(self, x, y):
144            pixmap, mask = gtk.gdk.pixmap_create_from_xpm(
145                self.area.window, self.style.bg[gtk.STATE_NORMAL], "gtk.xpm")
146
147            self.area.window.draw_drawable(self.gc, pixmap, 0, 0, x+15, y+25,
148                                           -1, -1)
```

```
149             self.pangolayout.set_text("Mapa de Píxeles")
150             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ↩
    )
151             return
152 
153         def draw_polygon(self, x, y):
154             points = [(x+10,y+60), (x+10,y+20), (x+40,y+70),
155                       (x+30,y+30), (x+50,y+40)]
156             self.area.window.draw_polygon(self.gc, gtk.TRUE, points)
157             self.pangolayout.set_text("Polígono")
158             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ↩
    )
159             return
160 
161         def draw_rgb_image(self, x, y):
162             b = 80*3*80*['\0']
163             for i in range(80):
164                 for j in range(80):
165                     b[3*80*i+3*j] = chr(255-3*i)
166                     b[3*80*i+3*j+1] = chr(255-3*abs(i-j))
167                     b[3*80*i+3*j+2] = chr(255-3*j)
168             buff = string.join(b, '')
169             self.area.window.draw_rgb_image(self.gc, x, y, 80, 80,
170                                             gtk.gdk.RGB_DITHER_NONE, buff, 80*3)
171             self.pangolayout.set_text("Imagen RGB")
172             self.area.window.draw_layout(self.gc, x+5, y+80, self.pangolayout ↩
    )
173             return
174 
175     def main():
176         gtk.main()
177         return 0
178 
179     if __name__ == "__main__":
180         DrawingAreaExample()
181         main()
```

Capítulo 13

Control de Vista de Texto

13.1. Perspectiva general de la Vista de Texto

El control TextView y sus objetos asociados (TextBuffers, TextMarks, TextIters, TextTags y TextTagTables) proporcionan un potente marco para la edición de textos multilínea.

Un TextBuffer (Buffer de Texto) contiene el texto que se visualizará en uno o más controles TextView (Vista de Texto)

En GTK+ 2.0 el texto se codifica en UTF-8 de modo que la codificación de un caracter puede estar compuesta por varios bytes. Dentro de un TextBuffer es necesario diferenciar entre contadores de carácteres (llamados desplazamientos) y contadores de bytes (llamados índices).

Los `TextIters` (Iteradores de Texto) proporcionan una representación efímera de la posición entre dos carácteres dentro de un TextBuffer . Los `TextIters` son válidos hasta que el número de caracteres en el TextBuffer cambia; Por ejemplo, siempre que se inserten o se borren carácteres en el TextBuffer todos los `TextIters` se invalidan. Los `TextIters` son la principal forma de especificar localizaciones en un TextBuffer para manipular texto.

Los `TextMarks` (Marcas de Texto) se proporcionan para permitir almacenar posiciones en un TextBuffer que se mantienen entre modificaciones del buffer. Una marca es cómo un TextIter (representa una posición entre dos carácteres en un TextBuffer) pero si el texto alrededor de la marca se borra, la marca permanece donde estaba el texto borrado. De la misma forma, si se inseta texto en la marca, la marca acaba bien a la izquierda o bien a la derecha del texto insertado, dependiendo de la gravedad de la marca - gravedad a la derecha deja la marca a la derecha del texto insertado mientras que gravedad a la izquierda deja la marca a la izquierda. Las TextMarks se pueden asociar a un nombre o dejarlas anónimas si no se les da un nombre. Cada TextBuffer tiene dos marcas predefinidas llamadas *`insert`* (insertar) y *`selection_bound`* (límite de selección). Estas marcas se refieren al punto de inserción y al límite de la selección (la selección está entre las marcas *`insert`* y *`selection_bound`*).

Las TextTags (Etiquetas de Texto) son objetos que especifican un conjunto de atributos que se pueden aplicar a un rango de texto en un TextBuffer. Cada TextBuffer tiene una TextTagTable (Tabla de Etiquetas de Texto) que contiene las etiquetas disponibles en ese buffer. Las TextTagTables se pueden compartir entre TextBuffers para ofrecer consistencia. Los TextTags normalmente se usan para cambiar la apariencia de un rango de texto pero también pueden usarse para evitar que un rango de texto sea editado.

13.2. Vistas de Texto

Sólo hay una función para crear un control `TextView` (Vista de Texto).

```
textview = gtk.TextView(buffer=None)
```

Cuando se crea una `TextView` también se creará un TextBuffer (Buffer de Texto) asociado y una TextTagTable (Tabla de Etiquetas de Texto) de forma predeterminada. Si quieres usar un TextBuffer ya existente en una vista de texto `TextView` puedes especificarlo en el método anterior. Para cambiar el TextBuffer que usa una `TextView` usa el siguiente método:

```
textview.set_buffer(buffer)
```

Usa el siguiente método para obtener una referencia al TextBuffer a partir de una `TextView`:

```
buffer = textview.get_buffer()
```

Un control de `TextView` no tiene barras de desplazamiento para ajustar la vista en caso de que el texto sea más grande que la ventana. Para incluir barras de desplazamiento, es necesario incluir la `TextView` en una ScrolledWindow (Ventana de Desplazamiento).

Una `TextView` se puede usar para permitir la edición de un cuerpo de texto, o para mostrar varias líneas de un texto de sólo lectura al usuario o usuaria. Para cambiar entre estos modos de operación se utiliza el método:

```
textview.set_editable(setting)
```

El argumento *`setting`* puede ser `TRUE` o `FALSE` y especifica si se permite la edición del contenido del control `TextView`. El modo de edición de la `TextView` se puede cambiar por zonas de texto dentro del TextBuffer usando TextTags.

Puedes obtener el modo actual de edición usando el método:

```
setting = textview.get_editable()
```

Cuando la `TextView` no es editable probablemente se debería ocultar el cursor usando el método:

```
textview.set_cursor_visible(setting)
```

El argumento *`setting`* puede ser `TRUE` o `FALSE` y especifica si el cursor debe ser visible. La `TextView` puede ajustar las líneas de texto que son demasiado largas para que quepan en una única línea de la ventana. El comportamiento predeterminado es no ajustar las líneas. Es posible cambiarlo usando el método:

```
textview.set_wrap_mode(wrap_mode)
```

Este método te permite especificar que el texto debe ajustarse en los límites de palabras o caracteres. El argumento *`word_wrap`* puede ser:

```
gtk.WRAP_NONE # sin ajuste
gtk.WRAP_CHAR # ajuste por caracteres
gtk.WRAP_WORD # ajuste por palabras
```

La justificación predetermianda del texto en una `TextView` se puede establecer y obtener usando los métodos:

```
textview.set_justification(justification)
justification = textview.get_justification()
```

donde *`justification`* puede ser:

```
gtk.JUSTIFY_LEFT   # justificación a la izquierda
gtk.JUSTIFY_RIGHT  # justificación a la derecha
gtk.JUSTIFY_CENTER # justificación al centro
```

NOTA

La *`justificación`* será `JUSTIFY_LEFT` si el *`wrap_mode`* (modo de ajuste) es `WRAP_NONE`. Las etiquetas asociadas con un TextBuffer pueden cambiar la justificación predeterminada.

Se pueden modificar y ver otros atributos predeterminados en una `TextView`, tales como el margen izquierdo, el margen derecho, las tabulaciones y la indentación de párrafos, usando los siguientes métodos:

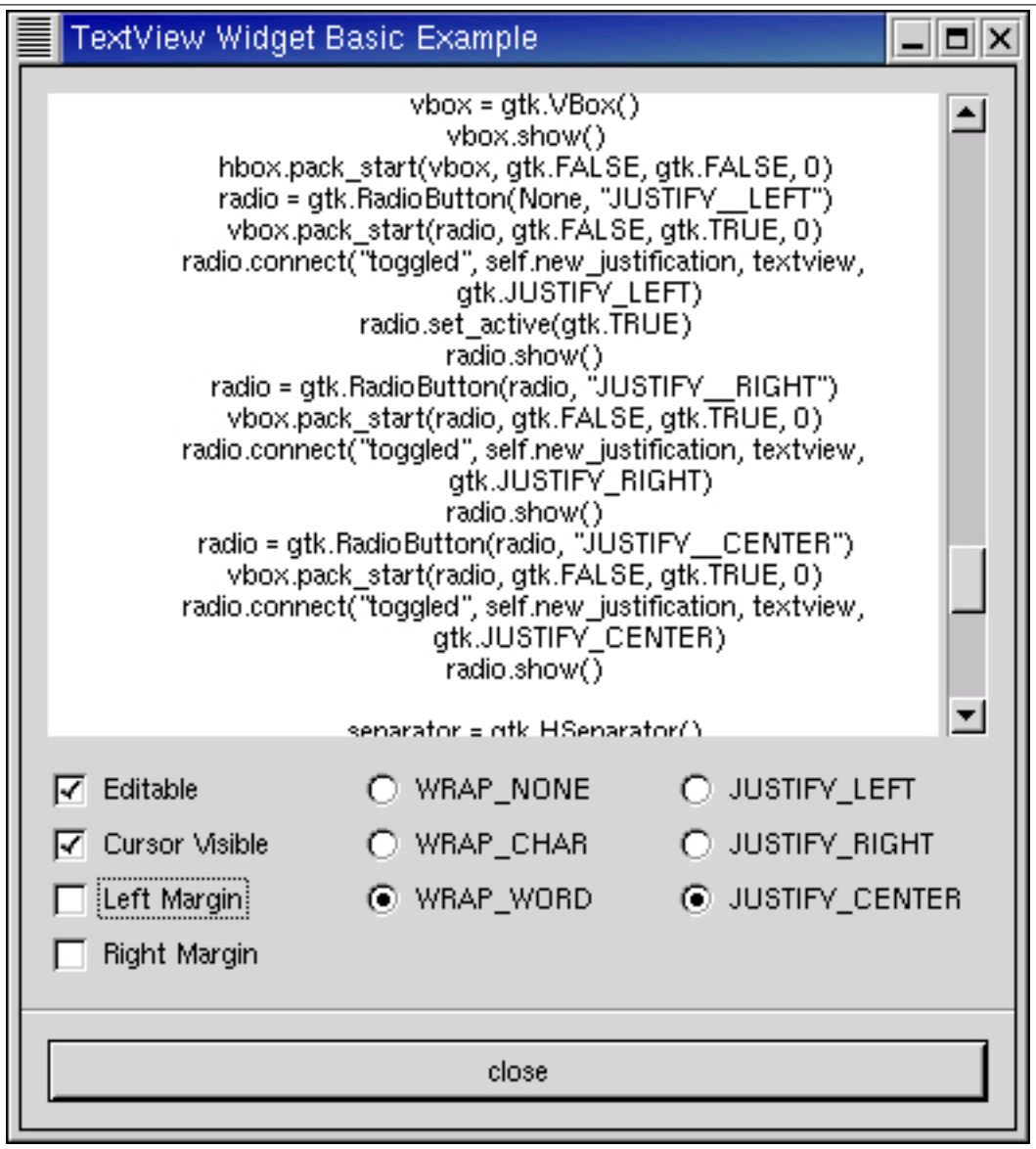
```
# margen izquierdo
textview.set_left_margin(left_margin)
left_margin = textview.get_left_margin()
```

```
# margen derecho
textview.set_right_margin(right_margin)
right_margin = textview.get_right_margin()

# indentación
textview.set_indent(indent)
indent = textview.get_indent()

#espacio anterior de línea (en píxeles)
textview.set_pixels_above_lines(pixels_above_line)
pixels_above_line = textview.get_pixels_above_lines()

#espacio posterior de línea (en píxeles)
textview.set_pixels_below_lines(pixels_below_line)
pixels_below_line = textview.get_pixels_below_lines()

# píxeles en ajuste
textview.set_pixels_inside_wrap(pixels_inside_wrap)
pixels_inside_wrap = textview.get_pixels_inside_wrap()

# tabulaciones
textview.set_tabs(tabs)
tabs = textview.get_tabs()
```

left_margin, *right_margin*, *indent*, *pixels_above_lines*, *pixels_below_lines* y *pixels_inside_wrap* se especifican en píxeles. Los valores predeterminados de estos parámetros se pueden modificar con etiquetas asociadas a un TextBuffer. *tabs* es un `pango.TabArray`.

El programa de ejemplo textview-basic.py ilustra el uso básico del control `TextView`:

Figura 13.1 Ejemplo básico de Vista de Texto



El código fuente del programa es:

```
 1 #!/usr/bin/env python
 2
 3 # example textview-basic.py
 4
 5 import pygtk
 6 pygtk.require('2.0')
 7 import gtk
 8
 9 class TextViewExample:
10     def toggle_editable(self, checkbutton, textview):
11         textview.set_editable(checkbutton.get_active())
12
13     def toggle_cursor_visible(self, checkbutton, textview):
14         textview.set_cursor_visible(checkbutton.get_active())
15
16     def toggle_left_margin(self, checkbutton, textview):
17         if checkbutton.get_active():
18             textview.set_left_margin(50)
19         else:
20             textview.set_left_margin(0)
```

```
21
22      def toggle_right_margin(self, checkbutton, textview):
23          if checkbutton.get_active():
24              textview.set_right_margin(50)
25          else:
26              textview.set_right_margin(0)
27
28      def new_wrap_mode(self, radiobutton, textview, val):
29          if radiobutton.get_active():
30              textview.set_wrap_mode(val)
31
32      def new_justification(self, radiobutton, textview, val):
33          if radiobutton.get_active():
34              textview.set_justification(val)
35
36      def close_application(self, widget):
37          gtk.main_quit()
38
39      def __init__(self):
40          window = gtk.Window(gtk.WINDOW_TOPLEVEL)
41          window.set_resizable(gtk.TRUE)
42          window.connect("destroy", self.close_application)
43          window.set_title("TextView Widget Basic Example")
44          window.set_border_width(0)
45
46          box1 = gtk.VBox(gtk.FALSE, 0)
47          window.add(box1)
48          box1.show()
49
50          box2 = gtk.VBox(gtk.FALSE, 10)
51          box2.set_border_width(10)
52          box1.pack_start(box2, gtk.TRUE, gtk.TRUE, 0)
53          box2.show()
54
55          sw = gtk.ScrolledWindow()
56          sw.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
57          textview = gtk.TextView()
58          textbuffer = textview.get_buffer()
59          sw.add(textview)
60          sw.show()
61          textview.show()
62
63          box2.pack_start(sw)
64          # Load the file textview-basic.py into the text window
65          infile = open("textview-basic.py", "r")
66
67          if infile:
68              string = infile.read()
69              infile.close()
70              textbuffer.set_text(string)
71
72          hbox = gtk.HButtonBox()
73          box2.pack_start(hbox, gtk.FALSE, gtk.FALSE, 0)
74          hbox.show()
75
76          vbox = gtk.VBox()
77          vbox.show()
78          hbox.pack_start(vbox, gtk.FALSE, gtk.FALSE, 0)
79          # check button to toggle editable mode
80          check = gtk.CheckButton("Editable")
81          vbox.pack_start(check, gtk.FALSE, gtk.FALSE, 0)
82          check.connect("toggled", self.toggle_editable, textview)
83          check.set_active(gtk.TRUE)
84          check.show()
```

```
 85         # check button to toggle cursor visiblity
 86         check = gtk.CheckButton("Cursor Visible")
 87         vbox.pack_start(check, gtk.FALSE, gtk.FALSE, 0)
 88         check.connect("toggled", self.toggle_cursor_visible, textview)
 89         check.set_active(gtk.TRUE)
 90         check.show()
 91         # check button to toggle left margin
 92         check = gtk.CheckButton("Left Margin")
 93         vbox.pack_start(check, gtk.FALSE, gtk.FALSE, 0)
 94         check.connect("toggled", self.toggle_left_margin, textview)
 95         check.set_active(gtk.FALSE)
 96         check.show()
 97         # check button to toggle right margin
 98         check = gtk.CheckButton("Right Margin")
 99         vbox.pack_start(check, gtk.FALSE, gtk.FALSE, 0)
100         check.connect("toggled", self.toggle_right_margin, textview)
101         check.set_active(gtk.FALSE)
102         check.show()
103         # radio buttons to specify wrap mode
104         vbox = gtk.VBox()
105         vbox.show()
106         hbox.pack_start(vbox, gtk.FALSE, gtk.FALSE, 0)
107         radio = gtk.RadioButton(None, "WRAP__NONE")
108         vbox.pack_start(radio, gtk.FALSE, gtk.TRUE, 0)
109         radio.connect("toggled", self.new_wrap_mode, textview, gtk. ↩
    WRAP_NONE)
110         radio.set_active(gtk.TRUE)
111         radio.show()
112         radio = gtk.RadioButton(radio, "WRAP__CHAR")
113         vbox.pack_start(radio, gtk.FALSE, gtk.TRUE, 0)
114         radio.connect("toggled", self.new_wrap_mode, textview, gtk. ↩
    WRAP_CHAR)
115         radio.show()
116         radio = gtk.RadioButton(radio, "WRAP__WORD")
117         vbox.pack_start(radio, gtk.FALSE, gtk.TRUE, 0)
118         radio.connect("toggled", self.new_wrap_mode, textview, gtk. ↩
    WRAP_WORD)
119         radio.show()
120
121         # radio buttons to specify justification
122         vbox = gtk.VBox()
123         vbox.show()
124         hbox.pack_start(vbox, gtk.FALSE, gtk.FALSE, 0)
125         radio = gtk.RadioButton(None, "JUSTIFY__LEFT")
126         vbox.pack_start(radio, gtk.FALSE, gtk.TRUE, 0)
127         radio.connect("toggled", self.new_justification, textview,
128                       gtk.JUSTIFY_LEFT)
129         radio.set_active(gtk.TRUE)
130         radio.show()
131         radio = gtk.RadioButton(radio, "JUSTIFY__RIGHT")
132         vbox.pack_start(radio, gtk.FALSE, gtk.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, gtk.FALSE, gtk.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, gtk.FALSE, gtk.TRUE, 0)
144         separator.show()
145
```

```
146         box2 = gtk.VBox(gtk.FALSE, 10)
147         box2.set_border_width(10)
148         box1.pack_start(box2, gtk.FALSE, gtk.TRUE, 0)
149         box2.show()
150
151         button = gtk.Button("close")
152         button.connect("clicked", self.close_application)
153         box2.pack_start(button, gtk.TRUE, gtk.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()
```

Las líneas 10-34 definen las retrollamadas para los botones de exclusión mútua y los botones de activación que se usan para cambiar los atributos predeterminados de la `TextView`. Las líneas 55-63 crean una ScrolledWindow que contenga la `TextView`. La ScrolledWindow se empaqueta en una `VBox` con los botones que se crean en las líneas 72-140. El TextBuffer asociado con la `TextView` se rellena con el contenido del archivo fuente en las líneas 64-70.

13.3. Buffers de Texto

El `TextBuffer` (Buffer de Texto) es el componente principal del sistema de edición de texto de PyGTK. Contiene el texto, las TextTag (Etiquetas de Texto) en una TextTagTable (Tabla de Etiquetas de Texto) y las TextMark (Marcas de Texto) que juntos describen cómo debe visualizarse el texto y permiten la modificación del texto de forma interactiva. Como ya se dijo en la sección anterior, un `TextBuffer` está asociado con una o más TextView (Vistas de Texto), que muestran el contenido del `TextBuffer`.

Un `TextBuffer` se puede crear automáticamente cuando se crea una TextView o se puede crear con la función:

```
textbuffer = TextBuffer(tabla=None)
```

donde *`tabla`* es una TextTagTable. Si no se especifica *`tabla`* (o si es `None`) se creará una TextTagTable para el Buffer de Texto.

Existen numerosos métodos que se pueden usar para:

- insertar y borrar texto en un buffer
- crear, borrar y manipular marcas
- manipular el cursor y la selección
- crear, aplicar y borrar etiquetas
- especificar y manipular TextIter (Iteradores de Texto)
- obtener información de estado

13.3.1. Información de estado de un Buffer de Texto

Se puede conocer el número de líneas en un *`textbuffer`* usando el método:

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

De manera análoga es posible saber el número de caracteres en el *`textbuffer`* usando:

```
char_count = textbuffer.get_char_count()
```

Cuando el contenido del *textbuffer* cambia la bandera de modificación del buffer de texto se activa. El estado de la bandera de modificación se puede consultar usando el método:

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

Si el programa guarda el contenido del buffer de texto el siguiente método se puede usar para reiniciar la bandera de modificación:

```
textbuffer.set_modified(setting)
```

13.3.2. Creación de Iteradores de Texto

Un TextIter se usa para especificar una localización dentro de un `TextBuffer` entre dos caracteres. Los métodos de `TextBuffer` que manipulan texto usan Iteradores de Texto TextIter para especificar dónde se aplicará el método. Los Iteradores de Texto TextIter tienen un gran número de métodos que se describen en la sección TextIter.

Los métodos básicos del `TextBuffer` que se usan para crear TextIters son:

```
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() crea un iterador que se situa después de tantos caracteres como diga el argumento *char_offset* a partir del comienzo del buffer de texto.

get_iter_at_line() crea un iterador que está justo antes del primer caracter en la línea que diga el parámetro *line_number*.

get_iter_at_line_offset() crea un iterador que está justo detrás del carácter especificado por el parámetro *line_offset* en la línea especificada por el parámetro *line_number*.

get_iter_at_mark() crea un iterador que está en la misma posición que la marca especificada por el parámetro *mark*.

Los siguientes métodos crean uno o más iteradores en localizaciones específicas del buffer de texto:

```
startiter = textbuffer.get_start_iter()

enditer = textbuffer_get_end_iter()

startiter, enditer = textbuffer.get_bounds()

start, end = textbuffer.get_selection_bounds()
```

get_start_iter() crea un iterador que está justo antes del primer caracter en el buffer de texto.

get_end_iter() crea un iterador que está justo después del último caracter en el buffer de texto.

get_bounds() crea una tupla de dos iteradores que están justo antes del primer caracter y justo detrás del último caracter del buffer de texto respectivamente.

get_selection_bounds() crea una tupla de dos iteradores que están en las mismas posiciones que las marcas *insert* y *selection_bound* en el buffer de texto.

13.3.3. Inserción, Obtención y Eliminación de Texto

El texto de un `TextBuffer` se puede fijar con el método:

```
textbuffer.set_text(text)
```

Este método reemplaza el contenido actual del buffer de texto con el texto *text* que se le pasa como parámetro.

El método más general para insertar caracteres en un buffer de texto es:

```
textbuffer.insert(iter, text)
```

el cual inserta el texto *text* en la posición del buffer de texto especificada por el iterador *iter*.

Si quieres simular la inserción de texto que produciría un usuario interactivo usa este método:

```
result = textbuffer.insert_interactive(iter, text, default_editable)
```

el cual inserta el texto *text* en la posición del buffer de texto especificada por el iterador *iter* pero sólo si la posición es editable (es decir, no tiene una etiqueta que especifique que el texto no es editable) y el argumento *default_editable* es TRUE. El resultado indica si el texto fue insertado o no.

El argumento *default_editable* indica si es editable o no cuando el texto no tiene una etiqueta que lo especifique; *default_editable* normalmente se determina llamando al método get_editable() de TextView.

Otros métodos que insertan texto son:

```
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() es una función auxiliar que inserta el texto en la posición actual del cursor (señalada por *insert*).

insert_range() copia el texto, pixbuffers y etiquetas entre *start* y *end* de un TextBuffer (si las etiquetas pertenecen a otro buffer de texto la tabla de etiquetas debe ser la misma) y los inserta en el buffer de texto en la posición especificada por *iter*.

Las versiones interactivas de estos métodos funcionan de la misma forma excepto que sólo insertarán el texto si las posiciones son editables.

Finalmente, el texto se puede insertar y asociar a etiquetas al mismo tiempo usando los métodos:

```
textbuffer.insert_with_tags(iter, text, tag1, tag2, ...)

textbuffer.insert_with_tags_by_name(iter, text, tagname1, tagname2, ...)
```

insert_with_tags() inserta el texto *text* en el buffer de texto en la posición especificada por *iter* y le aplica las etiquetas especificadas.

insert_with_tags_by_name() hace lo mismo pero te permite especificar las etiquetas por su nombre.

El texto de un buffer de texto se puede borrar usando los métodos:

```
textbuffer.delete(start, end)

result = textbuffer.delete_interactive(start, end, default_editable)
```

delete() borra el texto entre los iteradores TextIter *start* y *end* en el buffer de texto.

delete_interactive() borra todo el texto editable (determinado por las etiquetas de texto aplicables y el argumento *default_editable*) entre *start* y *end*.

Puedes obtener una copia del texto de un buffer de texto usando los métodos:

```
text = textbuffer.get_text(start, end, include_hidden_chars=TRUE)

text = textbuffer.get_slice(start, end, include_hidden_chars=TRUE)
```

get_text() devuelve una copia del texto en el buffer de texto entre *start* y *end*; si el argumento *include_hidden_chars* es FALSE el texto que no se visualice no se devuelve. Los caracteres que representan imágenes incrustadas o controles, son excluidos.

get_slice() es igual que get_text() excepto que el texto que devuelve incluye un carácter 0xFFFC por cada imágene incrustada o control.

13.3.4. Marcas de Texto (TextMark)

Las TextMarks (Marcas de Texto) son similares a los TextIter ya que también especifican posiciones en un TextBuffer entre dos caracteres. Sin embargo, las Marcas de Texto TextMark mantienen su

posición aunque el buffer cambie. Los métodos de las Marcas de Texto TextMark se describirán en la sección TextMark.

Un buffer de texto contiene de serie dos marcas predeterminadas: la marca *insert* (insertar) y la marca *selection_bound* (límite de la selección). La marca *insert* es la posición predeterminada de inserción del texto y la marca *selection_bound* combinada con la marca *insert* define un rango de selección.

Las marcas predeterminadas de serie se pueden obtener con los métodos:

```
insertmark = textbuffer.get_insert()

selection_boundmark = textbuffer.get_selection_bound()
```

Las marcas *insert* y *selection_bound* se pueden colocar simultáneamente en una posición usando el método:

```
textbuffer.place_cursor(where)
```

donde *where* es un iterador de texto que especifica la posición. El método place_cursor() es necesario para evitar crear una selección temporal si las marcas se movieran individualmente.

Las TextMarks se crean usando el método:

```
mark = textbuffer.create_mark(mark_name, where, left_gravity=FALSE)
```

donde *mark_name* es el nombre que se le asigna a la marca (puede ser `None` para crear una marca anónima), *where* es el iterador de texto que especifica la posición de la marca en el buffer de texto y *left_gravity* indica dónde se pondrá la marca cuando se inserte texto en la marca (a la izquierda si es `TRUE` o a la derecha si es `FALSE`).

Se puede mover una marca en el buffer de texto usando los métodos:

```
textbuffer.move_mark(mark, where)

textbuffer.move_mark_by_name(name, where)
```

mark especifica la marca que se va a mover. *name* especifica el nombre de la marca que se va a mover. *where* es un iterador de texto qeu especifica la nueva posición.

Una marca se puede borrar de un buffer de texto usando los métodos:

```
textbuffer.delete_mark(mark)

textbuffer.delete_mark_by_name(name)
```

Se puede recuperar una marca a partir de su nombre con el método:

```
mark = textbuffer.get_mark(name)
```

13.3.5. Creación y Uso de Etiquetas de Texto

Las TextTags (Etiquetas de Texto) contienen uno o más atributos (por ejemplo, colores de frente y de fondo, fuentes de texto, editabilidad) que se pueden aplicar a uno o más rangos de texto en un buffer de texto. Los atributos que se pueden especificar mediante propiedades de una TextTag se describirán en la sección TextTag.

Una TextTag se puede crear con atributos e instalada en la TextTagTable (Tabla de Etiquetas de Texto) de un `TextBuffer` (Buffer de Texto) usando el siguiente método:

```
tag = textbuffer.create_tag(name=None, attr1=val1, attr2=val2, ...)
```

donde *name* es una cadena de texto que especifica el nombre de la etiqueta o `None` si la etiqueta es una etiqueta anónima y los pares de clave-valor especifican los atributos que tendrá la etiqueta. Mira la sección TextTag para obtener más información acerca de qué atributos se pueden establecer mediante las propiedades de una TextTag.

Una etiqueta se puede aplicar a un rango de texto en un buffer de texto usando los métodos:

```
textbuffer.apply_tag(tag, start, end)

textbuffer.apply_tag_by_name(name, start, end)
```

tag es la etiqueta que se va a aplicar al texto. *name* es el nombre de la etiqueta a aplicar. *start* (comienzo) y *end* (final) son los iteradores de texto que especifican el rango de texto que será afectado por la etiqueta.

Se puede borrar una etiqueta de un rango de texto usando los métodos:

```
textbuffer.remove_tag(tag, start, end)

textbuffer.remove_tag_by_name(name, start, end)
```

Se pueden borrar todas las etiquetas de un rango de texto usando el método:

```
textbuffer.remove_all_tags(start, end)
```

13.3.6. Inserción de Imágenes y Controles

Además de texto, un `TextBuffer` puede contener imágenes y un punto de anclaje para controles. Se puede añadir un control a una TextView en un punto de anclaje. Se puede añadir un control diferente en cada TextView que tenga un buffer con un punto de anclaje.

Se puede insertar un pixbuf con el siguiente método:

```
textbuffer.insert_pixbuf(iter, pixbuf)
```

donde *iter* especifica la posición en el *textbuffer* donde insertar el *pixbuf*. La imagen contará como un caracter y será representada en lo que devuelve get_slice() (pero no en lo que devuelve get_text()) como el caracter Unicode "0xFFFC".

Se puede insertar un control GTK+ en una TextView en una posición del buffer especificada por un `TextChildAnchor` (Anclaje de Hijo del Texto). El `TextChildAnchor` contará como un carácter representado por "0xFFFC" de manera similar a un pixbuf.

El `TextChildAnchor` se puede crear e insertar en el buffer usando este método:

```
anchor = text_buffer.create_child_anchor(iter)
```

donde *iter* es la posición del anclaje.

También se puede crear e insertar un `TextChildAnchor` con dos operaciones por separado:

```
anchor = gtk.TextChildAnchor()

text_buffer.insert_child_anchor(iter, anchor)
```

Después se puede añadir el control al TextView en la posición del anclaje usando el método:

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

Se puede obtener la lista de controles en una posición especifica del buffer usando el método:

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

También se puede añadir un control al TextView usando el método:

```
text_view.add_child_in_window(child, which_window, xpos, ypos)
```

donde el control *child* se coloca en la ventana *which_window* en la posición especificada por *xpos* e *ypos*. El parámetro *which_window* indica en cuál de las ventanas que componen el control TextView se colocará el control:

```
gtk.TEXT_WINDOW_TOP     # ventana superior del texto
gtk.TEXT_WINDOW_BOTTOM  # ventana inferior del texto
gtk.TEXT_WINDOW_LEFT    # ventana izquierda del texto
gtk.TEXT_WINDOW_RIGHT   # ventana derecha del texto
gtk.TEXT_WINDOW_TEXT    # ventana de texto del texto
gtk.TEXT_WINDOW_WIDGET  # ventana de control del texto
```

13.4. Iteradores de Texto

Los Iteradores de Texto `TextIter` representan una posición entre dos caracteres en un TextBuffer. Los `TextIters` se crean normalmente usando un método de la clase TextBuffer. Los `TextIters` se invalidan cuando el número de caracteres de un TextBuffer cambia (excepto para el `TextIter` que se usa para inserción o borrado). Insertar o borrar pixbuffers o anclajes también invalida un `TextIter`.

Hay un gran número de métodos asociados con un objeto `TextIter`. Se agrupan en las siguientes secciones según realicen funciones similares.

13.4.1. Atributos de los Iteradores de Texto

El TextBuffer que contiene el `TextIter` se puede recuperar usando el método:

```
buffer = iter.get_buffer()
```

Los siguientes métodos se pueden usar para obtener la posición del `TextIter` en el TextBuffer:

```
offset = iter.get_offset()    # devuelve el desplazamiento en el buffer del ↩
    iterador

line_number = iter.get_line()   # devuelve el número de línea del iterador

line_offset = iter.get_line_offset()  # devuelve el desplazamiento en la línea

numchars = iter.get_chars_in_line() # devuelve el número de caracteres en la ↩
    línea
```

13.4.2. Atributos de Texto de un Iterador de Texto

El objeto `PangoLanguage` que se usa en una determinada posición del iterador en el TextBuffer se obtiene llamando al método:

```
language = iter.get_language()
```

Hay otro método más general para obtener los atributos de texto en una posición de un `TextIter` :

```
result = iter.get_attributes(values)
```

donde *`result`* indica si el parámetro *`values`* (un objeto de la clase `TextAttributes`) fue modificado. El parámetro *`values`* se obtiene usando el siguiente método de la clase TextView :

```
values = textview.get_default_attributes()
```

Los siguientes atributos se pueden obtener a partir de un objeto de la clase `TextAttributes` (no está implementado en PyGTK <=1.99.15):

bg_color	color de fondo
fg_color	color de frente
bg_stipple	bitmap de patrón de fondo
fg_stipple	bitmap de patrón de frente
rise	desplazamiento del texto sobre la línea base
underline	estilo de subrayado
strikethrough	indica si el texto aparece tachado
draw_bg	`TRUE` si algunas marcas afectan al dibujado del fondo
justification	estilo de la justificación
direction	la dirección del texto
font	PangoFontDescription en uso
font_scale	escala de la fuente en uso
left_margin	posición del márgen izquierdo
right_margin	posición del márgen derecho
pixels_above_lines	espacio en píxeles sobre una línea
pixels_below_lines	espacio en píxeles debajo de una línea

pixels_inside_wrap	espacio en píxeles entre líneas solapadas
tabs	PangoTabArray en uso
wrap_mode	modo de ajuste en uso
language	PangoLanguage en uso
invisible	si el texto es invisible (sin implementar en GTK+ 2.0)
bg_full_height	si el fondo está ocupando el alto completo de línea
editable	si el texto es editable
realized	si el texto está realizado
pad1	
pad2	
pad3	
pad4	

13.4.3. Copiar un Iterador de Texto

Se puede duplicar un `TextIter` usando el método:

```
iter_copy = iter.copy()
```

13.4.4. Recuperar Texto y Objetos

Se pueden obtener varias cantidades de texto y objetos de un TextBuffer usando los siguientes métodos TextBuffer :

```
char = iter.get_char()    # devuelve un caracter o 0 si se ha llegado al final ↩
    del buffer

text = start.get_slice(end)   # devuelve el texto entre los iteradores de ↩
    principio y fin

text = start.get_text(end)    # devuelve el texto entre los iteradores de ↩
    principio y fin

pixbuf = iter.get_pixbuf()    # devuelve el pixbuf en esa posición (o None)

anchor = iter.get_child_anchor()  # devuelve el anclaje (o None)

mark_list = iter.get_marks()    # devuelve una lista de marcas

tag_list = iter.get_toggled_tags()  # devuelve una lista de etiquetas que están ↩
     activadas o desactivadas

tag_list = iter.get_tags()    # devuelve una lista de etiquetas por prioridades
```

13.4.5. Comprobar Condiciones en un Iterador de Texto

Las condiciones de marcado de en la posición de un iterador `TextIter` se pueden comprobar usando los siguientes métodos:

```
result = iter.begins_tag(tag=None)  # TRUE si la etiqueta está activada en el ↩
    iterador

result = iter.ends_tag(tag=None)  # TRUE si la etiqueta está desactivada en el ↩
    iterador

result = iter.toggles_tag(tag=None) # TRUE si la etiqueta está activa o ↩
    desactivada en el iterador

result = iter.has_tag(tag)    # TRUE si existe la etiqueta en el iterador
```

Estos métodos devuelven TRUE si la marca que representa el parámetro *tag* satisface la condición en las posición del iterador *iter*. En los tres primeros métodos, si la marca representada por el parámetro *tag* es None entonces el resultado es TRUE si cualquier etiqueta satisface la condición dada en la posición del iterador *iter*.

Los siguientes métodos indican si el texto en la posición del iterador TextIter es editable o permite inserción de texto:

```
result = iter.editable()

result = iter.can_insert(default_editability)
```

El método editable() indica si el *iter* está en un rango editable de texto mientras que el método can_insert() indica si el texto se puede insertar en el iterador *iter* considerando la editabilidad predeterminada de la TextView, el TextBuffer y las etiquetas aplicables. La editabilidad predeterminada *default_editability* se obtiene con el método:

```
default_editability = textview.get_editable()
```

La equivalencia de dos iteradores TextIter se puede determinar con el método:

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

Dos iteradores TextIter se pueden comparar con el método:

```
result = lhs.compare(rhs)
```

result será: -1 si *lhs* es menor que *rhs*; 0 si *lhs* es igual que *rhs*; y, 1 si *lhs* es mayor que *rhs*.

Para determinar si un iterador TextIter está situado entre otros dos iteradores TextIter se usa el método:

```
result = iter.in_range(start, end)
```

El *result* será TRUE si *iter* está entre *start* y *end*. Atención: *start* y *end* deben estar en orden ascendente. Esto se puede garantizar con el método:

```
first.order(second)
```

que reordenará los desplazamientos de los iteradores TextIter para que *first* esté antes que *second*.

13.4.6. Comprobar la posición en un Texto

La posición de un TextIter con respecto al texto en un TextBuffer se puede determinar con los siguientes métodos:

```
result = iter.starts_word()      # empieza palabra

result = iter.ends_word()        # termina palabra

result = iter.inside_word()      # dentro de palabra

result = iter.starts_sentence()  # empieza frase

result = iter.ends_sentence()    # termina frase

result = iter.inside_sentence()  # dentro de frase

result = starts_line()           # empieza línea

result = iter.ends_line()        # termina línea
```

result será TRUE si el TextIter está en la posición dada. Estos métodos son autoexplicativos. La definición de los elementos de texto y sus límites se determina por el lenguaje usado en el iterador TextIter. Obsérvese que una línea es una colección de frases similar a un párrafo.

Los siguientes métodos se pueden usar para determinar si un TextIter está al principio o al final de un TextBuffer:

```
result = iter.is_start()

result = iter.is_end()
```

result es TRUE si el iterador TextIter está al principio o al final del TextBuffer.

Ya que un TextBuffer puede contener múltiples caracteres que se visualizan en la práctica como una única posición del cursor (por ejemplo la combinación de retorno de carro y nueva línea o una letra con un símbolo de acento) es posible que un TextIter pueda estar en una posición que no corresponde con una posición de cursor. El siguiente método indica si un iterador TextIter está en una posición del cursor:

```
result = iter.is_cursor_position()
```

13.4.7. Movimiento a través del Texto

Los TextIters se pueden mover por un TextBuffer en saltos de varias unidades de texto. La definición de las unidades de texto se establece en el objeto PangoLanguage que se use en la posición del TextIter . Los métodos básicos son:

```
result = iter.forward_char()      # avanzar un caracter

result = iter.backward_char()     # retroceder un caracter

result = iter.forward_word_end()    # avanzar hasta el final de la palabra

result = iter.backward_word_start()   # retroceder al principio de la palabra

result = iter.forward_sentence_end()    # avanzar al final de la frase

result = iter.backward_sentence_start() # retroceder al principio de la frase

result = iter.forward_line()      # avanzar al principio de la línea

result = iter.backward_line()     # retroceder al principio de la línea

result = iter.forward_to_line_end()   # avanzar al final de la línea

result = iter.forward_cursor_position() # avanzar una posición del cursor

result = iter.backward_cursor_position()  # retroceder una posición del cursor
```

result es TRUE si el TextIter se movió y FALSE si el TextIter está al principio o al final del TextBuffer.

Todos estos métodos (excepto forward_to_line_end()) tienen métodos equivalentes que reciben una cantidad (que puede ser positiva o negativa) para mover el TextIter en un salto de múltiples unidades de texto:

```
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. Moverse a una Posición Determinada

Un iterador `TextIter` se puede mover a una posición específica en el TextBuffer haciendo uso de los siguientes métodos:

```
iter.set_offset(char_offset)    # moverse al desplazamiento de caracteres ↩
    específico

iter.set_line(line_number)    # moverse al principio de la línea dada como ↩
    parámetro

iter.set_line_offset(char_on_line)  # moverse al caracter especificado en la ↩
    línea actual

iter.forward_to_end()       # moverse al final del buffer
```

Además, un iterador `TextIter` se puede mover a una posición donde una etiqueta esté activada o desactivada usando los métodos:

```
result = iter.forward_to_tag_toggle(tag)

result = iter.backward_to_tag_taoggle(tag)
```

result es `TRUE` si el `TextIter` se movió a la nueva posición donde exista la etiqueta *tag*. Si la etiqueta *tag* es `None` entonces el iterador `TextIter` se moverá a la siguiente posición donde exista una etiqueta.

13.4.9. Búsqueda en el Texto

Una búsqueda de una cadena de texto en un TextBuffer se hace con los siguientes métodos:

```
match_start, match_end = iter.forward_search(str, flags, limit=None)  #  ↩
    búsqueda hacia adelante

match_start, match_end = iter.backward_search(str, flags, limit=None) #  ↩
    búsqueda hacia atrás
```

El valor de retorno es una tupla que contiene los iteradores `TextIter` que indican la posición del primer caracter que coincide con la búsqueda y la posición del primer caracter después de la coincidencia. *str* es la cadena a buscar. El parámetro *flags* modifica las condiciones de la búsqueda y puede tomar los siguientes valores:

```
gtk.TEXT_SEARCH_VISIBLE_ONLY    # se ignoran los caracteres invisibles

gtk.TEXT_SEARCH_TEXT_ONLY   # se ignoran los pixbuffers y los anclajes de hijos
```

limit es un parámetro opcional que limita el rango de la búsqueda.

13.5. Marcas de Texto

Una `TextMark` (Marca de Texto) indica una posición en un TextBuffer entre dos caracteres que se mantiene aunque se modifique el buffer. Las `TextMarks` se crean, se mueven y se borran usando los métodos del TextBuffer que se describen en la sección TextBuffer .

Un TextBuffer tiene dos marcas incluidas de serie llamadas: *insert* y *selection_bound* que se refieren al punto de inserción y el límite de la selección (puede que se refieran a la misma posición).

El nombre de una `TextMark` se puede obtener usando el método:

```
name = textmark.get_name()
```

Por defecto las marcas que no son *insert* no son visibles (esa marca se muestra como una barra vertical). La visibilidad de una marca se puede activar y obtener usando los métodos:

```
setting = textmark.get_visible()

textmark.set_visible(setting)
```

donde *setting* es TRUE si la marca es visible.
El TextBuffer que contiene una TextMark se puede recuperar usando el método:

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

Puedes determinar si una TextMark ha sido borrada usando el método:

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

La gravedad izquierda de una TextMark se puede recuperar usando el método:

```
setting = textmark.get_left_gravity()
```

La gravedad izquierda de una TextMark indica donde acabará la marca después de una inserción. Si la gravedad izquierda es TRUE la marca se pondrá a la izquierda de la inserción; si es FALSE, a la derecha de la inserción.

13.6. Etiquetas de Texto y Tablas de Etiquetas

Las etiquetas de texto TextTags especifican atributos que se pueden aplicar a un rango de texto en un buffer de texto. Cada buffer de texto TextBuffer tiene una tabla de etiquetas de texto TextTagTable que contiene las etiquetas de texto TextTags que se pueden aplicar dentro del buffer TextBuffer. Las tablas de etiquetas de texto TextTagTable se pueden usar en más de un buffer de texto para ofrecer estilos de texto consistentes.

13.6.1. Etiquetas de Texto

Las TextTags (Etiquetas de Texto) pueden tener nombre o ser anónimas. Una TextTag se crea usando la función:

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

Si el *name* (nombre) no se especifica o si es None la *tag* (etiqueta) será anónima. Las TextTags también se pueden crear usando el método de TextBuffer create_tag() que también te permite especificar los atributos y añade la etiqueta a la tabla de etiquetas del buffer (veáse la subsección TextBuffer).
Los atributos que pueden aparecer en una TextTag son:

name	Lectura / Escritura	Nombre de la etiqueta de texto. None si es anónima.
background	Escritura	Color de fondo como una cadena de texto
foreground	Escritura	Color de frente como una cadena de texto
background-gdk	Lectura / Escritura	Color de fondo como un GdkColor
foreground-gdk	Lectura / Escritura	Color de frente como un GdkColor
background-stipple	Lectura / Escritura	Bitmap a usar como una máscara cuando se dibuje el texto de fondo
foreground-stipple	Lectura / Escritura	Bitmap a usar como una máscara cuando se dibuje el texto de frente
font	Lectura / Escritura	Descripción de la fuente como una cadena de texto, por ejemplo, "Sans Italic 12"
font-desc	Lectura / Escritura	Descripción de la feunte como un objeto PangoFontDescription
family	Lectura / Escritura	Nombre de la familia de la fuente, por ejemplo, Sans, Helvetica, Times, Monospace

style	Lectura / Escritura	Estilo de la fuente como un PangoStyle, por ejemplo, pango.STYLE_ITALIC.
variant	Lectura / Escritura	Variante de la fuente como un PangoVariant, por ejemplo, pango.VARIANT_SMALL_CAPS.
weight	Lectura / Escritura	Peso de la fuente como un entero, mira los valores predefinidos en PangoWeight; por ejemplo, pango.WEIGHT_BOLD.
stretch	Lectura / Escritura	Estrechamiento de la fuente como un PangoStretch, por ejemplo, pango.STRETCH_CONDENSED.
size	Lectura / Escritura	Tamaño de fuente en unidades Pango.
size-points	Lectura / Escritura	Tamaño de fuente en puntos
scale	Lectura / Escritura	Tamaño de fuente como un factor de escala relativo al tamaño de fuente predeterminado. Esta propiedad se adapta a los cambios en el tema, etc, por tanto se recomienda su uso. Pango tiene algunos valores predefinidos tales como pango.SCALE_X_LARGE.
pixels-above-lines	Lectura / Escritura	Píxeles de espacio blanco por encima de los párrafos
pixels-below-lines	Lectura / Escritura	Píxeles de espacio blanco por debajo de los párrafos
pixels-inside-wrap	Lectura / Escritura	Píxeles de espacio blanco entre las líneas de un párrafo
editable	Lectura / Escritura	Si el texto puede modificarse por el usuario
wrap-mode	Lectura / Escritura	Si las líneas no se ajustan, se ajustan en limites de palabra o se ajustan en limites de caracteres
justification	Lectura / Escritura	Justificación izquierda, derecha o central
direction	Lectura / Escritura	Dirección del Texto, por ejemplo, derecha a izquierda o izquierda a derecha
left-margin	Lectura / Escritura	Ancho del márgen izquierdo en píxeles
indent	Lectura / Escritura	Cantidad de indentado para los párrafos, en píxeles
strikethrough	Lectura / Escritura	Si hay que tachar el texto
right-margin	Lectura / Escritura	Ancho del márgen derecho en píxeles
underline	Lectura / Escritura	Estilo de subrayado para este texto
rise	Lectura / Escritura	Desplazamiento del texto por encima de la línea base (por debajo de la línea base si es negativo) en píxeles
background-full-height	Lectura / Escritura	Si el color de fondo rellena la altura completa de la línea o sólo la altura de los caracteres marcados
language	Lectura / Escritura	El idioma en el que está el texto, como un código ISO. Pango puede usar esto como una ayuda para visualizar el texto. Si no entiendes este parámetro, probablemente no lo necesitas.
tabs	Lectura / Escritura	Tabulaciones personalizadas para el texto
invisible	Lectura / Escritura	Si el texto está oculto. No implementado en GTK 2.0

Se pueden establecer los atributos con este método:

```
tag.set_property(name, value)
```

Donde *name* es una cadena de texto que contiene el nombre de la propiedad y *value* es el valor que se le va a poner.

De la misma forma, el valor de un atributo se puede recuperar con el método:

```
value = tag.get_property(name)
```

Ya que la etiqueta no tiene un valor para cada atributo hay una serie de propiedades booleanas que indican si el atributo ha sido establecido:

background-set	Lectura / Escritura
foreground-set	Lectura / Escritura
background-stipple-set	Lectura / Escritura
foreground-stipple-set	Lectura / Escritura
family-set	Lectura / Escritura
style-set	Lectura / Escritura
variant-set	Lectura / Escritura
weight-set	Lectura / Escritura
stretch-set	Lectura / Escritura
size-set	Lectura / Escritura
scale-set	Lectura / Escritura
pixels-above-lines-set	Lectura / Escritura
pixels-below-lines-set	Lectura / Escritura
pixels-inside-wrap-set	Lectura / Escritura
editable-set	Lectura / Escritura
wrap-mode-set	Lectura / Escritura
justification-set	Lectura / Escritura
direction-set	Lectura / Escritura
left-margin-set	Lectura / Escritura
indent-set	Lectura / Escritura
strikethrough-set	Lectura / Escritura
right-margin-set	Lectura / Escritura
underline-set	Lectura / Escritura
rise-set	Lectura / Escritura
background-full-height-set	Lectura / Escritura
language-set	Lectura / Escritura
tabs-set	Lectura / Escritura
invisible-set	Lectura / Escritura

Por tanto, para obtener el atributo de una etiqueta, primero tienes que comprobar si el atributo ha sido establecido en la etiqueta. Por ejemplo, para obtener un valor correcto de justificación tienes que hacer algo así como:

```
if tag.get_property("justification-set"):
justification = tag.get_property("justification")
```

La prioridad predeterminada de una etiqueta es el orden en el que se añade a la TextTagTable. Las etiquetas con prioridad más alta tienen preferencia si hay múltiples etiquetas para establecer el mismo atributo para un rango de texto. La prioridad se puede obtener y fijar con los métodos:

```
priority = tag.get_priority()

tag.set_priority(priority)
```

La prioridad de una etiqueta debe estar entre 0 y uno menos del tamaño de la TextTagTable.

13.6.2. Tablas de Etiquetas de Texto

Una `TextTagTable` (Tabla de Etiquetas de Texto) se crea al crear un TextBuffer . También se puede crear una `TextTagTable` con la función:

```
table = TextTagTable()
```

Se puede añadir una TextTag (Etiqueta de Texto) a una `TextTagTable` usando el método:

```
table.add(tag)
```

La etiqueta *tag* no puede estar ya en la tabla y no puede tener el mismo nombre que otra etiqueta en la tabla.

Es posible buscar una etiqueta en una tabla con el método:

```
tag = table.lookup(name)
```

Este método devuelve la etiqueta *tag* en la tabla que tenga el nombre *name* o None si no hay ninguna etiqueta con ese nombre.

Se puede borrar una TextTag de una TextTagTable con el método:

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

El tamaño de la TextTagTable se puede consultar con el método:

```
size = table.get_size()
```

13.7. Un ejemplo de Vista de Texto

El programa de ejemplo testtext.py (que se deriva del programa testtext.c incluido en la distribución GTK+ 2.0.x) demuestra el uso del control TextView y sus objetos asociados: TextBuffers, TextIters, TextMarks, TextTags, TextTagTables. La figura Figura 13.2 ilustra su funcionamiento:

Figura 13.2 Ejemplo de Vista de Texto

El programa testtext.py define unas cuantas clases además de la clase principal `TestText`:

- La clase `Buffer` , en las líneas 99-496, es una subclase de la clase `gtk.TextBuffer`. Proporciona las capacidades de edición del buffer que usan los objetos `View`.
- La clase `View` , líneas 498-1126, es una subclase de la clase `gtk.Window` y contiene un objeto `gtk.TextView` que usa un objeto `Buffer` en lugar de un objeto `gtk.TextBuffer`. Proporciona una ventana y visualiza los contenidos de un objeto `Buffer` además de una barra de menú.
- La clase `FileSel` , líneas 73-97, es una subclase de la clase `gtk.FileSelection` que proporciona una selección de ficheros para los contenidos de la clase `Buffer` .
- La clase `Stack` proporciona objetos de pilas simples.

La ventana de ciclo de color se implementa usando etiquetas de texto que se aplican a una sección del texto en un buffer. Las líneas 109-115 (en el método __init__()) crean estas etiquetas y las líneas 763-784 (en el método do_apply_colors()) aplican las etiquetas de colores a una sección del texto de dos en dos caracteres. Las lienas 202-239 proporcionan los métodos (color_cycle_timeout(), set_colors()

y cycle_colors()) que producen el ciclo de color cuando está activado. El ciclo de color se activa estableciendo (línea 220) la propiedad *foreground_gdk* de las etiquetas individuales *color_tags* (que también establecen la propiedad *foreground_set*). El ciclo de color se desactiva poniendo la propiedad *foreground_set* a FALSE (línea 222). Los colores se cambian periodicamente al desplazar el tono del color (*start_hue* (line 237))

Un nuevo Buffer se rellena con un contenido de ejemplo cuando se selecciona el menú Test → Example (el método fill_example_buffer() en las líneas 302-372). El buffer de ejemplo contiene texto de varios colores, estilos, idiomas y pixbuffers. El método init_tags() (líneas 260-300) establece una variedad de TextTags para usarlas con el texto de ejemplo. La señal de evento de estas etiquetas se conecta al método tag_event_handler() (líneas 241-256) para ilustrar la captura de los eventos de botón y movimiento.

El modo de ajuste del control TextView está puesto a WRAP_WORD (línea 580) y las ventanas de borde del TextView se visualizan al establecer su tamaño en las líneas 587-588 y las líneas 596-597. Las ventanas del borde derecho e izquierdo se usan para mostrar los números de línea y las ventanas de borde superior e inferior muestran las posiciones de tabulación cuando se utilizan tabulaciones personalizadas. Las ventanas de borde se actualizan cuando se recibe la señal "expose-event" en el TextView (líneas 590 y 599) El método line_numbers_expose() (líneas 1079-1116) determina si las ventanas de borde izquierdo o derecho tienen un evento de exposición y, si es así, calculan el tamaño del área de exposición. Entonces, la localización del principio de la línea y el número de línea para cada línea en el área de exposición se calcula en el método get_lines() (líneas 1057-1077). Los números de línea se dibujan en la ventana de borde en la posición (transformada por la línea 1109).

Las posiciones personalizadas de tabulación se visualizan en las ventanas de borde superior e inferior de una forma similar (líneas 1013-1055). Sólo se visualizan cuando el cursor se mueve dentro de un rango de texto que tiene el atributo de tabulaciones propias. Esto se detecta manejando la señal "mark-set" en el método cursor_set_handler() (líneas 999-1011) e invalidando las ventanas de borde superior e inferior si la marca a activar es la marca *insert*.

Los controles móviles se añaden al objeto View con el método do_add_children() (líneas 892-899) el cual llama al método add_movable_children() (líneas 874-890). Los hijos son gtk.Labels (Etiquetas) que pueden arrastrarse por las ventanas que forman parte de un control TextView .

De la misma forma, los controles se añaden a las ventanas del TextView de una View y el Buffer usando el método do_add_focus_children() (líneas 901-949).

Capítulo 14

Control de Vista de Árbol (TreeView)

El control `TreeView` (control de vista de árbol) muestra listas y árboles de múltiples columnas. Sustituye los antiguos controles List, `CList`, `Tree` y `CTree` por un conjunto de objetos mucho más potente y flexible que utilizan el patrón Modelo-Vista-Controlador (MVC) para proporcionar las siguientes funcionalidades:

- dos modelos predefinidos: uno para listas y otro para árboles
- las múltiples vistas de un modelo son automáticamente actualizadas cuando se producen cambios en éste
- visualización selectiva de los datos del modelo
- uso de datos del modelo para personalizar la visualización del control en cada fila
- objetos de representación de datos predefinidos para mostrar texto, imágenes y datos booleanos
- modelos acumulables para obtener vistas ordenadas o filtradas del modelo de datos subyacente
- columnas que permiten su reordenación y redimensionado
- ordenación automática haciendo click en los encabezados de las columnas
- soporte para arrastrar y soltar
- soporte para modelos personalizados escritos totalmente en Python
- soporte para intérpretes de celdas (cell renderers) personalizados escritos completamente en Python

Obviamente, todas estas posibilidades conllevan el coste de un conjunto de objetos e interfaces significativamente más complejo y que puede llegar a parecer intimidatorio inicialmente. En el resto del capítulo exploraremos los objetos e interfaces de la vista de árbol (TreeView) para comprender sus usos habituales. Los usos más esotéricos tendrán que ser investigados por el lector.

Comenzaremos con una rápida visión general de objetos e interfaces para posteriormente bucear en la interfaz de `TreeModel` y en las clases predefinidas `ListStore` y `TreeStore`.

14.1. Introducción

El control `TreeView` (vista de árbol) es el objeto de interfaz de usuario que muestra los datos almacenados en un objeto que implemente la interfaz `TreeModel`. En PyGTK 2.0 se proporcionan dos clases base para modelos de árbol.:

- `TreeStore` (almacén de datos en árbol), que proporcionan almacenamiento para datos jerárquicos, organizándolos como filas de un árbol y con los datos en columnas. Cada fila del árbol puede tener cero o más filas hijas, aunque todas las filas deben tener el mismo número de columnas.

- `ListStore` (almacén de datos en lista), que proporciona almacenamiento para datos tabulares, con una organización en filas y columnas, de una manera similar a la tabla de una base de datos relacional. Un `ListStore` (almacén de datos en lista) realmente consiste en una versión simplificada de un `TreeStore` (almacén de datos en árbol), cuyas filas no tienen descendientes. Fue creada para proporcionar una interfaz más sencilla (y presuntamente más eficiente) a un modelo de datos tan común. Y,

estos dos modelos adicionales se superponen (o se interponen) a los modelos base:

- `TreeModelSort`, que proporciona un modelo en el que los datos del modelo de árbol subyacente se mantienen ordenados. Y,
- `TreeModelFilter`, que proporciona un modelo que contiene un subconjunto de los datos del modelo subyacente. Este modelo únicamente está disponible desde la versión de PyGTK 2.4 y posteriores.

Un `TreeView` (vista de árbol) muestra todas las filas que contiene un `TreeModel`, pero permite mostrar selectivamente algunas de las columnas. También es posible presentar las columnas en un orden diferente al de almacenamiento en el `TreeModel`.

Un `TreeView` usa objetos del tipo `TreeViewColumn` (columnas de vista de árbol) para organizar la visualización de los datos en columnas. Cada `TreeViewColumn` muestra una columna con un encabezado opcional que puede contener los datos de diversas columnas de un `TreeModel`. Los objetos `TreeViewColumn` se empaquetan (de forma semejante a los contenedores `HBox`) con objetos `CellRenderer` (intérpretes de celda), para generar la visualización de los datos asociados situados en una fila y columna de un `TreeModel`. Existen tres clases de `CellRenderer` predefinidas:

- `CellRendererPixbuf`, que muestra una imagen del tipo pixbuf en las celdas de un `TreeViewColumn`.
- `CellRendererText`, que muestra una cadena en las celdas de un `TreeViewColumn`. Si es necesario convierte los datos de la columna a cadenas de texto. Por ejemplo, si se mostrase una columna de datos consistente en números en coma flotante, el `CellRendererText` los convertiría en cadenas de texto antes de mostrarlos.
- `CellRendererToggle`, que muestra un valor booleano como un botón conmutable en las celdas de un `TreeViewColumn`.

Una `TreeViewColumn` puede contener varios objetos `CellRenderer` para proporcionar una columna que, por ejemplo, pueda contener una imagen y un texto juntos.

Finalmente, los objetos `TreeIter` (iterador de árbol), `TreeRowReference` (referencia de fila de árbol) y `TreeSelection` (selección de árbol) proporcionan un puntero transitorio a la fila de un `TreeModel`, un puntero persistente a la fila de un `TreeModel` y un objeto que gestiona una selección en una `TreeView`.

La visualización de un `TreeView` se compone de las siguientes operaciones generales (aunque no necesariamente en este orden):

- Se crea un objeto de modelo de árbol (`TreeModel`), generalmente un `ListStore` o un `TreeStore` con una o más columnas de un tipo de datos determinado.
- El modelo de árbol puede entonces llenarse con una o más filas de datos.
- Se crea un control `TreeView` y se asocia al modelo de árbol.
- Se crean una o más columnas `TreeViewColumn` y se insertan en el `TreeView`. Cada una de ellas resultará en la visualización de una columna.
- Para cada `TreeViewColumn` se crea uno o más intérpretes de celda `CellRenderer`, que son añadidos a cada `TreeViewColumn`.
- Se fijan los atributos de cada `CellRenderer` para indicar de qué columnas del modelo de árbol han de tomar los datos del atributo, como por ejemplo, el texto que se ha de mostrar. Esto permite que los `CellRenderer` muestren las columnas de cada fila de forma distinta.
- El `TreeView` se inserta y es mostrado en una ventana (`Window`) o una ventana con barras de desplazamiento (`ScrolledWindow`).

- Los datos del modelo de árbol se manipulan programáticamente en respuesta a las acciones de los usuarios y usuarias. El `TreeView` rastreará de forma automática los cambios.

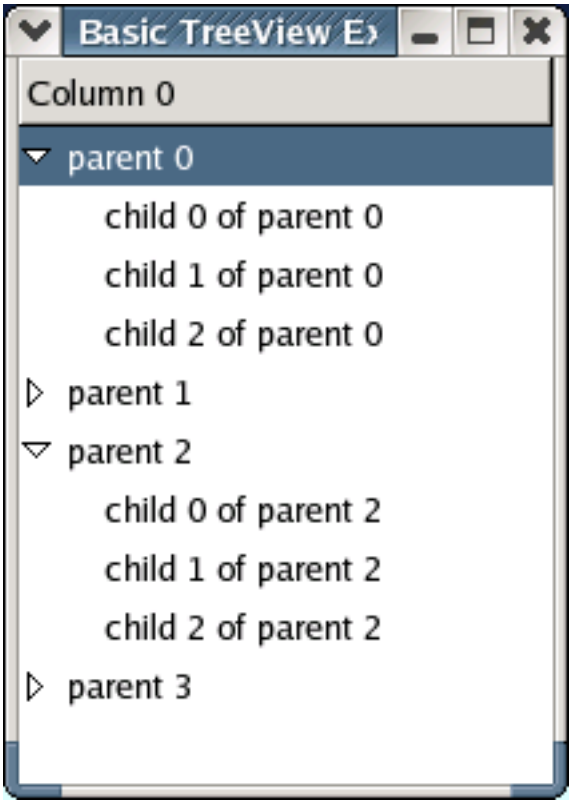
El programa de ejemplo basictreeview.py ilustra la creación y visualización de un objeto sencillo `TreeView`:

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

# example basictreeview.py

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

class BasicTreeViewExample:

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

```
60
61          # Allow drag and drop reordering of rows
62          self.treeview.set_reorderable(True)
63
64          self.window.add(self.treeview)
65
66          self.window.show_all()
67
68  def main():
69      gtk.main()
70
71  if __name__ == "__main__":
72      tvexample = BasicTreeViewExample()
73      main()
```

En programas reales el `TreeStore` probablemente se llenaría de datos después de que la visualización del `TreeView` se produjese por una acción del usuario. Nos detendremos en los detalles de las interfaces de `TreeView` en secciones posteriores. Figura 14.1 muestra la ventana creada por el programa basictreeview.py tras la expanxión de un par de filas madre.

Figura 14.1 Programa elemental de ejemplo de TreeView



Ahora examinemos la interfaz de `TreeModel` y los modelos que la implementan.

14.2. La Interfaz y Almacén de Datos TreeModel

14.2.1. Introducción

La interfaz `TreeModel` es implementada por todas las subclases de `TreeModel` y aporta métodos para:

- la obtención de las características del conjunto de datos que almacena, tales como el número de columnas y el tipo de datos de éstas.
- la obtención de un iterador (`TreeIter`), una referencia temporal que apunta a una fila del modelo.

- la obtención de información sobre un nodo (o fila), tal como el número de nodos que descienden de él, una lista de éstos nodos hijos, el contenido de sus columnas y un puntero al nodo padre.
- informar de los cambios que se produzcan elos datos del modelo (`TreeModel`).

14.2.2. Creación de Objetos TreeStore (árbol) y ListStore (lista)

Las clases base para el almacenamiento de datos `ListStore` (almacén de datos en lista) y `TreeStore` (almacén de datos en árbol) permiten definir y gestionar las filas y columnas de datos existentes en un modelo de árbol. Los constructores de ambos objetos precisan que el tipo de cada una de las columnas sea alguno de los siguientes:

- tipos básicos de Python, tal como: int, str, long, float y object.
- tipos de PyGTK tales como `Button`, `VBox`, `gdk.Rectangle`, `gdk.Pixbuf`, etc.
- tipos `GObject` (GTypes de GTK+) especificados bien como constantes GObject Type o como cadenas. La mayoría de GTypes son traducidos a un tipo de Python:
  - gobject.TYPE_CHAR o 'gchar'
  - gobject.TYPE_UCHAR o 'guchar'
  - gobject.TYPE_BOOLEAN o 'gboolean'
  - gobject.TYPE_INT o 'gint'
  - gobject.TYPE_UINT o 'guint'
  - gobject.TYPE_LONG o 'glong
  - gobject.TYPE_ULONG o 'gulong
  - gobject.TYPE_INT64 o 'gint64'
  - gobject.TYPE_UINT64 o 'guint64'
  - gobject.TYPE_FLOAT o 'gfloat'
  - gobject.TYPE_DOUBLE o 'gdouble'
  - gobject.TYPE_STRING o 'gchararray'
  - gobject.TYPE_OBJECT o 'GObject

Por ejemplo, para crear un `ListStore` o `TreeStore` cuyas columnas contuviesen un `gdk.Pixbuf` (un tipo de imagen), un entero, una cadena y un tipo booleano habría que hacier algo así:

```
liststore = ListStore(gtk.gdk.Pixbuf, int, str, 'gboolean')

treestore = TreeStore(gtk.gdk.Pixbuf, int, str, 'gboolean')
```

Una vez que se crea un almacén de datos en Lista (`ListStore`) o un almacén de datos en árbol (`TreeStore`) y sus columnas han sido definidas, ya no es posible hacer en ellos cambios o modificaciones. Así mismo es importante tener en cuenta que no existe una relación predefinida entre las columnas de una vista de árbol (`TreeView`) y las columnas de su correspondiente modelo (`TreeModel`). Es decir, la quinta columna de datos de un modelo (`TreeModel`) puede mostrarse en la primera columna de una determinada vista de árbol (`TreeView`) y en la tercera columna en otra vista diferente. Por tanto, no es necesario preocuparse de cómo se mostrarán los datos a la hora de crear los almacenes de datos (Stores).

Si estos dos tipos de almacenes de datos no se ajustan a las necesidades de una aplicación es posible definir otros almacenes de datos personalizados en Python siempre que éstos implementen la interfaz TreeModel. Retomaremos esta cuestión más adelante en la sección dedicada a GenericTreeModel .

14.2.3. Cómo referirse a las filas de un modelo TreeModel

Antes de que podamos hablar de la gestion de las filas de datos de un almacén `TreeStore` o `ListStore` necesitamos una forma de especificar a qué columna nos queremos referir. PyGTK dispone de tres formas de indicar columnas de un modelo `TreeModel` rows: un camino de árbol, un iterador `TreeIter` y una referencia `TreeRowReference`.

14.2.3.1. Caminos de árbol (Tree Paths)

Un camino de árbol es una representación mediante enteros, una cadena o una tupla de la localización de una fila en un almacén de datos. Un valor entero especifica la fila del nivel superior del almacén, empezando por 0. Por ejemplo, un valor de camino de 4 especificaría la quinta fila del almacén de datos. De la misma manera, una representación mediante una cadena resultaría "4" y la representación como tupla (4,). Esto es suficiente para llegar a determinar cualquier fila en un almacén de datos en lista (`ListStore`), pero en el caso de un árbol (`TreeStore`) necesitamos poder indicar las filas hijas. Para este caso debemos usar la representación mediante una cadena o una tupla.

Dado que un almacén en árbol (`TreeStore`) puede tener una jerarquía de una profudidad arbitraria, la representación como cadena especifica el camino desde el nivel más alto hasta la fila escogida utilizando enteros separados mediante el carácter ":". De forma similar, la representación mediante una tupla especifica el camino en el árbol empezando desde el vértice hasta la fila como una secuencia de enteros. Como ejemplos correctos de representaciones con cadenas de caminos tenemos: "0:2" (especifica la fila que es la tercera hija de la primera fila) y "4:0:1" (especifica la fila que es la segunda hija del primer hijo de la quinta fila). De forma semejante, los mismos caminos son representados respectivamente por las tuplas (0, 2) y (4, 0, 1).

Un camino de árbol es la única manera en la que se puede hacer corresponder una fila de una vista de árbol (`TreeView`) a una fila de un modelo (`TreeModel`) debido a que los caminos de una y otra son iguales. También existen otros problemas con los caminos de árbol:

- un camino de árbol puede especificar una fila que no existe en el almacén ListStore o TreeStore.
- un camino de árbol puede apuntar a datos distintos tras insertar o borrar una fila en un `ListStore` o `TreeStore`.

PyGTK usa la representación como tupla al devolver caminos, pero acepta cualquiera de las tres formas de representar un camino. Por coherencia se debería usar la representación en forma de tupla.

Se puede obtener un camino de árbol a partir de un iterador `TreeIter` utilizando el método get_path():

```
path = store.get_path(iter)
```

donde *`iter`* es un iterador `TreeIter` que apunta a una fila en el almacén y *`path`* es la tupla que representa el camino a la fila.

14.2.3.2. Iteradores TreeIter

Un iterador `TreeIter` es un objeto que aporta una referencia transitoria a la fila de un almacén `ListStore` o `TreeStore`. Si los contenidos del almacén cambian (generalmente porque se ha añadido o quitado una fila) el iterador `TreeIter` puede no ser ya válido. Un modelo `TreeModel` que dé soporte a iteradores persistentes debe establecer la bandera `gtk.TREE_MODEL_ITERS_PERSIST`. Una aplicación puede comprobar dicha bandera haciendo uso del método get_flags().

Los iteradore `TreeIter` se crean utilizando alguno de los métodos del modelo `TreeModel`, que son aplicables tanto a objetos del tipo `TreeStore` como `ListStore`:

```
treeiter = store.get_iter(path)
```

donde *`treeiter`* apunta a la fila del camino *`path`*. La excepción ValueError se activa en el caso de que el camino no sea válido.

```
treeiter = store.get_iter_first()
```

donde *`treeiter`* es un iterador TreeIter que apunta a la fila en el camino (0,). *`treeiter`* será `None` si el almacén está vacío.

```
treeiter = store.iter_next(iter)
```

donde *`treeiter`* es un iterador `TreeIter` que apunta a la siguiente fila en el mismo nivel que el iterador `TreeIter` especificado por *`iter`*. *`treeiter`* tendrá el valor `None` si no hay una fila siguiente (*`iter`* también es invalidado).

Los siguientes métodos son de utilidad únicamente cuando se vaya a obtener un iterador `TreeIter` a partir de un almacén de datos de árbol (`TreeStore`):

```
treeiter = treestore.iter_children(parent)
```

donde *treeiter* es un iterador TreeIter que apunta a la primera fija que desciende de la fila indicada por el iterador TreeIter indicado por *parent*. *treeiter* será None en caso de que no tenga descendientes.

```
treeiter = treestore.iter_nth_child(parent, n)
```

donde *treeiter* es un iterador TreeIter que señala la fila hija (de índice *n*) de la fila especificada por el iterador TreeIter *parent*. *parent* puede ser None para obtener una fila del nivel superior . *treeiter* será None si no hay descendientes.

```
treeiter = treestore.iter_parent(child)
```

donde *treeiter* es un TreeIter que apunta a la fila madre de la fila especificada por el iterador TreeIter *child*. treeiter será None si no hay descendientes.

Se puede obtener un camino a partir de un iterador TreeIter usando el método get_path() :

```
path = store.get_path(iter)
```

donde *iter* es un iterador Treeiter que apunta a una fila del almacén, y *path* es su camino expresado como tupla.

14.2.3.3. Referencias persistentes a filas (TreeRowReferences)

Una referencia del tipo TreeRowReference es una referencia persistente a una fila de datos en un almacén. Mientras que el camino (es decir, su localización) de una fila puede cambiar a medida que se añden o quitan filas al almacén, una referencia del tipo TreeRowReference apuntará a la misma fila de datos en tanto exista.

NOTA

Las referencias TreeRowReference únicamente están disponibles a partir de la versión 2.4 de PyGTK.

Se puede crear una referencia del tipo TreeRowReference utilizando su constructor:

```
treerowref = TreeRowReference(model, path)
```

donde *model* es el modelo TreeModel que contiene la fila y *path* es el camino de la fila que hay que referenciar. Si *path* no es un camino válido para el modelo *model* entonces se devuelve None.

14.2.4. Adición de filas

14.2.4.1. Adición de filas a un almacén de datos del tipo ListStore

Una vez que se ha creado un almacén del tipo ListStore se han de añadir filas utilizando alguno de los métodos siguientes:

```
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)
```

Cada uno de estos métodos insertan una fila en una posición implícita o especificada en el almacén ListStore. Los métodos append() y prepend() usan posiciones implícitas: tras la última fila y antes de la primera fila, respectivamente. El método insert() requiere un entero (el parámetro *position*) que especifica la localización en la que se insertará la fila. Los otros dos métodos necesitan un iterador TreeIter (*sibling*) que hace referencia a una fila en el almacén ListStore ante o tras la cual se insertará la fila.

El parámetro *row* especifica los datos que deberían ser insertados en la fila tras su creación. Si *row* es None o no se especifica, se crea una fila vacía. Si *row* se especifica debe ser una tupla o una lista que contenga tantos elementos como el número de columnas que posea el almacén ListStore. Los

elementos también deben coincidir con los tipos de las correspondientes columnas del almacén `ListStore`.

Todos los métodos devuelven un iterador `TreeIter` que apunta a la fila recién insertada. El siguiente fragmento de código ilustra la creación de un almacén `ListStore` y la adición de filas de datos en él:

```
...
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. Adición de filas a un almacén de datos del tipo TreeStore

La adición de filas a un almacén del tipo `TreeStore` es semejante a la operación sobre el tipo `ListStore`, salvo que también se debe especificar una fila madre (usando un iterador `TreeIter`) a la que añadir la nueva fila. Los métodos para el amacén `TreeStore` son:

```
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)
```

Si *parent* es `None`, la fila se añadirá a las filas del nivel superior.

Cada uno de estos métodos inserta una fila en una posición implícita o especifica da en el almacén `TreeStore`. Los métodos append() y prepend() usan posiciones implícitas: tras la última fila hija y antes de la primera fila hija, respectivamente. El método insert() requiere un entero (el parámetro *position*), que especifica el lugar en el que se insertará la fila hija. Los otros dos métodos necesitan un iterador `TreeIter` (*sibling*), que hace referencia a una fila hija en el almacén `TreeStore` ante o tras la que se insertará la fila.

El parámetro *row* especifica qué datos se deben insertar en la fila tras su creación. Si *row* es `None` o no se especifica, entonces se crea una fila vacía. Si *row* se especifica, debe ser una tupla o una lista que contenga tantos elementos como el número de columnas que posea el almacén `TreeStore`. Los elementos también deben coincidir en sus tipos con los de las correspondientes columnas del almacén `TreeStore`.

Todos los métodos devuelven un iterador `TreeIter` que apunta a la recién creada fila. El siguiente fragmento de código ilustra la creación de un `TreeStore` y la adición de filas de datos al mismo:

```
...
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. Almacenes de datos de gran tamaño

Cuando los almacenes `ListStore` o `TreeStore` contienen un gran número de filas de datos la adición de nuevas filas puede llegar a ser muy lenta. Hay un par de cosas que se pueden hacer para mitigar este problema:

- En el caso de añadir un gran número de filas, desconectar el modelo (`TreeModel`) de su vista (`TreeView`) usando el método set_model() con el parámetro *model* puesto a `None` evitar la actualización de la vista (`TreeView`) con cada inserción.

- Igualmente útil puede ser desactivar la ordenación al insertar un gran número de filas. Para ello se usa el método set_default_sort_func() con el parámetro *sort_func* puesto a None.
- Limitar el número de referencias del tipo TreeRowReference en uso también es importante, puesto que se actualiza su camino con cada adición o emilinación de filas.
- Fijar la propiedad de la vista (TreeView) "fixed-height-mode" a TRUE, de forma que todas las filas tengan la misma altura y se evite el cálculo individual de la altura de cada fila. Esta opción solamente está disponible a partir de la versión 2.4 de PyGTK.

14.2.5. Eliminación de Filas

14.2.5.1. Eliminación de filas de un almacén ListStore

Es posible eliminar una fila de datos de un almacén ListStore usando el método remove() :

```
treeiter = liststore.remove(iter)
```

donde *iter* es un iterador TreeIter que apunta a la fila que se ha de eliminar. El iterador devuelto TreeIter (*treeiter*) apunta a la siguiente fila o no es válido si *iter* apunta a la última fila.

El método clear() elimina todas las filas del almacén ListStore:

```
liststore.clear()
```

14.2.5.2. Eliminación de filas de un almacén TreeStore

Los métodos que sirven para eliminar filas de un almacén TreeStore son similares a los métodos equivalentes del almacén ListStore:

```
result = treestore.remove(iter)
treestore.clear()
```

donde *result* es TRUE si la fila ha sido eliminada e *iter* apunta a la siguiente fila válida. En otro caso, *result* es FALSE e *iter* es invalidado.

14.2.6. Gestión de los datos de las filas

14.2.6.1. Establecemiento y obtención de los valores de los datos

Los métodos para acceder a los valores de los datos en los almacenes ListStore y TreeStore tienen el mismo formato. Todas las manipulaciones de datos en almacenes usan un iterador TreeIter que especifica la fila con la que se trabaja. Una vez obtenido, dicho iterador (TreeIter) puede usarse para obtener los valores de la columna de una fila utilizando el método get_value() :

```
value = store.get_value(iter, column)
```

donde *iter* es un iterador TreeIter que apunta a una fila, *column* es un número de columna en el almacén *store*, y, *value* es el valor guardado en la posición fila-columna señalada.

Para obtener los valores de varias columnas en una única llamada se debe usar el método get() :

```
values = store.get(iter, column, ...)
```

donde *iter* es un iterador TreeIter que apunta a una fila, *column* es un número de columna en el almacén *store*, y, ... representa cero o más números de columna adicionales y *values* es una tupla que contiene los valores obtenidos. Por ejemplo, para obtener los valores de las columnas 0 y 2:

```
val0, val2 = store.get(iter, 0, 2)
```

NOTA

El método get() solamente está disponible a partir de la versión 2.4 de PyGTK.

Para establecer el valor de una columna se emplea el método set_value() :

```
store.set_value(iter, column, value)
```

donde *iter* (un iterador TreeIter) y *column* (un entero) especifican la posición fila-columna en el almacén *store* y *column* es el número de columna cuyo valor *value* debe ser fijado. *value* debe ser del mismo tipo de dato que la columna del almacén *store*.

En el caso de que se desee establecer de una sola vez el valor de más de una columna en una fila se debe usar el método set() :

```
store.set(iter, ...)
```

donde *iter* especifica la fila del almacén y ... es uno o más pares de número de columna - valor que indican la columna y el valor que se va a fijar. Por ejemplo, la siguiente llamada:

```
store.set(iter, 0, 'Foo', 5, 'Bar', 1, 123)
```

establece el valor de la primera columna como 'Foo', el de la sexta columna como 'Bar' y el de la segunda columna como 123, en la fila del almacén *store* especificada por el iterador *iter*.

14.2.6.2. Reorganización de filas en almacenes ListStore

Se pueden mover individualmente las filas de un almacén ListStore usando alguno de los siguientes métodos disponibles desde la versión 2.2 de PyGTK:

```
liststore.swap(a, b)
liststore.move_after(iter, position)
liststore.move_before(iter, position)
```

swap() permuta la posición de las filas a las que hacen referencia los iteradores TreeIter *a* y *b*. move_after() y move_before() mueven la fila señalada por el iterador (TreeIter) *iter* a una posición anterior o posterior a la fila indicada por el iterador (TreeIter) *position*. Si *position* tiene el valor None, move_after() situará la fila al principio principio del almacén, mientras que move_before() lo hará al final del mismo.

En caso de que se desee reorganizar completamente las filas de datos de un almacén ListStore entonces es preferible usar el siguiente método:

```
liststore.reorder(new_order)
```

donde *new_order* es una lista de enteros que especifican el nuevo orden de filas de la siguiente manera:

```
new_order[nuevaposición] = antiguaposición
```

Por ejemplo, si *liststore* contuviese cuatro filas:

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

La llamada al método:

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

produciría el siguiente orden:

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

NOTA

Estos métodos únicamente reorganizarán almacenes ListStore no ordenados.

En el caso de querer reorganizar las filas de datos en la versión 2.0 de PyGTK es necesario recurrir a la eliminación e inserción de filas utilizando los métodos descritos en las secciones Adición de filas y Eliminación de filas.

14.2.6.3. Reorganización de filas en almacenes TreeStore

Los métodos que se usan para reorganizar las filas de un almacén `TreeStore` son semejantes a los utilizados con los almacenes del tipo `ListStore`, exceptuando que los primeros únicamente afectan a las filas hijas de una determinada fila madre. Es decir, no es posible, por ejemplo, intercambiar filas con filas madre distintas.:

```
treestore.swap(a, b)
treestore.move_after(iter, position)
treestore.move_before(iter, position)
```

swap() intercambia las posiciones de las filas hija indicadas por los iteradores (TreeIter) *a* y *b*. *a* y *b* deben compartir su fila madre. move_after() y move_before() mueven la fila indicada por el iterador (`TreeIter`) *iter* a una posición posterior o anterior a la fila señalada por el iterador (`TreeIter`) *position*. *iter* y *position* deben compartir su fila madre. Si *position* tiene el valor `None`, move_after() situará la fila al principio del almacén, mientras que el método move_before() lo hará al final del mismo.

El método reorder() precisa un parámetro adicional que especifique la fila madre cuyas filas hijas serán reordenadas:

```
treestore.reorder(parent, new_order)
```

donde *new_order* es una lista de enteros que especifican el nuevo orden de filas hijas de la fila madre especificada por el iterador (`TreeIter`) *parent* de la siguiente manera:

```
new_order[nuevaposición] = antiguaposición
```

Por ejemplo, si *treestore* contuviese estas cuatro filas:

```
'parent'
    'one'
    'two'
    'three'
    'four'
```

La llamada al método:

```
treestore.reorder(parent, [2, 1, 3, 0])
```

produciría el siguiente orden:

```
'parent'
    'three'
    'two'
    'four'
    'one'
```

NOTA

Estos métodos únicamente reorganizarán almacenes `TreeStore` no ordenados.

14.2.6.4. Gestión de múltiples filas

Uno de los aspectos más problemáticos de la manipulación de almacenes `ListStore` y `TreeStore` es el trabajo con múltiples filas, como por ejemplo, mover varias filas de una fila madre a otra, o eliminar un conjunto de filas en función de determinados criterios. La dificultad surge de la necesidad de

utilizar un iterador `TreeIter` que puede no ser ya válido como resultado de la operación que se realice. Es posible comprobar si los iteradores de unos almacenes `ListStore` y `TreeStore` con persistentes utilizando el método get_flags() y contrastando la existencia de la bandera `gtk.TREE_MODEL_ITERS_PERSIST`. Sin embargo, las clases apilables del tipo `TreeModelFilter` y `TreeModelSort` no poseen iteradores `TreeIter` persistentes.

Si aceptamos que los iteradores `TreeIter` no son persistentes, ¿cómo movemos todas las filas hijas de una dada a otra?. Para ello tenemos que:

- iterar sobre la descendencia de la fila madre
- obtener los datos de cada fila
- eliminar cada fila hija
- insertar una nueva fila con los datos de la antigua fila en la lista de la nueva fila madre

No podemos confiar en que el método remove() devuelva un iterador `TreeIter` correcto, por lo que simplemente solicitaremos el iterador del primer descendiente hasta que devuelva `None`. Una función posible para mover filas hijas sería:

```
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
```

La función anterior abarca el caso sencillo en el que se mueven todas las filas hijas de una única fila madre, pero ¿qué ocurre si se desea elminar todas las filas del almacén `TreeStore` en función de determinado criterio, por ejemplo, el valor de la primera columna?. A primera vista, se podría pensar en la posibilidad de usar el método foreach(), de manera que se itere sobre todas las filas y entonces se eliminarían las que cumpliesen el criterio elegido:

```
store.foreach(func, user_data)
```

donde *`func`* es una función que es llamada por cada fila y tiene la siguiente signatura:

```
def func(model, path, iter, user_data):
```

donde *`model`* es el almacén de datos `TreeModel`, *`path`* es el camino de una fila en el modelo *`model`*, *`iter`* es un iterador `TreeIter` que apunta al camino *`path`* y *`user_data`* son los datos aportados. Si *`func`* devuelve `TRUE` entonces el método foreach() dejará de iterar y retornará.

El problema con este enfoque es que al cambiar los contenidos del almacén mientras se produce la iteración del método foreach() puede dar lugar a resultados impredecibles. El uso del método foreach() para crear y guardar referencias persistentes TreeRowReferences de las filas que van a ser eliminadas, para posteriormente eliminar estas referencias tras la finalización del método foreach() podría ser una estrategia adecuada, salvo que no funcionaría en las versiones 2.0 y 2.2 de PyGTK, donde no están disponibles las referencias persistentes del tipo `TreeRowReference`.

Una estrategia fiable que abarca todas las variantes de PyGTK consiste en llamar al método foreach() para reunir los caminos de las filas que se van a eliminar y luego proceder a eliminar dichas filas en orden inverso, de forma que se preserve la validez de los caminos en el árbol. El siguiente fragmento de código ejemplifica esta estrategia:

```
...
# esta función selecciona una fila si el valor de la primera columna es >= que ↩
    el valor de comparación
# data es una tupla que contiene el valor de comparación y una lista para ↩
    guardar los caminos
def match_value_cb(model, path, iter, data):
  if model.get_value(iter, 0) >= data[0]:
    data[1].append(path)
  return False     # mantiene la iteración de foreach
```

```
pathlist = []
treestore.foreach(match_value_cb, (10, pathlist))

# foreach funciona en orden de llegada (FIFO)
pathlist.reverse()
for path in pathlist:
  treestore.remove(treestore.get_iter(path))
...
```

En el caso de que se quisiese buscar en un almacén `TreeStore` la primera fila que cumpliese determinado criterio, probablemente sería preferible llevar a cabo personalmente la iteración con algo similar a:

```
treestore = TreeStore(str)
...
def match_func(model, iter, data):
    column, key = data # data es una tupla que contiene número de columna, ↩
        clave (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'))
```

La función de búsqueda `search`() itera recursivamente sobre la fila (especificada por el iterador *iter*) y sus descendientes y sus fijas hijas en orden de llegada, en búsqueda de una fila que tenga una columna cuyo valor coincida con la cadena clave dada. La búsqueda termina al encontrarse una fila.

14.2.7. Soporte del protocolo de Python

Las clases que implementan la interfaz `TreeModel` (`TreeStore` y `ListStore` y en PyGTK 2.4 también `TreeModelSort` y `TreeModelFilter`) también soportan los protocolos de Python de mapeado e iteración (protocolos mapping e iterator). El protocolo de iterador permite usar la función de Python `iter`() sobre un `TreeModel` o crear un iterador que sirva para iterar sobre todas las filas superiores del `TreeModel`. Una característica más útil es la de iterar utilizando la orden `for` o la comprensión de listas. Por ejemplo:

```
...
liststore = gtk.ListStore(str, str)
...
# añadimos algunas filas al almacén liststore
...
# bucle for
for row in liststore:
    # procesado de cada fila
...
# comprensión de lista que devuelve una lista de los valores de la primera ↩
    columna
values = [ r[0] for r in liststore ]
...
```

Otras partes del protocolo de mapeado que están soportadas son el uso de `del` para eliminar un fila del modelo y la obtención de un `TreeModelRow` de PyGTK del modelo utilizando un valor de clave que sea un camino o in iterador `TreeIter`. Por ejemplo, las siguientes instrucciones devuelven la primera fila de un modelo `TreeModel` y la instrucción final borra la primera fila hija de la primera fila:

```
row = model[0]
row = model['0']
row = model["0"]
row = model[(0,)]
i = model.get_iter(0)
row = model[i]
del model[(0,0)]
```

Además, se pueden fijar los valores en una fila existente de forma parecida a esto:

```
...
liststore = gtk.ListStore(str, int, object)
...
liststore[0] = ['Button', 23, gtk.Button('Label')]
```

Los objetos `TreeModelRow` de PyGTK soportan los protocolos de Python de secuencia e iterador. Se puede obtener un iterador para recorrer los valores de columna en una fila o utilizar la instrucción for así como la comprensión de listas. Un `TreeModelRow` utiliza el número de columna como índice para extraer un valor. Por ejemplo:

```
...
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
...
```

Haciendo uso del ejemplo de la sección anterior para iterar sobre un almacén `TreeStore` y localizar una fila que contenga un valor concreto, el código quedaría:

```
treestore = TreeStore(str)
...
def match_func(row, data):
    column, key = data # data es una tupla que contiene número de columna,  ↩
        clave (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'))
```

También se puede fijar el valor de una columna utilizando:

```
treestore[(1,0,1)][1] = 'abc'
```

Un `TreeModelRow` también soporta la instrucción `del` y la conversión a listas y tuplas utilizando las funciones de Python `list`() and `tuple`(). Tal como se ilustra en el ejemplo superior, un objeto `TreeModelRow` posee el método iterchildren(), que devuelve un iterador para recorrer las filas hijas del objeto `TreeModelRow`.

14.2.8. Señales de TreeModel

Las aplicaciones pueden seguir los cambios de un modelo `TreeModel` conectándose a las señales que son emitidas por un modelo `TreeModel`: "row-changed" (fila modificada), "row-deleted" (fila elim-

inada), "row-inserted" (fila insertada), "row-has-child-toggled" (cambio de fila tiene descendencia) y "rows-reordered" (filas reordenadas). Estas señales son utilizadas por las vistas `TreeView` para seguir los cambios en su modelo `TreeModel`.

Si una aplicación se conecta a estas señales es posible que se produzcan grupos de señales al llamar algunos métodos. Por ejemplo, una llamada para añadir la primera fila a una fila madre:

```
treestore.append(parent, ['qwe', 'asd', 123])
```

causará la emisión de las siguientes señales:

- "row-inserted" en donde la fila insertada estará vacía.
- "row-has-child-toggled" puesto que la fila madre (*parent*) no tenía previamente ninguna fila hija.
- "row-changed" para la fila insertada al establecer el valor 'qwe' en la primera columna.
- "row-changed" para la fila insertada al establecer el valor 'asd en la segunda columna.
- "row-changed" para la fila insertada al establecer el valor 123 en la tercera columna.

Nótese que no es posible obtener el orden de las fila en la retrollamada de "rows-reordered" puesto que el nuevo orden de las filas se pasa como un puntero opaco a un vector de enteros.

Consulte el Manual de Referencia de PyGTK para saber más sobre las señales de `TreeModel`.

14.2.9. Ordenación de filas de modelos TreeModel

14.2.9.1. La interfaz TreeSortable

Los objetos `ListStore` y `TreeStore` implementan la interfaz `TreeSortable` que les aporta métodos para controlar la ordenación de las filas de un modelo `TreeModel`. El elemento clave de la interfaz es "sort column ID", que es un valor entero arbitrario que está referido a una función de comparación de orden y a unos datos asociados de usuario. Dicho identificador de orden de columna debe ser mayor o igual a cero, y se crea utilizando el siguiente método:

```
treesortable.set_sort_func(sort_column_id, sort_func, user_data=None)
```

donde *sort_column_id* es un valor entero asignado por el programador, *sort_func* es una función o método utilizado para comparar filas y *user_data* son los datos contextuales. *sort_func* tiene la siguiente signatura:

```
def sort_func_function(model, iter1, iter2, data)
def sort_func_method(self, model, iter1, iter2, data)
```

donde *model* el el modelo `TreeModel` que contiene las filas señaladas por los iteradores `TreeIter` *iter1* e *iter2* y *data* es *user_data*. *sort_func* debería devolver: -1 si la fila correspondiente a *iter1* debe preceder a la fila correspondiente a *iter2*; 0, si las filas son iguales; y, 1 si la fila correspondiente a *iter2* debe preceder a la indicada por *iter1*. La función de comparación de orden debería presuponer siempre que el orden de clasificación es ascendente (`gtk.SORT_ASCENDING`) puesto que el orden de clasificación será tenido en cuenta por las implementaciones de `TreeSortable`.

Puede usarse la misma función de comparación de orden para varios identificadores de orden de columna, haciendo variar los datos de información contextual contenidos en *user_data*. Por ejemplo, los datos *user_data* especificados en el método set_sort_func() podrían consistir en los índices de las columnas de donde se extraerían los datos de clasificación.

Una vez que se ha creado un identificador de orden de columna (sort column ID) en un almacén, es posible usarla para ordenar los datos llamando al método:

```
treesortable.set_sort_column_id(sort_column_id, order)
```

donde *order* es el orden de clasificación, bien `gtk.SORT_ASCENDING` (ascendente) o `gtk.SORT_DESCENDING` (descendente).

Un identificador de orden de columna (*sort_column_id*) de valor igual a -1 indica que el almacén debería utilizar la función de comparación por defecto que se establece a través del método:

```
treesortable.set_default_sort_func(sort_func, user_data=None)
```

Es posible comprobar si un almacén tiene una función de comparación por defecto haciendo uso del método:

```
result = treesortable.has_default_sort_func()
```

que devuelve `TRUE`si se ha establecido una función por defecto para las comparaciones.

Una vez que se ha establecido un identificador de orden de columna para un modelo `TreeModel` que implementa la interfaz `TreeSortable` este modelo ya no puede volver al estado original no ordenado. Es posible cambiar su función de clasificación o utilizar la función por defecto, pero ya no es posible establecer que ese modelo `TreeModel` carezca de dicha función.

14.2.9.2. Clasificación en almacenes ListStore y TreeStore

Cuando se crea un objeto del tipo `ListStore` o `TreeStore` automáticamente se establecen identificadores de orden de columna que se corresponden con el número de índice de las columnas. Por ejemplo, un almacén `ListStore` con tres columnas tendrían tres identificadores de orden de columna (0, 1, 2) generados automáticamente. Estos identificadores están asociados a una función de comparación interna que maneja los tipos fundamentales:

- 'gboolean'
- str
- int
- long
- float

Inicialmente, tanto los almacenes `ListStore` como los `TreeStore` se fijan con un identificador de orden de columna igual a -2, que indica que no se utiliza una función de ordenación y que el almacén no está ordenado. Una vez que se fija un identificador de clasificación de columna en un `ListStore` o `TreeStore` ya no es posible volver a establecerlo al valor -2.

En caso de que se desee mantener los identificadores de orden de columna por defecto se debe establecer su valor fuera del rango del número de columnas, tal como 1000 o más. entonces es posible cambiar entre las funciones de ordenación por defecto y las de la aplicación según sea necesario.

14.3. TreeViews (Vistas de árbol)

Un `TreeView` es fundamentalmente un contenedor de objetos de columna `TreeViewColumn` e intérpretes de celda `CellRenderer` que son los responsables de llevar a cabo en último término la visualización de los datos del almacén de datos. También aporta una interfaz para las filas de datos mostradas y para las características que controlan la visualización de esos datos.

14.3.1. Creación de un TreeView (vista de árbol)

Un `TreeView` se crea utilizando su constructor:

```
treeview = gtk.TreeView(model=None)
```

donde *`model`* es un objeto que implementa la interfaz `TreeModel` (generalmente un `ListStore` o `TreeStore`). Si *`model`* es `None` o no se especifica, entonces el `TreeView` no estará asociado a un almacén de datos.

14.3.2. Obtención y establecimiento del Modelo de un TreeView

El modelo de árbol que proporciona el almacén de datos de un `TreeView` puede obtenerse utilizando el método get_model() :

```
model = treeview.get_model()
```

Un `TreeModel` puede asociarse simultáneamente con más de un `TreeView`, que cambia automáticamente su visualización en el memento en el que cambian los datos del `TreeModel`. Mientras que un`TreeView` siempre muestra todas las filas de su modelo de árbol, permite mostrar selectivamente algunas de las columnas. Ello significa que dos `TreeView`s asociados al mismo `TreeModel` pueden realizar visualizaciones completamente distintas de los mismos datos.

Es también importante darse cuenta de que no existe relación preestablecida entre las columnas de un `TreeView` y las columnas de su `TreeModel`. Por tanto, la quinta columna de los datos de un `TreeModel` pueden mostrarse en la primera columna de un `TreeView` y en la tercera columna de otro.

Un `TreeView` puede cambiar su modelo de árbol utilizando el método set_model() :

```
treeview.set_model(model=None)
```

donde *`model`* es un objeto que implementa la interfaz `TreeModel` (p.e. `ListStore` y `TreeStore`). Si *`model`* es `None`, entonces se descarta el modelo actual.

14.3.3. Definición de las propiedades de un TreeView

`TreeView` tiene una serie de propiedades que se pueden gestionar utilizando sus métodos:

"enable-search"	Lectura-Escritura	Si es `TRUE`, el usuario puede hacer búsquedas a traves de las columnas de forma interactiva. Por defecto es `TRUE`
"expander-column"	Lectura-Escritura	La columna usada para el elemento de expansión. Por defecto es 0
"fixed-height-mode"	Lectura-Escritura	Si es `TRUE`, asume que todas las filas tiene la misma altura, lo que acelera la visualización. Disponible a partir de GTK+ 2.4. Por defecto es `FALSE`
"hadjustment"	Lectura-Escritura	El control `Adjustment` (ajuste) horizontal del control. Se crea uno nuevo por defecto.
"headers-clickable"	Escritura	Si es `TRUE`, entonces los encabezados de las columnas responden a los eventos de click. Por defecto es `FALSE`
"headers-visible"	Lectura-Escritura	Si es `TRUE`, entonces muestra los botones de encabezado de columna. Por defecto es `TRUE`
"model"	Lectura-Escritura	El modelo del tree view. Por defecto es `None`
"reorderable"	Lectura-Escritura	Si es `TRUE`, la vista es reorganizable. Por defecto es `FALSE`
"rules-hint"	Lectura-Escritura	Si es `TRUE`, entonces indicar al motor de temas que dibuje las filas en colores alternados. Por defecto es `FALSE`
"search-column"	Lectura-Escritura	Indica la columna del modelo en la que buscar cuando se hace a través de código. Por defecto es -1.
"vadjustment"	Lectura-Escritura	El `Adjustment` (ajuste) vertical para el control. Se crea uno nuevo por defecto.

Los métodos correspondientes son:

```
enable_search = treeview.get_enable_search()
treeview.set_enable_search(enable_search)

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

hadjustment = treeview.get_hadjustment()
treeview.set_hadjustment(adjustment)

treeview.set_headers_clickable(active)

headers_visible = treeview.get_headers_visible()
treeview.set_headers_visible(headers_visible)

reorderable = treeview.get_reorderable()
treeview.set_reorderable(reorderable)
```

```
riles_hint = treeview.get_rules_hint()
treeview.set_rules_hint(setting)

column = treeview.get_search_column()
treeview.set_search_column(column)

vadjustment = treeview.get_vadjustment()
treeview.set_vadjustment(adjustment)
```

La función de la mayoría de ellos resulta obvia por su descripción. Sin embargo, la propiedad "enable-search" necesita que se haya definido correctamente la propiedad "search-column" como un número de una columna válida del modelo de árbol. Entonces, cuando el usuario o usuaria pulsa Control+f aparece un diálogo de búsqueda en el que se puede escribir. Se selecciona la primera fila coincidente a medida que se teclea..

Análogamente, la propiedad "headers-clickable" realmente solamente fija la propiedad "clickable" de las columnas `TreeViewColumn`s subyacentes. Una `TreeViewColumn` no será ordenable salvo que su modelo de árbol implemente la interfaz `TreeSortable` y se haya llamado el método `TreeViewColumn` set_sort_column_id() con un número válido de columna.

La propiedad "reorderable" permite que usuarios y usuarias reordenen el modelo `TreeView` arrastrando y soltando las filas mostradas del `TreeView`.

La propiedad "rules-hint" no debe establecerse salvo en el caso de tener muchas columnas y cuando resulte útil mostrarlas con colores alternos.

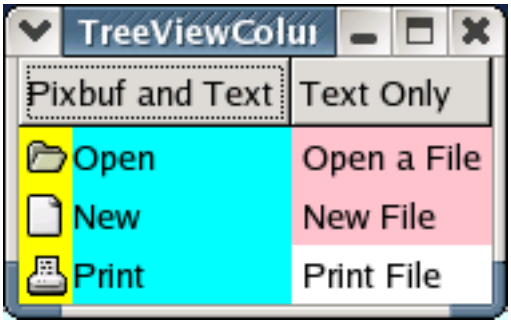
14.4. Visualizadores de Celda (CellRenderer)

14.4.1. Introducción

Las columnas de vista de árbol (`TreeViewColumn`) y los Visualizadores o Intérpretes de Celda (`CellRenderer`) colaboran en la visualización de una columna de datos de una vista de árbol (`TreeView`). La clase `TreeViewColumn` proporciona el título de columna y un espacio vertical para que los `CellRenderer` muestren una porción de los datos de los que contiene el almacén del `TreeView`. Un `CellRenderer` maneja la visualización de los datos de cada fila y columna dentro de los confines de una `TreeViewColumn`. Una `TreeViewColumn` puede contener más de un `CellRenderer` para proporcionar una visualización de fila similar a la de una `HBox`. Un uso habitual con múltiples `CellRenderer` es la combinación de un Visualizador de Imágenes en Celda (`CellRendererPixbuf`) y un Visuallizador de Texto en Celda (`CellRendererText`) en la misma columna.

El ejemplo Figura 14.2 muestra un ejemplo que ilustra la composición de dos `TreeViewColumn`, una con dos `CellRenderer` y la otra con uno sólo:

Figura 14.2 TreeViewColumns con CellRenderers



La aplicación de cada `CellRenderer` se indica mediante un color de fondo diferenciado: amarillo en el caso de un `CellRendererPixbuf`, cian para un `CellRendererText`, y rosa para el otro `CellRendererText`. Hay que resaltar que el `CellRendererPixbuf` y el primer `CellRendererText` están en la misma columna, encabezada con el texto "Pixbuf and Text". El color de fondo del `CellRendererText` que muestra "Print File" es el color predeterminado que muestra el área de la aplicación en una fila.

Figura 14.2 se creó con el programa treeviewcolumn.py.

14.4.2. Tipos de Visualizadores CellRenderer

El tipo del `CellRenderer` que se necesita para cada caso viene determinado por el tipo de visualización requerida por los datos del modelo de árbol usado. PyGTK posee tres `CellRenderer` predefinidos:

`CellRendererPixbuf` visualiza imágenes de píxeles (pixbuf) que pueden haber sido creadas por el programa, o tratarse de una imagen de serie predefinida.

`CellRendererText` visualiza cadenas de texto, así como números que pueden ser convertidos en cadenas (enteros, reales y booleanos incluidos).

`CellRendererToggle` visualiza un valor booleano como un botón biestado o como un botón de exclusión

14.4.3. Propiedade de un CellRenderer

Las propiedades de un `CellRenderer` determinan la forma en que se visualizarán los datos:

"mode"	Lectura-Escritura	El modo de edición del `CellRenderer`. Es uno de estos: `gtk.CELL_RENDERER_MODE_INERT` (inerte), `gtk.CELL_RENDERER_MODE_ACTIVATABLE` (activable) o `gtk.CELL_RENDERER_MODE_EDITABLE` (editable)
"visible"	Lectura-Escritura	Si es `TRUE` se muestra la celda.
"xalign"	Lectura-Escritura	La fracción de espacio *libre* a la izquierda de la celda dentro del intervalo 0.0 a 1.0.
"yalign"	Lectura-Escritura	La fracción de espacio *libre* sobre la celda dentro del intervalo 0.0 a 1.0.
"xpad"	Lectura-Escritura	La cantidad de margen a la derecha e izquierda de la celda.
"ypad"	Lectura-Escritura	La cantidad de margen sobre y bajo la celda.
"width"	Lectura-Escritura	La anchura fija de la celda.
"height"	Lectura-Escritura	La altura fija de la celda.
"is-expander"	Lectura-Escritura	Si es `TRUE` la fila tiene descendientes (hijas)
"is-expanded"	Lectura-Escritura	Si es `TRUE` la fila tiene descendientes y se expande para mostrarlas.
"cell-background"	Escritura	El color de fondo de la celda indicada como cadena.
"cell-background-gdk"	Lectura-Escritura	El color de fondo indicado como `gtk.gdk.Color`.
"cell-background-set"	Lectura-Escritura	Si es `TRUE` el color de fondo de la celda lo determina este visualizador

Las propiedades anteriores están disponibles para todas las subclases de `CellRenderer`. Pero los distintos tipos de `CellRenderer` también tienen propiedades exclusivas.

Los `CellRendererPixbuf` tienen estas propiedades:

"pixbuf"	Lectura-Escritura	El pixbuf que se visualizará (es anulada por "stock-id")
"pixbuf-expander-open"	Lectura-Escritura	Pixbuf para el expansor cuando está desplegado.
"pixbuf-expander-closed"	Lectura-Escritura	Pixbuf para el expansor cuando está cerrado.
"stock-id"	Lectura-Escritura	El stock ID del icono de serie que se visualizará.
"stock-size"	Read-Write	El tamaño del icono representado.

"stock-detail"	Lectura-Escritura	Detalle de visualización que se proporcionará al motor de temas.

Los `CellRendererText` tienen un gran número de propiedades que tratan fundamentalmente con la especificación de estilos:

"text"	Lectura-Escritura	Texto que se visualizará.
"markup"	Lectura-Escritura	Texto con marcas que se visualizará.
"attributes"	Lectura-Escritura	Una lista de atributos de estilo que se aplicarán al texto del visualizador.
"background"	Escritura	Color de fondo, como cadena de texto.
"foreground"	Escritura	Color de primer plano, como cadena de texto.
"background-gdk"	Lectura-Escritura	Color de fondo, como `gtk.gdk.Color`
"foreground-gdk"	Lectura-Escritura	Color de primer plano, como `gtk.gdk.Color`
"font"	Lectura-Escritura	Descripción de la fuente, como cadena de texto.
"font-desc"	Lectura-Escritura	Descripción de la fuente, como `pango.FontDescription`.
"family"	Lectura-Escritura	Nombre de la familia de la fuente, p.e. Sans, Helvetica, Times, Monospace.
"style"	Lectura-Escritura	Estilo de fuente.
"variant"	Lectura-Escritura	Variante de la fuente.
"weight"	Lectura-Escritura	Peso de la fuente.
"stretch"	Lectura-Escritura	Estirado de la fuente.
"size"	Lectura-Escritura	Tamaño de la fuente.
"size-points"	Lectura-Escritura	Tamaño de la fuente en puntos.
"scale"	Lectura-Escritura	Factor de escala de la fuente.
"editable"	Lectura-Escritura	Si es `TRUE` el texto puede ser cambiado por el usuario.
"strikethrough"	Lectura-Escritura	Si es `TRUE` se tacha el texto
"underline"	Lectura-Escritura	Estilo de subrayado del texto.
"rise"	Lectura-Escritura	Elevación del texto por encima de la línea base (o por debajo si el valor es negativo)
"language"	Lectura-Escritura	El idioma del texto, como código ISO. Pango puede utilizarlo como pista al representar el texto. Si no se entiende este parámetro... probablemente es que no se necesita. Solamente disponible a partir de GTK+ 2.4.
"single-paragraph-mode"	Lectura-Escritura	Si es `TRUE`, se deja todo el texto en un único párrafo. Solamente disponible a partir de GTK+ 2.4.

"background-set"	Lectura-Escritura	Si es `TRUE` se aplica el color de fondo.
"foreground-set"	Lectura-Escritura	Si es `TRUE` se aplica el color de primer plano.

"family-set"	Lectura-Escritura	Si es `TRUE` se aplica la familia de la fuente.
"style-set"	Lectura-Escritura	Si es `TRUE` se aplica el estilo de la fuente.
"variant-set"	Lectura-Escritura	Si es `TRUE` se aplica la variante de la fuente.
"weight-set"	Lectura-Escritura	Si es `TRUE` se aplica el peso de la fuente.
"stretch-set"	Lectura-Escritura	Si es `TRUE` se aplica el estirado de la fuente.
"size-set"	Lectura-Escritura	Si es `TRUE` se aplica el tamaño de fuente.
"scale-set"	Lectura-Escritura	si es `TRUE` se escala la fuente.
"editable-set"	Lectura-Escritura	Si es `TRUE` se aplica la editabilidad del texto.
"strikethrough-set"	Lectura-Escritura	Si es `TRUE` se aplica el tachado.
"underline-set"	Lectura-Escritura	Si es `TRUE` se aplica el subrayado de texto.
"rise-set"	Lectura-Escritura	Si es `TRUE` se aplica la elevación del texto.
"language-set"	Lectura-Escritura	Si es `TRUE` se aplica el idioma usado para mostrar el texto. A partir de GTK+ 2.4.

Casi cada una de las propiedades de `CellRendererText` posee una propiedad booleana asociada (con el suffijo "-set") que indica si se aplica dicha propiedad. Esto permite fijar globalmente una propiedad y activar o desactivar su aplicación selectivamente.

Los `CellRendererToggle` poseen las siguientes propiedades:

"activatable"	Lectura-Escritura	Si es `TRUE`, el botón biestado se puede activar.
"active"	Lectura-Escritura	Si es `TRUE`, el botón está activo.
"radio"	Lectura-Escritura	Si es `TRUE`, se dibuja el botón como un botón de exclusión.
"inconsistent"	Lectura-Escritura	Si es `TRUE`, el botón está en un estado inconsistente. A partir de GTK+ 2.2.

Las propiedades se pueden fijar para todas las filas utilizando el método gobject.set_property(). Véase el programa treeviewcolumn.py como ejemplo del uso de este método.

14.4.4. Atributos de un CellRenderer

Un atributo asocia una columna de un modelo de árbol a una propiedad de un `CellRenderer`. El `CellRenderer` fija la propiedad en función del valor de una columna de la fila antes de representar la celda. Esto permite personalizar la visualización de la celda utilizando los datos del modelo de árbol. Se puede añadir un atributo al conjunto actual con:

```
treeviewcolumn.add_attribute(cell_renderer, attribute, column)
```

donde la propiedad especificada por *attribute* se fija para el *cell_renderer* en la columna *column*. Por ejemplo:

```
treeviewcolumn.add_attribute(cell, "cell-background", 1)
```

establece el fondo del `CellRenderer` al color indicado por la cadena de la segunda columna del almacén de datos.

Para eliminar todos los atributos y establecer varios atributos nuevos de una vez se usa:

```
treeviewcolumn.set_attributes(cell_renderer, ...)
```

donde los atributos de *cell_renderer* se determinan mediante pares clave-valor: propiedad=columna. Por ejemplo, en el caso de un `CellRendererText`:

```
treeviewcolumn.set_attributes(cell, text=0, cell_background=1, xpad=3)
```

indica, para cada fila, el texto en la primera columna, el color de fondo en la segunda y el margen horizontal desde la cuarta columna. Véase el programa treeviewcolumn.py para ver ejemplos del uso de estos métodos.

Los atributos de un `CellRenderer` se pueden limpiar utilizando:

```
treeviewcolumn.clear_attributes(cell_renderer)
```

14.4.5. Función de Datos de Celda

Si no es suficiente el uso de atributos para cubrir nuestras necesidades, también es posible indicar una función que será llamada en cada fila y que determine las propiedades del `CellRenderer` utilizando:

```
treeviewcolumn.set_cell_data_func(cell_renderer, func, data=None)
```

donde *func* tiene la signatura:

```
def func(column, cell_renderer, tree_model, iter, user_data)
```

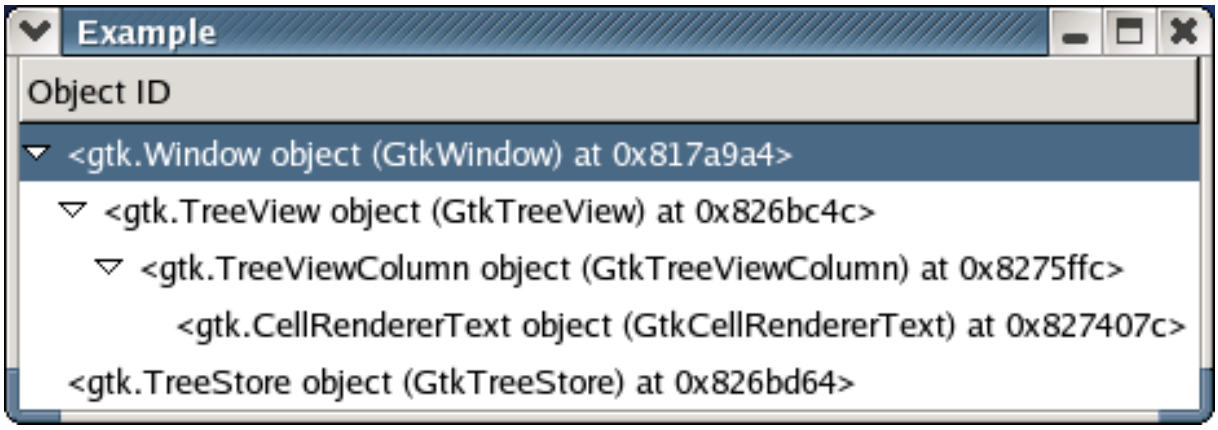
donde *column* es la `TreeViewColumn` que contiene el visualizador *cell_renderer*, *tree_model* es el almacén de datos e *iter* es un iterador `TreeIter` que apunta a una fila en *tree_model*. *user_data* es el valor de *data* que se pasó a set_cell_data_func().

En *func* se establecen las propiedades que se deseen para *cell_renderer*. Por ejemplo, el siguiente fragmento de código establece la propiedad de texto de manera que muestre los objetos de PyGTK como una cadena de identificación ID.

```
...
def obj_id_str(treeviewcolumn, cell_renderer, 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)
treeview.append_column(tvcolumn)
iter = treestore.append(None, [win])
iter = treestore.append(iter, [treeview])
iter = treestore.append(iter, [tvcolumn])
iter = treestore.append(iter, [cell])
iter = treestore.append(None, [treestore])
...
```

El resultado debería ser algo como Figura 14.3:

Figura 14.3 Función de Datos de Celda



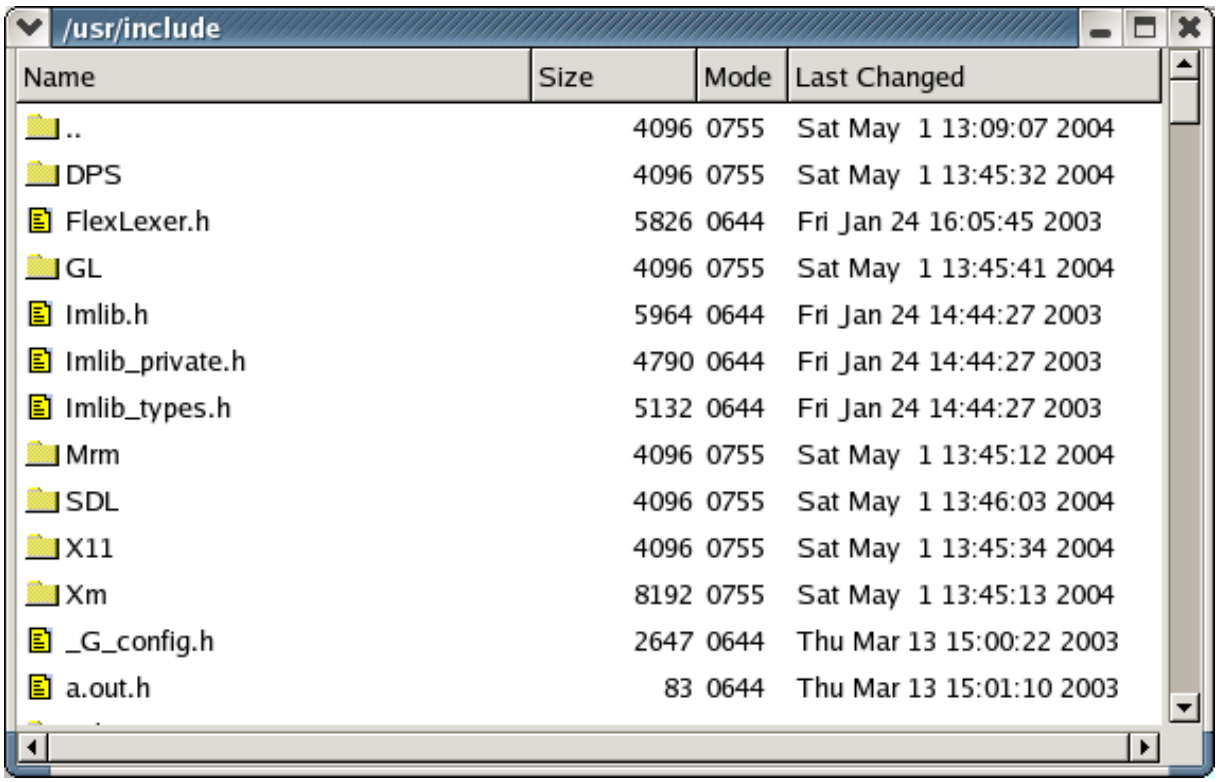
Otro posible uso de la función de datos de celda es el control del formato de visualización de un texto numérico, p.e. un valor real. Un `CellRendererText` hará una conversión de forma automática del valor real a una cadena, pero con el formato predeterminado "%f".

Con funciones de datos de celda se pueden generar incluso los datos de las celdas a partir de datos externos. Por ejemplo, el programa filelisting.py usa un almacén `ListStore` con una única columna que contiene una lista de nombres de archivos. La `TreeView` muestra columnas que incluyen una imagen pixbuf, el nombre de archivo y su tamaño, modo y fecha del último cambio. Los datos son generados por las siguientes funciones de datos de celda:

```
    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
```

Estas funciones obtienen la información de los archivos utilizando el nombre, extraen los datos necesarios y establecen las propiedades de celda 'text' o 'pixbuf' con los datos. Figura 14.4 muestra el programa de ejemplo en acción:

Figura 14.4 Ejemplo de Listado de Archivos Utilizando Funciones de Datos de Celda



14.4.6. Etiquetas de Marcado en CellRendererText

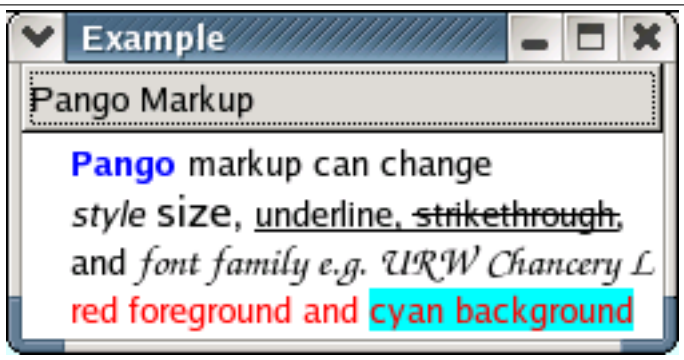
Un `CellRendererText` puede utilizar etiquetas de marcado de Pango (estableciendo la propiedad "markup") en vez de una cadena de texto sencilla para codificar diversos atributos de texto y proporcionar una visualización rica, con múltiples cambios de estilos de fuente. Véase la referencia Pango Markup en el Manual de Referencia de PyGTK para obtener más detalles sobre el lenguaje de marcado de Pango.

El siguiente fragmento de código ilustra el uso de la propiedad "markup":

```
  ...
  liststore = gtk.ListStore(str)
  cell = gtk.CellRendererText()
  tvcolumn = gtk.TreeViewColumn('Pango Markup', cell, markup=0)
  ...
  liststore.append(['<span foreground="blue"><b>Pango</b></span> markup can'
' change\n<i>style</i> <big>size</big>, <u>underline,'
 <s>strikethrough</s></u>,\n'
'and <span font_family="URW Chancery L"><big>font family '
'e.g. URW Chancery L</big></span>\n<span foreground="red">red'
' foreground and <span background="cyan">cyan background</span></span>'])
  ...
```

produce un resultado semejante a Figura 14.5:

Figura 14.5 Etiquetas de Marcado para CellRendererText



Si se crean etiquetas de marcado sobre la marcha es preciso tener cuidado y sustituir los caracteres con especial significado en el lenguaje de marcas: "<", ">", "&". La función de la biblioteca de Python `cgi.escape()` permite hacer estas conversiones básicas.

14.4.7. Celdas de Texto Editables

Las celdas `CellRendererText` pueden hacerse editalbes de forma que una usuaria pueda editar los contenidos de la celda que seleccione haciendo clic en ella o pulsando las teclas **Return**, **Enter**, **Space** o Shift+Space. Se hace editable un `CellRendererText` en todas sus filas estableciendo su propiedad "editable" a `TRUE` de la siguiente manera:

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

Se pueden establecer individualmente celdas editables añadiendo un atributo a la `TreeViewColumn` utilizando un `CellRendererText` parecido a:

```
treeviewcolumn.add_attribute(cellrenderertext, "editable", 2)
```

que establece que el valor de la propiedad "editable" se indica en la tercera columna del almacén de datos.

Una vez que la edición de la celda termina, la aplicación debe gestionar la señal "edited" para obtener el nuevo texto y establecer los datos asociados del almacén de datos. De otro modo, el valor de la celda recuperará su valor inicial. La signatura del manejador de llamada de la señal "edited" es:

```
def edited_cb(cell, path, new_text, user_data)
```

donde *`cell`* es el `CellRendererText`, *`path`* es el camino de árbol (como cadena) a la fila que contiene la celda editada, *`new_text`* es el texto editado y *`user_data`* son datos de contexto. Puesto que se necesita el `TreeModel` para usar el camino *`path`* y establecer *`new_text`* en el almacén de datos, probablemente se quiera pasar el `TreeModel` como *`user_data`* en el método connect():

```
cellrenderertext.connect('edited', edited_cb, model)
```

Si se tienen dos o más celdas editables en una fila, se podría pasar el número de columna del `TreeModel` como parte de los datos adicionales *`user_data`* así como el modelo `TreeModel`:

```
cellrenderertext.connect('edited', edited_cb, (model, col_num))
```

Así, se puede establecer el nuevo texto en el manejador de la señal "edited" de una forma parecida al siguiente ejemplo que usa un almacén de lista `ListStore`:

```
def edited_cb(cell, path, new_text, user_data):
    liststore, column = user_data
    liststore[path][column] = new_text
    return
```

14.4.8. Celdas Biestado Activables

Los botones de `CellRendererToggle` se pueden hacer activables estableciendo la propiedad "activatable" como `TRUE`. De forma parecida a la celdas editables de `CellRendererText` la propiedad "activatable" se puede fijar para un conjunto completo de celdas con `CellRendererToggle` utilizando el método set_property() o individualmente en algunas celdas añadiendo un atributo a la columna `TreeViewColumn` que contiene el `CellRendererToggle`.

```
cellrenderertoggle.set_property('activatable', True)

treeviewcolumn.add_attribute(cellrenderertoggle, "activatable", 1)
```

La creación de botones individuales biestado se puede deducir de los valores de una columna de un `TreeModel` añadiendo un atributo de manera similar a este ejemplo:

```
treeviewcolumn.add_attribute(cellrenderertoggle, "active", 2)
```

Se debe conectar a la señal "toggled" para disponer de notificación de las pulsaciones del usuario en los botones biestado, de manera que la aplicación pueda modificar los valores del almacén de datos. Por ejemplo:

```
cellrenderertoggle.connect("toggled", toggled_cb, (model, column))
```

La retrollamada tiene la signatura:

```
def toggled_cb(cellrenderertoggle, path, user_data)
```

donde *path* es el camino de árbol, como cadena, que apunta a la fila que contiene el botón biestado que ha sido pulsado. Es recomendable pasar el `TreeModel` y, tal vez, el índice de la columna como parte de los datos de usuario *user_data* para proporcionar el contexto necesario para establecer los valores del almacén de datos. Por ejemplo, la aplicación puede conmutar los valores del almacén de datos, así:

```
def toggled_cb(cell, path, user_data):
    model, column = user_data
    model[path][column] = not model[path][column]
    return
```

Si la aplicación desea mostrar los botones biestado como botones de exclusión y que únicamente uno de ellos esté activo, tendrá que recorrer los datos del almacén para desactivar el botón de exclusión activo y posteriormente activar el botón biestado. Por ejemplo:

```
def toggled_cb(cell, path, user_data):
    model, column = user_data
    for row in model:
        row[column] = False
    model[path][column] = True
    return
```

usa la estrategia "vaga" de poner todos los valores a `FALSE` antes de fijar el valor `TRUE` en la fila especificada en *path*.

14.4.9. Programa de Ejemplo de Celda Editable and Activable

El programa cellrenderer.py ilustra la utilización de celdas `CellRendererText` editables y de celdas `CellRendererToggle` activables en un almacén `TreeStore`.

```
 1  #!/usr/bin/env python
 2  # vim: ts=4:sw=4:tw=78:nowrap
 3  """ Demonstration using editable and activatable CellRenderers """
 4  import pygtk
 5  pygtk.require("2.0")
 6  import gtk, gobject
 7
 8  tasks =  {
 9      "Buy groceries": "Go to Asda after work",
10      "Do some programming": "Remember to update your software",
```

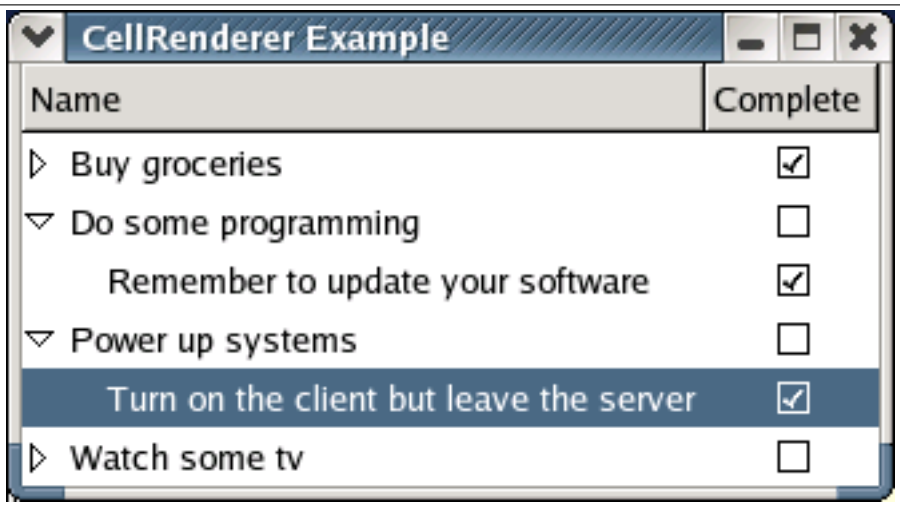
```
11        "Power up systems": "Turn on the client but leave the server",
12        "Watch some tv": "Remember to catch ER"
13        }
14
15    class GUI_Controller:
16        """ The GUI class is the controller for our application """
17        def __init__(self):
18            # establecer la ventana principal
19            self.root = gtk.Window(type=gtk.WINDOW_TOPLEVEL)
20            self.root.set_title("CellRenderer Example")
21            self.root.connect("destroy", self.destroy_cb)
22            # Obtener el modelo y vincularlo a la vista
23            self.mdl = Store.get_model()
24            self.view = Display.make_view( self.mdl )
25            # Añadir la vista a la ventana principal
26            self.root.add(self.view)
27            self.root.show_all()
28            return
29        def destroy_cb(self, *kw):
30            """ Destroy callback to shutdown the app """
31            gtk.main_quit()
32            return
33        def run(self):
34            """ run is called to set off the GTK mainloop """
35            gtk.main()
36            return
37
38    class InfoModel:
39        """ The model class holds the information we want to display """
40        def __init__(self):
41            """ Sets up and populates our gtk.TreeStore """
42            self.tree_store = gtk.TreeStore( gobject.TYPE_STRING,
43                                             gobject.TYPE_BOOLEAN )
44            # colocar los datos globales de la gente en la lista
45            # formamos un árbol simple.
46            for item in tasks.keys():
47                parent = self.tree_store.append( None, (item, None) )
48                self.tree_store.append( parent, (tasks[item],None) )
49            return
50        def get_model(self):
51            """ Returns the model """
52            if self.tree_store:
53                return self.tree_store
54            else:
55                return None
56
57    class DisplayModel:
58        """ Displays the Info_Model model in a view """
59        def make_view( self, model ):
60            """ Form a view for the Tree Model """
61            self.view = gtk.TreeView( model )
62            # configuramos el visualizador de celda de texto y permitimos
63            # la edición de las celdas.
64            self.renderer = gtk.CellRendererText()
65            self.renderer.set_property( 'editable', True )
66            self.renderer.connect( 'edited', self.col0_edited_cb, model )
67
68            # Se configura el visualizador de botones biestado y permitimos ↩
    que se
69            # pueda cambiar por el usuario.
70            self.renderer1 = gtk.CellRendererToggle()
71            self.renderer1.set_property('activatable', True)
72            self.renderer1.connect( 'toggled', self.col1_toggled_cb, model ) ↩
     73
```

```
74            # Conectamos la columna 0 de la visualización con la columna o de ↩
     nuestro modelo
75            # El visualizador mostrará lo que haya en la columna 0
76            # de nuestro modelo.
77            self.column0 = gtk.TreeViewColumn("Name", self.renderer, text=0)  ↩
      78
79            # El estado de activación del modelo se vincula a la segunda  ↩
    columna
80            # del modelo. Así, cuando el modelo dice True entonces el botón
81            # se mostrará activo, es decir, encendido.
82            self.column1 = gtk.TreeViewColumn("Complete", self.renderer1 )
83            self.column1.add_attribute( self.renderer1, "active", 1)
84            self.view.append_column( self.column0 )
85            self.view.append_column( self.column1 )
86            return self.view
87        def col0_edited_cb( self, cell, path, new_text, model ):
88            """
89            Called when a text cell is edited.  It puts the new text
90            in the model so that it is displayed properly.
91            """
92            print "Change '%s' to '%s'" % (model[path][0], new_text)
93            model[path][0] = new_text
94            return
95        def col1_toggled_cb( self, cell, path, model ):
96            """
97            Sets the toggled state on the toggle button to true or false.
98            """
99            model[path][1] = not model[path][1]
100           print "Toggle '%s' to: %s" % (model[path][0], model[path][1],)
101           return
102
103   if __name__ == '__main__':
104       Store = InfoModel()
105       Display = DisplayModel()
106       myGUI = GUI_Controller()
107       myGUI.run()
```

El programa proporcional celdas editables en la primera columna y celdas activables en la segunda columna. Las líneas 64-66 crean un `CellRendererText` editable y conectan la señal "edited" a la retrollamada col0_edited_cb() (líneas 87-94), que cambia el valor en la columna correspondiente de la fila en el almacén de árbol `TreeStore`. De la misma manera, las líneas 70-72 crean un `CellRendererToggle` activable y conectan la señal "toggled" a la retrollamada col1_toggled_cb() (líneas 95-101) para cambiar el valor de la fila correspondiente. Cuando se modifica una celada editable o activable se muestra un mensaje para indicar cuál ha sido el cambio.

Figura 14.6 ilustra el programa cellrenderer.py en ejecución.

Figura 14.6 Celdas Editables y Activables



14.5. TreeViewColumns (columnas de vista de árbol)

14.5.1. Creación de TreeViewColumns (columnas de vista de árbol)

Una `TreeViewColumn` se crea usando el constructor:

```
treeviewcolumn = gtk.TreeViewColumn(title=None, cell_renderer=None, ...)
```

donde *`title`* es la cadena que será usada como etiqueta en el encabezado de la columna, y *`cell_renderer`* es el primer `CellRenderer` que se empaqueta en la columna. Los argumentos adicionales que establecen los atributos de *`cell_renderer`* se pasan al constructor como valores con nombre (en el formato atributo=columna). Por ejemplo:

```
treeviewcolumn = gtk.TreeViewColumn('States', cell, text=0, foreground=1)
```

crea una `TreeViewColumn` en el que el `CellRendererText` llamado cell obtiene su texto de la primera columna del modelo de árbol y el color del texto de la segunda columna.

14.5.2. Gestión de los CellRenderers (Intérpretes de celda)

Se puede añadir un `CellRenderer` (intérprete de celda) a una `TreeViewColumn` usando uno de los siguientes métodos:

```
treeviewcolumn.pack_start(cell, expand)
treeviewcolumn.pack_end(cell, expand)
```

pack_start() y pack_end() añaden *`cell`* al principio o final, respectivamente, de la `TreeViewColumn`. Si *`expand`* es `TRUE`, *`cell`* compartirá el espacio disponible extra que haya reservado la `TreeViewColumn`.

El método get_cell_renderers() :

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

devuelve una lista de todos los `CellRenderer`s de una columna.

El método clear() elimina todos los atributos de `CellRenderer` de la columna `TreeViewColumn`:

```
treeviewcolumn.clear()
```

Hay muchos otros métodos disponibles para una `TreeViewColumn` (y la mayoría tienen que ver con el establecimiento y obtención de propiedades. Véase el Manual de Referencia de PyGTK para obtener más información sobre las propiedades de `TreeViewColumn`. La capacidad de usar la característica incluida de ordenación se determina con el método:

```
treeviewcolumn.set_sort_column_id(sort_column_id)
```

que establece *sort_column_id* como el identificador (ID) de columna de ordenación del modelo de árbol que se usará cuando se ordene la vista del TreeView. Este método también establece la propiedad "clickable" de la columna, lo que permite al usuario hacer click sobre en encabezado de la columna para activar la ordenación. Cuando se hace click en el encabezado de la columna el identificador (ID) de columna de ordenación de la TreeViewColumn se establece como identificador de columna de ordenación del TreeModel y las filas del TreeModel son reordenadas utilizando la función de comparación asociada. La herramienta de ordenación automática también conmuta el orden de clasificación de la columna y gestiona la visualización del indicador de orden. Véase la sección Ordenación de las filas de un Modelo de árbol para obtener más información sobre los identificadores de columna de ordenación y funciones de comparación. Habitualmente, cuando se usa un ListStore o TreeStore el identificador de columna de ordenación por defecto (el índice de la columna) de la columna del TreeModel asociada con la TreeViewColumn se establece como el identificador de columna de ordenación de la TreeViewColumn.

Si se usan los encabezados de las TreeViewColumns para hacer la ordenación, entonces si se utiliza el método set_sort_column_id() no es necesario utilizar el método TreeSortableset_sort_column_id() .

Se pueden rastrear las operaciones de ordenación o utilizar el click sobre los encabezados para propósitos específicos conectándose a la señal "clicked" de la columna de un TreeView. La función de retrollamada debería definirse así:

```
def callback(treeviewcolumn, user_data, ...)
```

14.6. Manipulación de TreeViews

14.6.1. Gestión de las Columnas

Las TreeViewColumns de un TreeView pueden obtenerse individualmente o como una lista utilizando los métodos:

```
treeviewcolumn = treeview.get_column(n)
columnlist = treeview.get_columns()
```

donde *n* es el índice (empezando desde 0) de la columna que se quiere obtener. Se puede eliminar una columna con el método:

```
treeview.remove_column(column)
```

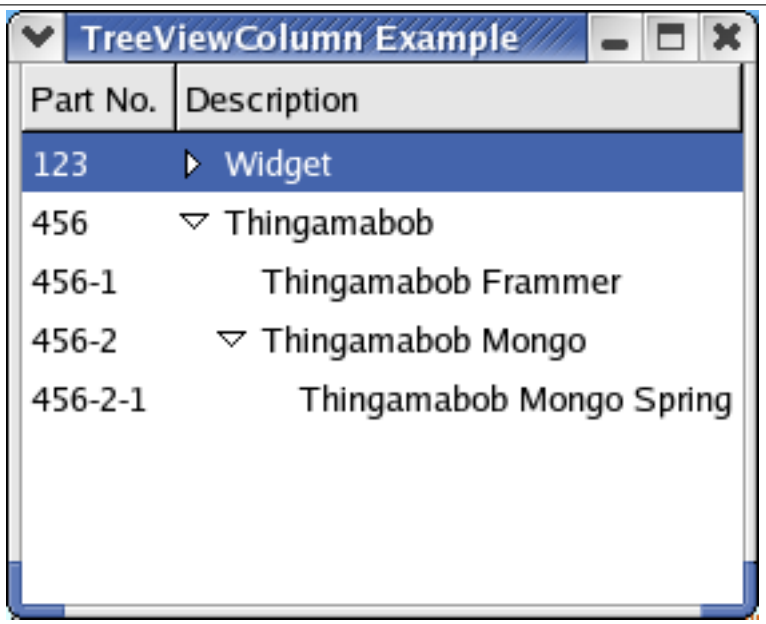
donde *column* es una TreeViewColumn en *treeview*.

Las filas que tienen filas hijas se muestran en el TreeView con una flecha de expansión Figura 14.3) que se puede pulsar para ocultar o mostrar las filas hijas. La columna en la que se muestra la flecha de expansión puede cambiarse utilizando el método:

```
treeview.set_expander_column(column)
```

donde *column* es una TreeViewColumn en treeview. Este método es útil cuando no se desea que se indente la primera columna. Por ejemplo, Figura 14.7 ilustra la flecha de expansión en la segunda columna:

Figura 14.7 Flecha de Expansión en la segunda Columna



14.6.2. Expansión y Contracción de Filas Hijas

Todas las filas mostradas en un `TreeView` pueden ser expandidas o contraídas desde un programa utilizando los métodos siguientes:

```
treeview.expand_all()
treeview.collapse_all()
```

Estos métodos son útiles si se quiere inicializar la visualización del `TreeView` en un estado determinado. Las filas individuales pueden ser expandidas o contraídas utilizando:

```
treeview.expand_row(path, open_all)
treeview.collapse_row(path)
```

donde *`path`* es el camino de árbol a una fila en treeview, y si *`open_all`* es `TRUE`, entonces todas las filas descendientes de *`path`* son expandidas; en otro caso, únicamente son expandidas las descendientes inmediatas.

Se puede determinar si una fila se expande utilizando el método:

```
is_expanded = treeview.row_expanded(path)
```

14.7. Señales de TreeView

Los controles `TreeView` emiten un gran número de señales que se pueden usar para seguir los cambios en la visualización del modelo. las señales caen generalmente en una de las siguientes categorías:

- expansión y contracción de filas: "row-collapsed", "row-expanded", "test-collapse-row", "test-expand-row" y "expand-collapse-cursor-row"
- el cursor: "cursor-changed", "expand-collapse-cursor-row", "move-cursor", "select-cursor-parent", "select-cursor-row" y "toggle-cursor-row"
- selección: "select-all", "select-cursor-parent", "select-cursor-row" y "unselect-all".
- varios: "columns-changed", "row-activated", "set-scroll-adjustments" y "start-interactive-search".

Las señales "test-collapse-row" y "test-expand-row" son emitidas antes de que se contraiga o expanda una fila. El valor devuelto por la retrollamada puede permitir o cancelar la operación (`TRUE` para permitirla y `FALSE` para cancelarla.

```
def callback(treeview, iter, path, user_data)
```

donde *iter* es un TreeIter y *path* es un camino de árbol que apunta a la fila y *user_data* son los datos especificados en el método connect() .

La señal "row-activated" se emite cuando se produce un doble click en una fila o cuando se selecciona una fila no editable y se pulsa una de las siguientes teclas: **Espacio**, Shift+Espacio, **Return** o **Enter**.

El resto de las señales se emiten tras haber cambiado el TreeView. El cursor es la fila marcada por una caja. En la mayoría de los casos la selección se mueve cuando se mueve el cursor. El cursor se puede mover de forma independiente mediante Control+Abajo o Control+Arriba y otras combinaciones de teclas.

Véase el Manual de Referencia de PyGTK para obtener más información sobre las señales de TreeView.

14.8. Selecciones TreeSelections

14.8.1. Obtención de TreeSelection

Las TreeSelections son objetos que gestionan las selecciones en un TreeView. Cuando se crea un TreeView también se crea automáticamente una TreeSelection. Puede obtenerse la TreeSelection de una TreeView utilizando el método:

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

Se puede obtener el TreeView asociado a una TreeSelection con el método:

```
treeview = treeselection.get_treeview()
```

14.8.2. Modos de una selección TreeSelection

Una selección TreeSelection soporta los siguientes modos de selección:

gtk.SELECTION_NONE No se permite hacer una selección.

gtk.SELECTION_SINGLE Se permite una única selección haciendo click.

gtk.SELECTION_BROWSE Se permite una única selección navegando con el puntero.

gtk.SELECTION_MULTIPLE Se pueden seleccionar múltiples elementos de una vez.

Se puede obtener el modo actual de selección llamando al método:

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

El modo puede establecerse utilizando:

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

donde *mode* es uno de los modos previamente indicados.

14.8.3. Obtención de la Selección

El método que es necesario usar par obtener la selección depende del modo actual de selección. Si el modo de selección es gtk.SELECTION_SINGLE o gtk.SELECTION_BROWSE, se debe usar el siguiente método:

```
(model, iter) = treeselection.get_selected()
```

que devuelve una tupla de dos elementos que contiene *model*, el TreeModel usado por el TreeView asociado con *treeselection* e *iter*, un iterador TreeIter que apunta a la fila seleccionada. Si no hay una fila seleccionada entonces *iter* es None. Si el modo de selección es gtk.SELECTION_MULTIPLE se produce una excepción TypeError.

Si se tiene una TreeView que utiliza el modo de selección gtk.SELECTION_MULTIPLE entonces se debería usar el método:

```
(model, pathlist) = treeselection.get_selected_rows()
```

que devuelve una tupla de dos elementos que contiene el modelo de árbol y una lista de los caminos de árbol de las filas seleccionadas. Este método no está disponible en PyGTK 2.0, por lo que es necesario utilizar una función auxiliar para obtener la lista mediante:

```
treeselection.selected_foreach(func, data=None)
```

donde *func* es una función que es llamada para cada fila seleccionada con *data*. La signatura de *func* es:

```
def func(model, path, iter, data)
```

donde *model* es el TreeModel, *path* es el camino de árbol de la fila seleccionada e *iter* es un TreeIter que señala la fila seleccionada.

Este método puede ser usado para simular el método get_selected_row() así:

```
...
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)
...
```

El método selected_foreach() no puede usarse para modificar el modelo de árbol o la selección, aunque sí permite cambiar los datos de las filas.

14.8.4. Uso de una Función de TreeSelection

Si se desea un control definitivo sobre la selección de filas se puede establecer una función que será llamada antes de que se seleccione o deselecciones una fila mediante el método:

```
treeselection.set_select_function(func, data)
```

donde *func* es una función de retrollamada y *data* son los datos de usuario que se pasarán a *func* cuando es llamada. *func* tiene la signatura:

```
def func(selection, model, path, is_selected, user_data)
```

donde *selection* es la selección TreeSelection, *model* es el TreeModel usado con el TreeView asociado con *selection*, *path* es el camino de árbol de la fila seleccionada, *is_selected* es TRUE si la fila está actualmente seleccionada y *user_data* es *data*. *func* debería devolver TRUE si el estado de selección de la fila debería ser conmutado.

Establecer una función de selección es útil en estos casos:

- se quiere controlar la selección o deselección de una fila en función de alguna información adicional de contexto. Se necesitará indicar de alguna manera que el cambio de selección no se puede llevar a cabo y, tal vez, el porqué. Por ejemplo, se puede diferenciar visualmente la fila o mostar un diálogo emergente del tipo MessageDialog.

- se necesita mantener una lista propia de filas seleccionadas o deseleccionadas, aunque esto mismo se puede hacer, con algo más de esfuerzo, conectándose a la señal "changed".

- se quiere hacer algún procesado adicional antes de que una fila sea seleccionada o deseleccionada. Por ejemplo, cambiar el aspecto de la fila o modificar los datos de la misma.

14.8.5. Selección y Deselección de Filas

Se puede cambiar la selección dentro de un programa utilizando los siguientes métodos:

```
treeselection.select_path(path)
treeselection.unselect_path(path)

treeselection.select_iter(iter)
treeselection.unselect_iter(iter)
```

Estos métodos seleccionan o deseleccionan una única fila que se especifica bien con *path*, un camino de árbol, o *iter*, un iterador `TreeIter` que apunta a la fila. Los siguientes métodos seleccionan o deseleccionan varias filas de una vez:

```
treeselection.select_all()
treeselection.unselect_all()

treeselection.select_range(start_path, end_path)
treeselection.unselect_range(start_path, end_path)
```

El método select_all() precisa que el modo de selección sea `gtk.SELECTION_MULTIPLE` al igual que el método select_range(). Los métodos unselect_all() y unselect_range() funcionarán con cualquier modo de selección. Nótese que el método unselect_all() no está disponible en PyGTK 2.0

Se puede comprobar si una fila está seleccionada utilizando uno de los siguientes métodos:

```
result = treeselection.path_is_selected(path)
result = treeselection.iter_is_selected(iter)
```

que devuelven `TRUE` si la fila especificada por *path* o *iter* está actualmente seleccionada. Se puede obtener el número de filas seleccionadas con el método:

```
count = treeselection.count_selected_rows()
```

Este método no está disponible en PyGTK 2.0 por lo que es preciso simularlo utilizando el método selected_foreach() de forma parecida a la simulación del método get_selected_rows() de la sección Obtención de la Selección. Por ejemplo:

```
...
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. Arrastrar y Soltar en TreeView

14.9.1. Reordenación mediante Arrastrar y Soltar

La reordenación de las filas de un `TreeView` (y de las filas del modelo de árbol subyacente) se activa usando el método set_reorderable() que se mencionó previamente. El método *set_reorderable*() fija la propiedad "reorderable" al valor especificado y permite o impide arrastrar y soltar en las filas del `TreeView`. Cuando la propiedad "reorderable" es `TRUE` es posible arrastar internamente filas del `TreeView` y soltarlas en una nueva posición. Esta acción provoca que las filas del `TreeModel` subyacente se reorganicen para coincidir con la nueva situación. La reordenación mediante arrastrar y soltar de filas funciona únicamente con almacenes no ordenados.

14.9.2. Arrastar y Soltar Externo

Si se quiere controlar el arrastar y soltar o tratar con el arrastrar y soltar desde fuentes externas de datos es necesario habilitar y controlar el arrastar y soltar con los siguientes métodos:

```
treeview.enable_model_drag_source(start_button_mask, targets, actions)
treeview.enable_model_drag_dest(targets, actions)
```

Estos métodos permiten utilizar filas como fuente de arrastre y como lugar para soltar respectivamente. *start_button_mask* es una máscara de modificación (véase referencia de constantes gtk.gtk Constants en el Manual de Referencia de PyGTK) que especifica los botones o teclas que deben ser pulsadas para iniciar la operación de arrastre. *targets* es una lista de tuplas de 3 elementos que describen la información del objetivo que puede ser recibido o dado. Para que tenga éxito el arrastar o soltar, por lo menos uno de los objetivos debe coincidir en la fuente o destino del arrastre (p.e. el objetivo "STRING"). Cada tupla de 3 elementos del objetivo contiene el nombre del objetivo, banderas (una combinación de `gtk.TARGET_SAME_APP` y `gtk.TARGET_SAME_WIDGET` o ninguno) y un identificador entero único. *actions* describe cuál debería ser el resultado de la operación:

`gtk.gdk.ACTION_DEFAULT`, `gtk.gdk.ACTION_COPY` Copiar los datos.

`gtk.gdk.ACTION_MOVE` Mover los datos, es decir, primero copiarlos, luego borrarlos de la fuente utilizando el objetivo `DELETE` del protocolo de selecciones de las X.

`gtk.gdk.ACTION_LINK` Añadir un enlace a los datos. Nótese que esto solamente es de utilidad si la fuente y el destino coinciden en el significado.

`gtk.gdk.ACTION_PRIVATE` Acción especial que informa a la fuente que el destino va a hacer algo que el destino no comprende.

`gtk.gdk.ACTION_ASK` Pide al usuario qué hacer con los datos.

Por ejemplo para definir un destino de un arrastrar y soltar:

```
treeview.enable_model_drag_dest([('text/plain', 0, 0)],
                gtk.gdk.ACTION_DEFAULT | gtk.gdk.ACTION_MOVE)
```

Entonces habrá que gestionar la señal del control `Widget` "drag-data-received" para recibir los datos recibidos - tal vez sustituyendo los datos de la fila en la que se soltó el contenido. La signatura de la retrollamada de la señal "drag-data-received" es:

```
def callback(widget, drag_context, x, y, selection_data, info, timestamp)
```

donde *widget* es el `TreeView`, *drag_context* es un `DragContext` que contiene el contexto de la selección, *x* e *y* son la posición en dónde ocurrió el soltar, *selection_data* es la `SelectionData` que contiene los datos, *info* es un entero identificador del tipo, *timestamp* es la hora en la que sucedió el soltar. La fila puede ser identificada llamando al método:

```
drop_info = treeview.get_dest_row_at_pos(x, y)
```

donde (*x*, *y*) es la posición pasada a la función de retrollamada y *drop_info* es una tupla de dos elementos que contiene el camino de una fila y una constante de posición que indica donde se produce el soltar respecto a la fila: `gtk.TREE_VIEW_DROP_BEFORE`, `gtk.TREE_VIEW_DROP_AFTER`, `gtk.TREE_VIEW_DROP_INTO_OR_BEFORE` o `gtk.TREE_VIEW_DROP_INTO_OR_AFTER`. La función de retrollamada podría ser algo parecido a:

```
treeview.enable_model_drag_dest([('text/plain', 0, 0)],
                gtk.gdk.ACTION_DEFAULT | gtk.gdk.ACTION_MOVE)
treeview.connect("drag-data-received", drag_data_received_cb)
...
...
def drag_data_received_cb(treeview, context, x, y, selection, info, timestamp):
    drop_info = treeview.get_dest_row_at_pos(x, y)
    if drop_info:
        model = treeview.get_model()
        path, position = drop_info
        data = selection.data
        # do something with the data and the model
        ...
    return
...
```

Si una fila se usa como fuente para arrastar debe manejar la señal de `Widget` "drag-data-get" que llena una selección con los datos que se devolverán al destino de arrastar y soltar con una función de retrollamada con la signatura:

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

Los parámetros de *`callback`* son similares a los de la función de retrollamada de "drag-data-received". Puesto que a la retrollamada no se le pasa un camino de árbol o una forma sencilla de obtener información sobre la fila que está siendo arrastrada, asumiremos que la fila que está siendo arrastrada está seleccionada y que el modo de selección es `gtk.SELECTION_SINGLE` o `gtk.SELECTION_BROWSE` de modo que podemos tener la fila obteniendo la `TreeSelection`, el modelo y el iterador `TreeIter` que apunta a la fila. Por ejemplo, se podría pasar texto así::

```
...
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
...
```

Un `TreeView` puede ser desactivado como fuente o destino para arrastrar y soltar utilizando los métodos:

```
treeview.unset_rows_drag_source()
treeview.unset_rows_drag_dest()
```

14.9.3. Ejemplo de Arrastrar y Soltar en TreeView

Es necesario un ejemplo para unir las piezas de código descritas más arriba. Este ejemplo (treeviewdnd.py) es una lista en la que se pueden arrastrar y soltar URLs. También se pueden reordenar las URLs de la lista arrastrando y soltando en el interior del `TreeView`. Un par de botones permiten limpiar la listar y eliminar un elemento seleccionado.

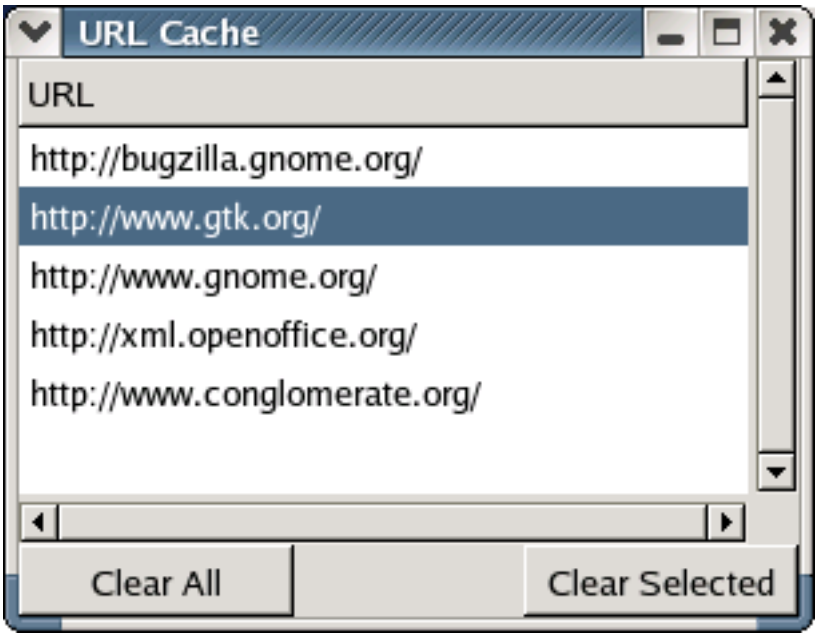
```
 1  #!/usr/bin/env python
 2
 3  # example treeviewdnd.py
 4
 5  import pygtk
 6  pygtk.require('2.0')
 7  import gtk
 8
 9  class TreeViewDnDExample:
10
11      TARGETS = [
12          ('MY_TREE_MODEL_ROW', gtk.TARGET_SAME_WIDGET, 0),
13          ('text/plain', 0, 1),
14          ('TEXT', 0, 2),
15          ('STRING', 0, 3),
16          ]
17      # close the window and quit
18      def delete_event(self, widget, event, data=None):
19          gtk.main_quit()
20          return gtk.FALSE
21
```

```
22      def clear_selected(self, button):
23          selection = self.treeview.get_selection()
24          model, iter = selection.get_selected()
25          if iter:
26              model.remove(iter)
27          return
28
29      def __init__(self):
30          # Create a new window
31          self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
32
33          self.window.set_title("URL Cache")
34
35          self.window.set_size_request(200, 200)
36
37          self.window.connect("delete_event", self.delete_event)
38
39          self.scrolledwindow = gtk.ScrolledWindow()
40          self.vbox = gtk.VBox()
41          self.hbox = gtk.HButtonBox()
42          self.vbox.pack_start(self.scrolledwindow, True)
43          self.vbox.pack_start(self.hbox, False)
44          self.b0 = gtk.Button('Clear All')
45          self.b1 = gtk.Button('Clear Selected')
46          self.hbox.pack_start(self.b0)
47          self.hbox.pack_start(self.b1)
48
49          # create a liststore with one string column to use as the model
50          self.liststore = gtk.ListStore(str)
51
52          # create the TreeView using liststore
53          self.treeview = gtk.TreeView(self.liststore)
54
55         # create a CellRenderer to render the data
56          self.cell = gtk.CellRendererText()
57
58          # create the TreeViewColumns to display the data
59          self.tvcolumn = gtk.TreeViewColumn('URL', self.cell, text=0)
60
61          # add columns to treeview
62          self.treeview.append_column(self.tvcolumn)
63          self.b0.connect_object('clicked', gtk.ListStore.clear, self. ↩
    liststore)
64          self.b1.connect('clicked', self.clear_selected)
65          # make treeview searchable
66          self.treeview.set_search_column(0)
67
68          # Allow sorting on the column
69          self.tvcolumn.set_sort_column_id(0)
70
71          # Allow enable drag and drop of rows including row move
72          self.treeview.enable_model_drag_source( gtk.gdk.BUTTON1_MASK,
73                                                  self.TARGETS,
74                                                  gtk.gdk.ACTION_DEFAULT|
75                                                  gtk.gdk.ACTION_MOVE)
76          self.treeview.enable_model_drag_dest(self.TARGETS,
77                                               gtk.gdk.ACTION_DEFAULT)
78
79          self.treeview.connect("drag_data_get", self.drag_data_get_data)
80          self.treeview.connect("drag_data_received",
81                                self.drag_data_received_data)
82
83          self.scrolledwindow.add(self.treeview)
84          self.window.add(self.vbox)
```

```
85            self.window.show_all()
86
87        def drag_data_get_data(self, treeview, context, selection, target_id,
88                               etime):
89            treeselection = treeview.get_selection()
90            model, iter = treeselection.get_selected()
91            data = model.get_value(iter, 0)
92            selection.set(selection.target, 8, data)
93
94        def drag_data_received_data(self, treeview, context, x, y, selection,
95                                    info, etime):
96            model = treeview.get_model()
97            data = selection.data
98            drop_info = treeview.get_dest_row_at_pos(x, y)
99            if drop_info:
100               path, position = drop_info
101               iter = model.get_iter(path)
102               if (position == gtk.TREE_VIEW_DROP_BEFORE
103                   or position == gtk.TREE_VIEW_DROP_INTO_OR_BEFORE):
104                   model.insert_before(iter, [data])
105               else:
106                   model.insert_after(iter, [data])
107           else:
108               model.append([data])
109           if context.action == gtk.gdk.ACTION_MOVE:
110               context.finish(True, True, etime)
111           return
112
113   def main():
114       gtk.main()
115
116   if __name__ == "__main__":
117       treeviewdndex = TreeViewDnDExample()
118       main()
```

El resultado de la ejecución del programa de ejemplo treeviewdnd.py se ilustra en Figura 14.8:

Figura 14.8 Ejemplo de Arrastrar y Soltar en TreeView



La clave para permitir tanto arrastrar y soltar externo como la reorganización interna de filas es la organización de los objetivos (el atributo `TARGETS` de la línea 11). Se crea y usa un objetivo específico de la

aplicación (MY_TREE_MODEL_ROW) para indicar un arrastar y soltar dentro del TreeView estableciendo la bandera gtk.TARGET_SAME_WIDGET. Estableciendo éste como el primer objetivo para el destino del arrastre hará que se intente hacerlo coincidir primero con los objetivos del origen de arrastre. Después, las acciones de fuente de arrastre deben incluir gtk.gdk.ACTION_MOVE y gtk.gdk.ACTION_DEFAULT (véanse las líneas 72-75). Cuando el destino recibe los datos de la fuente, si la acción DragContext es gtk.gdk.ACTION_MOVE, entonces se indica a la fuente que borre los datos (en este caso la fila) llamando al método del DragContext finish() (véanse las líneas 109-110). Un TreeView proporciona un conjunto de funciones internas que estamos aprovechando para arrastrar, soltar y eliminar los datos.

14.10. TreeModelSort y TreeModelFilter

Los objetos TreeModelSort y TreeModelFilter son modelos de árbol que se interponen entre el TreeModel de base (bien un TreeStore o un ListStore) y el TreeView que proporciona un modelo modificado mientras retiene la estructura original del modelo base. Estos modelos interpuestos implementan las interfaces TreeModel y TreeSortable pero no proporcionan ningún método para insertar o eliminar filas del modelo. Estas se deben insertar o eliminar del almacén subyacente. TreeModelSort proporciona un modelo cuyas filas están siempre ordenadas, mientras que TreeModelFilter proporciona un modelo que contiene un subconjunto de las filas del modelo base.

Si se desea, estos modelos pueden encadenarse en una sucesión arbitraria; es decir, un TreeModelFilter podría tener un TreeModelSort hijo, que podría tener otro TreeModelFilter hijo, y así sucesivamente. Mientras haya un almacén TreeStore o ListStore en el extrremo final de la cadena, todo debería funcionar. En PyGTK 2.0 y 2.2 los objetos TreeModelSort y TreeModelFilter no soportan el protocolo de mapeado de Python para TreeModel.

14.10.1. TreeModelSort (Modelo de Árbol Ordenado)

TreeModelSort mantiene un modelo ordenado del modelo hijo especificado en su constructor. El uso principal de un TreeModelSort el el de proporcionar múltiples vistas de un modelo que puedan ser ordenadas de forma distinta. Si se tienen múltiples vistas del mismo modelo entonces cualquier actividad de ordenación se ve reflejada en todas las vistas. Al usar un TreeModelSort el almacén base permanecen en su estado original, mientras que los modelos ordenados absorben toda la actividad de ordenación. Para crear un TreeModelSort se ha de usar el constructor:

```
treemodelsort = gtk.TreeModelSort(child_model)
```

donde *child_model* es un TreeModel. La mayoría de los métodos de un TreeModelSort tienen que ver con la conversión de los caminos de árbol e iteradores TreeIter desde el modelo hijo al modelo ordenado y viceversa:

```
sorted_path = treemodelsort.convert_child_path_to_path(child_path)
child_path = treemodelsort.convert_path_to_child_path(sorted_path)
```

Estos métodos de conversión de caminos devuelven None si el camino dado no puede ser convertido en un camino del modelo ordenado o el modelo hijo respectivamente. Los métodos de conversión de los iteradores TreeIter son:

```
sorted_iter = treemodelsort.convert_child_iter_to_iter(sorted_iter, child_iter)
child_iter = treemodelsort.convert_iter_to_child_iter(child_iter, sorted_iter)
```

Los métodos de conversión de iteradores TreeIter duplican el argumento convertido (es tanto el valor de retorno como el primer argumento) debido a cuestiones de compatiblidad con versiones anteriores; se deben fijar el primer argumento como None y simplemente utilizar el valor devuelto. Por ejemplo:

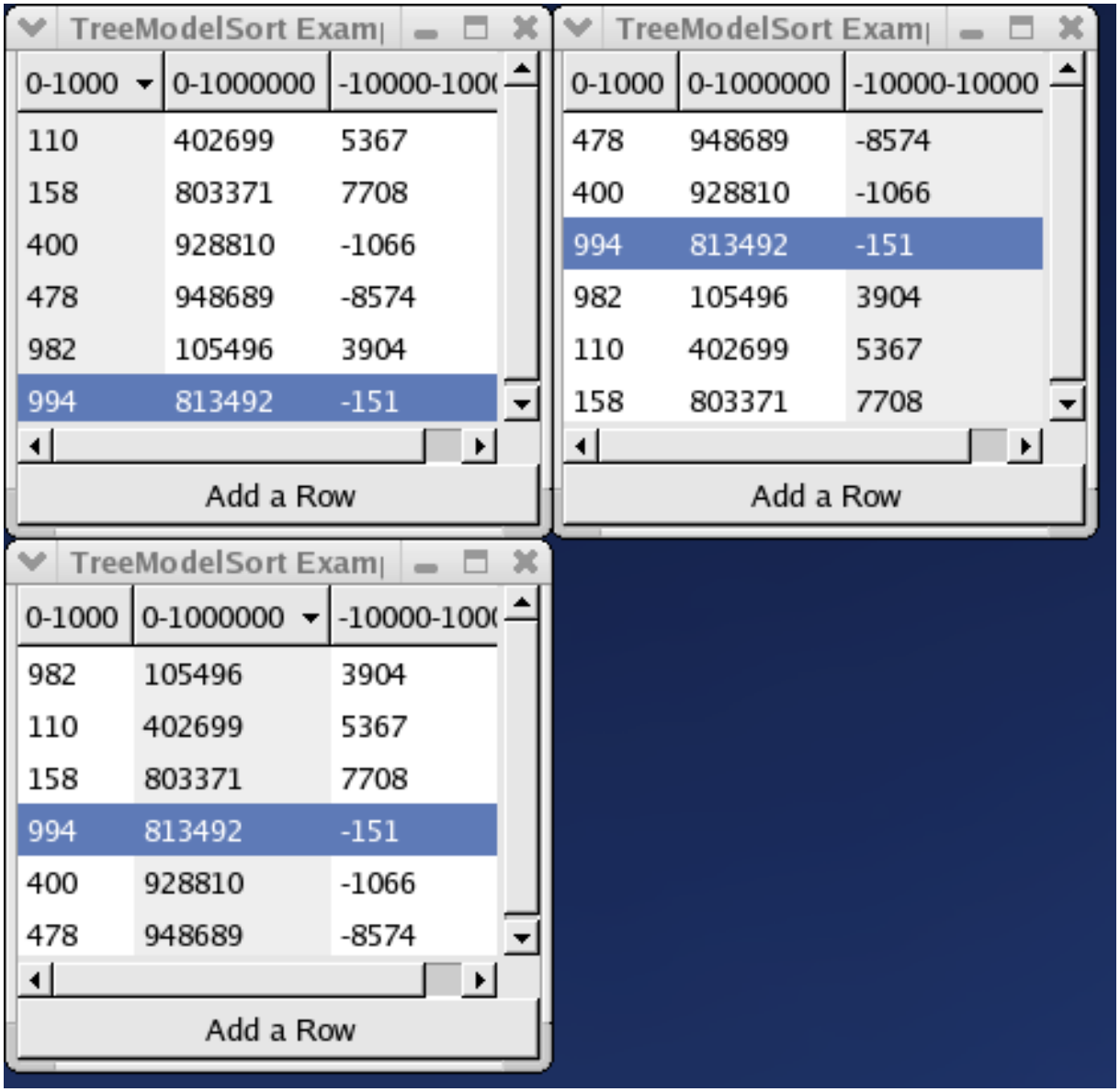
```
sorted_iter = treemodelsort.convert_child_iter_to_iter(None, child_iter)
child_iter = treemodelsort.convert_iter_to_child_iter(None, sorted_iter)
```

Al igual que los métodos de conversión de caminos, estos métodos devuelven None si el TreeIter dado no puede ser convertido.

Se puede obtener el TreeModel hijo usando el método get_model().

Un ejemplo sencillo que utiliza un objeto TreeModelSort es treemodelsort.py. Figura 14.9 ilustra el resultado de ejecutar el programa y añadir seis filas:

Figura 14.9 Ejemplo de TreeModelSort



En cada una de las columnas de las ventanas se puede hacer click para cambiar el orden de los elementos de forma independiente al de otras ventanas. Cuando se pulsa el botón "Add a Row" se añade una nueva fila al `ListStore` base y la nueva fila se muestra en cada `TreeView` como la fila seleccionada.

14.10.2. TreeModelFilter (Modelo de árbol filtrado)

NOTA

El objeto `TreeModelFilter` está disponible a partir de la versión 2.4 de PyGTK y posteriores.

Un objeto `TreeModelFilter` proporciona varias formas de modificar la visualización del `TreeModel` de base, y permiten:

- mostrar un subconjunto de las filas del modelo hijo en base a datos booleanos en una "columna

visible", o en función del valor de retorno de una "función visible", que toma el modelo hijo, un iterador `TreeIter` que apunta a la fila del modelo hijo, y datos de usuario. En ambos casos si el valor booleano es `TRUE` la fila se mostrará; de otro modo, la fila quedará oculta.

- usar un nodo raiz virtual para proporcionar una vista de un subárbol de los desdendientes de una fila en el modelo hijo. Esto únicamente tiene sentido si el almacén subyacente es del tipo `TreeStore`.
- sintetizar las columnas y datos de un modelo en base a los datos del modelo hijo. Por ejemplo, se puede proporcionar una columna cuyos datos son calculados a partir de los datos de varias columnas del modelo hijo.

Un objeto `TreeModelFilter` se crea usando el método de `TreeModel`:

```
treemodelfilter = treemodel.filter_new(root=None)
```

donde *root* es un camino de árbol en *treemodel* que especifica la raiz virtual del modelo o `None` si se va a usar el nodo raíz de *treemodel*.

Al establecer una "raíz virtual" cuando se crea el `TreeModelFilter`, se puede limitar la vista del modelo a las filas hijas de la fila "raíz" en la jerarquía del modelo hijo. Esto, naturalmente, solamente es útil cuando el modelo hijo está basado en un `TreeStore`. Por ejemplo, si se desea proporcionar una vista parcial del contenido de una unidad de CDROM, independiente del resto de elementos del ordenador.

Los modos de visibilidad son mútuamente excluyentes y únicamente se pueden fijar una vez. Es decir, una vez que se ha establecido una función o columna de visibilidad no puede ser cambiada y el modo alternativo no puede establecerse. El modo de visibilidad más simple extrae un valor booleano de una columna en el modelo hijo para determinar si la fila debería mostrarse. La columna de visibilidad se detemina usando:

```
treemodelfilter.set_visible_column(column)
```

donde *column* es el número de columna en el `TreeModel` hijo, del que extraer los valores booleanos. Por ejemplo, el siguiente fragmento de código usa los valores de la tercera columna para determinar la visibilidad de las filas:

```
...
treestore = gtk.TreeStore(str, str, "gboolean")
...
modelfilter = treestore.filter_new()
modelfilter.set_visible_column(2)
...
```

De esta manera, se mostrará cualquier fila de *treestore* que tenga un valor `TRUE` en la tercera columna.

Si se tiene criterios de visibilidad más complicados, una función de visibilidad debería proporcionar suficientes posibilidades:

```
treemodelfilter.set_visible_func(func, data=None)
```

donde *func* es la función llamada para cada una de las filas del modelo hijo y determinar si se debe mostrar y *data* son datos de usuario pasados a *func*. *func* debería devolver `TRUE` si la fila debe ser mostrada. La signatura de *func* es:

```
def func(model, iter, user_data)
```

donde *model* es el `TreeModel` hijo, *iter* es un `TreeIter` que apunta a una fila en *model* y *user_data* son los datos *data* entregados.

Si se hacen cambios en los criterios de visibilidad se debería hacer la llamada:

```
treemodelfilter.refilter()
```

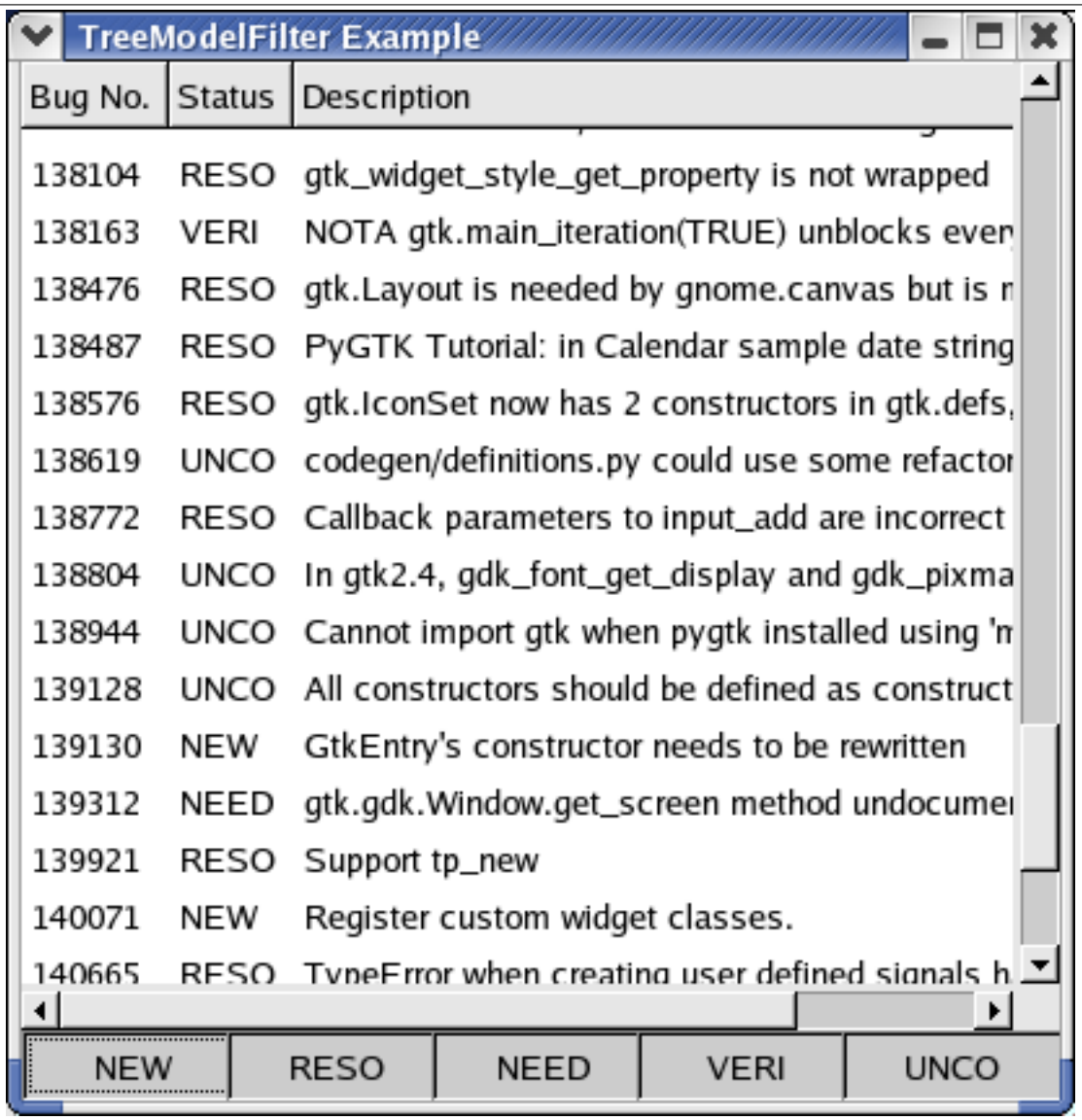
para forzar un refiltrado de las filas del modelo hijo.

Por ejemplo, el siguiente fragmento de código ilustra el uso de un `TreeModelFilter` que muestra filas en base a una comparación entre el valor de la tercera columna y los contenidos de los datos de usuario:

```
...
def match_type(model, iter, udata):
    value = model.get_value(iter, 2)
    return value in udata
...
show_vals = ['OPEN', 'NEW', 'RESO']
liststore = gtk.ListStore(str, str, str)
...
modelfilter = liststore.filter_new()
modelfilter.set_visible_func(match_type, show_vals)
...
```

El programa treemodelfilter.py ilustra el uso del método set_visible_func(). Figura 14.10 muestra el resultado de la ejecución del programa.

Figura 14.10 Ejemplo de Visibilidad en TreeModelFilter



Conmutando los botones de la parte inferior se cambian los contenidos del `TreeView` para mostrar únicamente las filas que coinciden con uno de los botones activos.

Una función de modificación proporciona otro nivel de control sobre la visualización en el `TreeView` hasta el punto en el que es posible sintetizar una o más columnas (incluso todas), que son representadas por el `TreeModelFilter`. Todavía es preciso usar un modelo hijo que sea un almacén `TreeStore` o

ListStore para determinar el número de filas y la jerarquía, pero las columnas pueden ser las que se especifiquen en el método:

```
treemodelfilter.set_modify_func(types, func, data=None)
```

donde *types* es una secuencia (lista or tupla) que especifica los tipos de las columnas que se representan, *func* es una función llamada para devolver el valor para una fila y columna y *data* es un argumento que pasar a *func*. La signatura de *func* es:

```
def func(model, iter, column, user_data)
```

donde *model* es el TreeModelFilter, *iter* es un TreeIter que apunta a una fila del modelo, *column* es el número de la columna para el que se precisa un valor y *user_data* es el parámetro *data*. *func* debe devolver un valor que coincida con el tipo de *column*.

Una función de modificación es útil cuando se quiere proporcionar una columna de datos que necesita ser generada utilizando los datos de las columnas del modelo hijo. Por ejemplo, si se tuviese una columna que contiene fechas de nacimiento y se quisiese mostrar una columna de edades, una función de modificación podría generar la información de edad usando la fecha de nacimiento y la fecha actual. Otro ejemplo sería el decidir qué imagen mostrar en base al análisis de los datos (por ejemplo, el nombre de un archivo) de una columna. Este efecto también se puede conseguir utilizando el método TreeViewColumn set_cell_data_func().

Generalmente, dentro de la función de modificación, se tendrá que convertir el TreeModelFilter TreeIter a un iterador TreeIter del modelo hijo haciendo uso de:

```
child_iter = treemodelfilter.convert_iter_to_child_iter(filter_iter)
```

Naturalmente, también es necesario obtener el modelo hijo usando:

```
child_model = treemodelfilter.get_model()
```

Estos métodos dan acceso a la fila del modelo hijo y a sus valores para generar el valor de la fila y columna especificadas del TreeModelFilter. También hay un método para convertir un TreeIter hijo a un TreeIter de un modelo filtrado y métodos para convertir caminos del modelo filtrado a y desde caminos de los árboles hijo:

```
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)
```

Naturalmente, es posible combinar los modos de visibilidad y la función de modificación para filtrar y sintetizar columnas. Para obtener incluso un mayor control sobre la vista sería necesario utilizar un TreeModel personalizado.

14.11. El Modelo de Árbol Genérico (GenericTreeModel)

En el momento en el que se encuentra que los modelo de árbol estándar TreeModels no son suficientemente potentes para cubrir las necesidades de una aplicación se puede usar la clase GenericTreeModel para construir modelos TreeModel personalizados en Python. La creación de un GenericTreeModel puede ser de utilidad cuando existen problemas de rendimiento con los almacenes estándar TreeStore y ListStore o cuando se desea tener una interfaz directa con una fuente externa de datos (por ejemplo, una base de datos o el sistema de archivos) para evitar la copia de datos dentro y fuera de los almacenes TreeStore o ListStore.

14.11.1. Visión general de GenericTreeMode

Con GenericTreeModel se construye y gestiona un modelo propio de datos y se proporciona acceso externo a través de la interfaz estándar TreeModel al definir un conjunto de métodos de clase. PyGTK implementa la interfaz TreeModel y hace que los nuevos métodos TreeModel sean llamados para proporcionar los datos del modelo existente.

Los detalles de implementación del modelo personalizado deben mantenerse ocultos a la aplicación externa. Ello significa que la forma en la que se identifica el modelo, guarda y obtiene sus datos es

desconocido a la aplicación. En general, la única información que se guarda fuera del `GenericTreeModel` son las referencia de fila que se encapsulan por los iteradores `TreeIter` externos. Y estas referencias no son visibles a la aplicación.

Examinemos en detalle la interfaz `GenericTreeModel` que es necesario proporcionar.

14.11.2. La Interfaz GenericTreeModel

La interfaz `GenericTreeModel` consiste en los siguientes métodos que deben implementarse en el modelo de árbol personalizado: `def on_get_flags(self)`

```
def on_get_n_columns(self)
def on_get_column_type(self, index)
def on_get_iter(self, path)
def on_get_path(self, rowref)
def on_get_value(self, rowref, column)
def on_iter_next(self, rowref)
def on_iter_children(self, parent)
def on_iter_has_child(self, rowref)
def on_iter_n_children(self, rowref)
def on_iter_nth_child(self, parent, n)
def on_iter_parent(self, child)
```

Debe advertirse que estos métodos dan soporte a toda la interfaz de `TreeModel` incluída: `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)
```

Para ilustrar el uso de `GenericTreeModel` modificaremos el programa de ejemplo filelisting.py y veremos cómo se crean los métodos de la interfaz. El programa filelisting-gtm.py muestra los archivos de una carpeta con un pixbuf que indica si el archivo es una carpeta o no, el nombre de archivo, el tamaño del mismo, el modo y hora del último cambio.

El método on_get_flags() debería devolver un valor que es una combinación de:

`gtk.TREE_MODEL_ITERS_PERSIST` Los `TreeIter`s sobreviven a todas las señales emitidas por el árbol.

`gtk.TREE_MODEL_LIST_ONLY` El modelo es solamente una lista, y nunca tiene hijos

Si el modelo tiene referencias de fila que son válidas entre cambios de fila (reordenación, adición o borrado) entonces se dbe establecer `gtk.TREE_MODEL_ITERS_PERSIST`. Igualmente, si el modelo es una lista entonces de debe fijar `gtk.TREE_MODEL_LIST_ONLY`. De otro modo, se debe devolver 0 si el modelo no tiene referencias de fila persistentes y es un modelo de árbol. Para nuestro ejemplo, el modelo es una lista con iteradores `TreeIter` persitentes.

```
    def on_get_flags(self):
        return gtk.TREE_MODEL_LIST_ONLY|gtk.TREE_MODEL_ITERS_PERSIST
```

El método on_get_n_columns() debería devolver el número de columnas que el modelo exporta a la aplicación. Nuestro ejemplo mantiene una lista de los tipos de las columnas, de forma que devolvemos la longitud de la lista:

```
class FileListModel(gtk.GenericTreeModel):
    ...
    column_types = (gtk.gdk.Pixbuf, str, long, str, str)
    ...
    def on_get_n_columns(self):
        return len(self.column_types)
```

El método on_get_column_type() debería devolver el tipo de la columna con el valor de índice *index* especificado. Este método es llamado generalmente desde una `TreeView` cuando se establece su modelo. Se puede, bien crear una lista o tupla que contenga la información de tipos de datos de las columnas o bien generarla al vuelo. En nuestro ejemplo:

```
    def on_get_column_type(self, n):
        return self.column_types[n]
```

La interfaz `GenericTreeModel` convierte el tipo de Python a un `GType`, de forma que el siguiente código:

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

mostraría:

```
<type 'str'> <GType gchararray (64)>
```

Los siguientes métodos usan referencias de fila que se guardan como datos privados en un `TreeIter`. La aplicación no puede ver la referencia de fila en un `TreeIter` por lo que se puede usar cualquier elemento único que se desee como referencia de fila. Por ejemplo, en un modelo que contiene filas como tuplas, se podría usar la identificación de tupla como referencia de la fila. Otro ejemplo sería el uso de un nombre de archivo como referencia de fila en un modelo que represente los archivos de un directorio. En ambos casos la referencia de fila no se modifica con los cambios en el modelo, así que los `TreeIter`s se podrían marcar como persistentes. La interfaz de aplicación de PyGTK `GenericTreeModel` extraerá esas referencias de fila desde los `TreeIter`s y encapsulará las referencias de fila en `TreeIter`s según sea necesario.

En los siguientes métodos `rowref` se refiere a una referencia de fila interna.

El método on_get_iter() debería devolver una rowref del camino de árbol indicado por *path*. El camino de árbol se representará siempre mediante una tupla. Nuestro ejemplo usa la cadena del nombre de archivo como rowref. Los nombres de archivo se guardan en una lista en el modelo, de manera que tomamos el primer índice del camino como índice al nombre de archivo:

```
    def on_get_iter(self, path):
        return self.files[path[0]]
```

Es necesario ser coherente en el uso de las referencias de fila, puesto que se obtendrán referencias de fila tras las llamadas a los métodos de `GenericTreeModel` que toman argumentos con iteradores `TreeIter`: on_get_path(), on_get_value(), on_iter_next(), on_iter_children(), on_iter_has_child(), on_iter_n_children(), on_iter_nth_child() y on_iter_parent().

El método on_get_path() debería devolver un camino de árbol dada una rowref. Por ejemplo, siguiendo con el ejemplo anterior donde el nombre de archivo se usa como rowref, se podría definir el método on_get_path() así:

```
    def on_get_path(self, rowref):
        return self.files.index(rowref)
```

Este método localiza el índice de la lista que contiene el nombre de archivo en *rowref*. Es obvio viendo este ejemplo que una elección juiciosa de la referencia de fila hará la implementación más eficiente. Se podría usar, por ejemplo, un diccionario de Python para traducir una *rowref* a un camino.

El método on_get_value() debería devolver los datos almacenados en la fila y columna especificada por *rowref* y la columna *column*. En nuestro ejemplo:

```
    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)
```

tiene que extraer la información de archivo asociada y devolver el valor adecuado en función de qué columna se especifique.

El método on_iter_next() debería devolver una rerferencia de fila a la fila (en el mismo nivel) posterior a la especificada por *rowref*. En nuestro ejemplo:

```
    def on_iter_next(self, rowref):
        try:
            i = self.files.index(rowref)+1
            return self.files[i]
        except IndexError:
            return None
```

El índice del nombre de archivo *rowref* es determinado y se devuelve el siguiente nombre de archivo o `None` si no existe un archivo después.

El método on_iter_children() debería devolver una referencia de fila a la primera fila hija de la fila especificada por *rowref*. Si *rowref* es `None` entonces se devuelve una referencia a la primera fila de nivel superior. Si no existe una fila hija se devuelve `None`. En nuestro ejemplo:

```
    def on_iter_children(self, rowref):
        if rowref:
            return None
        return self.files[0]
```

Puesto que el modelo es una lista únicamente la fila superior puede tener filas hijas (*rowref*=`None`). Se devuelve `None` si *rowref* contiene un nombre de archivo.

El método on_iter_has_child() debería devolver `TRUE` si la fila especificada por *rowref* tiene filas hijas o `FALSE` en otro caso. Nuestro ejemplo devuelve `FALSE` puesto que ninguna fila puede tener hijas:

```
    def on_iter_has_child(self, rowref):
        return False
```

El método on_iter_n_children() debería devolver el número de fijas hijas que tiene la fila especificada por *rowref*. Si *rowref* es `None` entonces se devuelve el número de las filas de nivel superior. Nuestro ejemplo devuelve 0 si *rowref* no es `None`:

```
    def on_iter_n_children(self, rowref):
        if rowref:
            return 0
        return len(self.files)
```

El método on_iter_nth_child() debería devolver una referencia de fila a la n-ésima fila hija de la fila especificada por *parent*. Si *parent* es `None` entonces se devuelve una referencia a la n-ésima fila de nivel superior. Nuestro ejemplo devuelve la n-ésima fila de nivel superior si *parent* es `None`. En otro caso devuelve `None`:

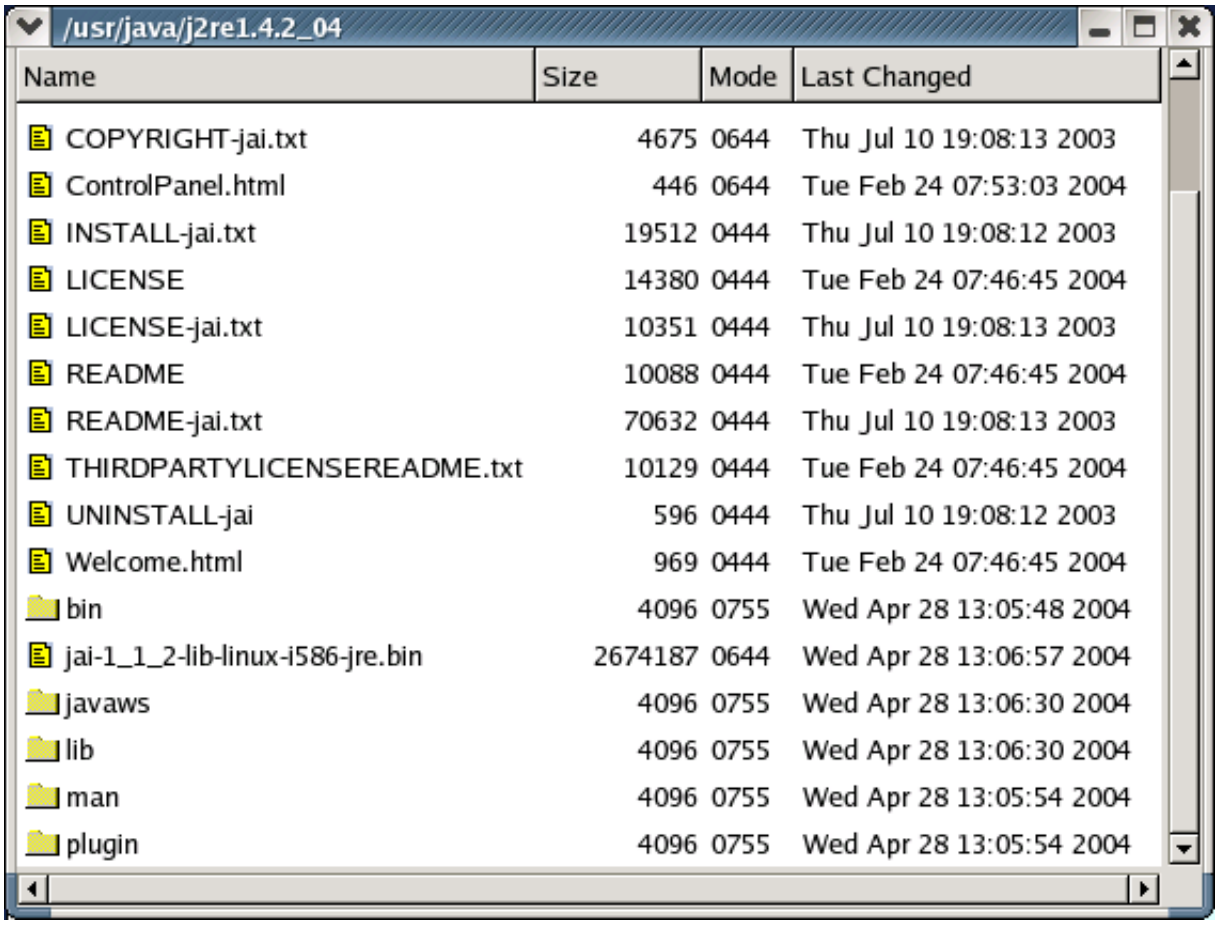
```
    def on_iter_nth_child(self, rowref, n):
        if rowref:
            return None
        try:
            return self.files[n]
        except IndexError:
            return None
```

El método on_iter_parent() debería devolver una referencia de fila a la fila padre de la fila especificada por *rowref*. Si *rowref* apunta a una fila de nivel superior se debe devolver `None`. Nuestro ejemplo siempre devuelve `None` asumiendo que *rowref* debe apuntar a una fila de nivel superior:

```
    def on_iter_parent(child):
        return None
```

Este ejemplo se ve de una vez en el programa filelisting-gtm.py. Figura 14.11 muestra el resultado de la ejecución del programa.

Figura 14.11 Programa de Ejemplo de Modelo de Árbol Genérico



14.11.3. Adición y Eliminación de Filas

El programa filelisting-gtm.py calcula la lista de nombres de archivo mientras se crea una instancia de `FileListModel`. Si se desea comprobar la existencia de nuevos archivos de forma periódica y añadir o eliminar archivos del modelo se podría, bien crear un nuevo modelo `FileListModel` de la misma carpeta, o bien se podrían incorporar métodos para añadir y eliminar filas del modelo. Dependiendo del tipo de modelo que se esté creando se necesitaría añadir unos métodos similares a los de los modelos `TreeStore` y `ListStore`:

- insert()

- insert_before()
- insert_after()
- prepend()
- append()
- remove()
- clear()

Naturalmente, ni todos ni cualquiera de estos métodos neceista ser implementado. Se pueden crear métodos propios que se relacionen de manera más ajustada al modelo.

Utilizando el programa de ejemplo anterior para ilustrar la adición de métodos para eliminar y añadir archivos, implementemos esos métodos: `def remove(iter)`
`def add(filename)`

El método remove() elimina el archivo especificado por *iter*. Además de eliminar la fila del modelo el método también debería eliminar el archivo de la carpeta. Naturalmente, si el usuario no tiene permisos para eliminar el archivo no se debería eliminar tampoco la fila. Por ejemplo:

```
def remove(self, iter):
    path = self.get_path(iter)
    pathname = self.get_pathname(path)
    try:
        if os.path.exists(pathname):
            os.remove(pathname)
        del self.files[path[0]]
        self.row_deleted(path)
    except OSError:
        pass
    return
```

Al método se le pasa un iterador `TreeIter` que ha de ser convertido a un camino que se usa para obtener el camino del archivo usando el método get_pathname(). Es posible que el archivo ya haya sido eliminado por lo que debemos comprobar si existe antes de intentar su eliminación. Si se emite una excepción OSError durante la eliminación del archivo, probablemente se deba a que es un directorio o que la usuaria no tiene los privilegios necesarios para elimnarlo. Finalmente, el archivo se elimina y la señal "row-deleted" se emite desde el método rows_deleted(). La señal "file-deleted" notifica a las `TreeViews` que usan el modelo que éste ha cambiado, por lo que pueden actualizar su estado interno y mostrar el modelo actualizado.

El método add() necesita crear un archivo con el nombre dado en la carpeta actual. Si se crea el archivo su nombre se añade a la lista de archivos del modelo. Por ejemplo:

```
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
```

Este ejemplo sencillo se asegura de que el archivo no existe y luego intenta abrir el archivo para escritura. Si tiene éxito, el archivo se cierra y el nombre de archivo ordenado en la lista de archivos. La ruta y el iterador `TreeIter` de la fila de archivo añadida se obtienen para usarlos en el método

row_inserted() que emite la señal "row-inserted". La señal "row-inserted" se usa para notificar a las TreeViews que usan el modelo que necesitan actualizar su estado interno y revisar su visualización.

Los otros métodos mencionados anteriormente (por ejemplo, append y prepend) no tienen sentido en el ejemplo, puesto que el modelo mantiene la lista de archivos ordenada.

Otros métodos que puede merecer la pena implementar en un TreeModel que herede de GenericTreeModel son:

- set_value()
- reorder()
- swap()
- move_after()
- move_before()

La implementación de estos métodos es similar a de los métodos anteriores. Es necesario sincronizar el modelo con el estado externo y luego notificar a las TreeViews cuando el modelo cambie. Los siguientes métodos se usan para notificar a las TreeViews de cambios en el modelo emitiendo la señal apropiada: def row_changed(*path*, *iter*)
fila cambió def row_inserted(*path*, *iter*)
fila insertada def row_has_child_toggled(*path*, *iter*)
cambio en la existencia de hijas en la fila def row_deleted(*path*)
fila eliminada def rows_reordered(*path*, *iter*, *new_order*)
filas reordenadas

14.11.4. Gestión de Memoria

Uno de los problemas de GenericTreeModel es que los TreeIters guardan una referencia a un objeto de Python devuelto por el modelo de árbol personalizado. Puesto que se puede crear e inicializar un TreeIter en código en C y que permanezca en la pila, no es posible saber cuándo se ha destruido el TreeIter y ya no se usa la referencia al objeto de Python. Por lo tanto, el objeto de Python referenciado en un TreeIter incrementa por defecto su cuenta de referencias, pero no se decrementa cuando se destruye el TreeIter. Esto asegura que el objeto de Python no será destruido mientras está en uso por un TreeIter, lo que podría causar una violación de segmento. Desgraciadamente, la cuenta de referencias extra lleva a la situación en la que, como bueno, el objeto de Python tendrá un contador de referencias excesivo, y como malo, nunca se liberará esa memoria incluso cuando ya no se usa. Este último caso lleva a pérdidas de memoria y el primero a pérdidas de referencias.

Para prever la situación en la que el TreeModel personalizado mantiene una referencia al objeto de Python hasta que ya no se dispone de él (es decir, el TreeIter ya no es válido porque el modelo ha cambiado) y no hay necesidad de perder referencias, el GenericTreeModel tiene la propiedad "leak-references". Por defecto "leak-references" es TRUE para indicar que el GenericTreeModel pierde referencias. Si "leak-references" se establece a FALSE entonces el contador de referencias del objeto de Python no se incrementará cuando se referencia en un TreeIter. Esto implica que el TreeModel propio debe mantener una referencia a todos los objetos de Python utilizados en los TreeIters hasta que el modelo sea destruido. Desgraciadamente, incluso esto no permite la protección frente a código erróneo que intenta utilizar un TreeIter guardado en un GenericTreeModel diferente. Para protegerse frente a este caso la aplicación debería mantener referencias a todos los objetos de Python referenciados desde un TreeIter desde cualquier instancia de un GenericTreeModel. Naturalmente, esto tiene finalmente el mismo resultado que la pérdida de referencias.

En PyGTK 2.4 y posteriores los métodos invalidate_iters() y iter_is_valid() están disponibles para ayudar en la gestión de los TreeIters y sus referencias a objetos de Python:

```
generictreemodel.invalidate_iters()

result = generictreemodel.iter_is_valid(iter)
```

Estas son particularmente útiles cuando la propiedad "leak-references" está fijada como FALSE. Los modelos de árbol derivados de GenericTreeModel están protegidos de problemas con TreeIters obsoletos porque se comprueba automáticamente la validez de los iteradores con el modelo de árbol.

Si un modelo de árbol personalizado no soporta iteradores persistentes (es decir, `gtk.TREE_MODEL_ITERS_PERSIST` no está activado en el resultado del método TreeModel.get_flags()) entonces puede llamar al método invalidate_iters() para invalidar los `TreeIter`s restantes cuando cambia el modelo (por ejemplo, tras insertar una fila). El modelo de árbol también puede deshacerse de cualquier objeto de Python que fue referenciado por `TreeIter`s tras llamar al método invalidate_iters().

Las aplicaciones pueden usar el método iter_is_valid() para determinar si un `TreeIter` es aún válido en el modelo personalizado.

14.11.5. Otras Interfaces

Los modelos `ListStore` y `TreeStore` soportan las interfaces `TreeSortable`, `TreeDragSource` y `TreeDragDest` además de la interfaz `TreeModel`. La clase `GenericTreeModel` unicamente soporta la interfaz `TreeModel`. Esto parece ser así dada la referencia directa al modelo en el nivel de C por las vistas `TreeView` y los modelos `TreeModelSort` y `TreeModelFilter` models. La creación y uso de `TreeIter`s precisa código de unión en C que haga de enlace con el modelo de árbol personalizado en Python que tiene los datos. Ese código de conexión lo aporta la clase `GenericTreeModel` y parece que no hay una forma alternativa de hacerlo puramente en Python puesto que las `TreeView`s y los otros modelo llaman a las funciones de GtkTreeModel en C y pasan sus referencias al modelo de árbol personalizado.

La interfaz `TreeSortable` también necesitaría código de enlace en C para funcionar con el mecanismo de ordenación predeterminado de `TreeViewColumn`, que se explica en la sección Ordenación de Filas del Modelo de Árbol. Sin embargo, un modelo personalizado puede hacer su propia ordenación y una aplicación puede gestionar el uso de los criterios de ordenación manejando los clic sobre las cabeceras de las `TreeViewColumn`s y llamando a los métodos personalizados de ordenación del modelo. El modelo completa la actualización de las vistas `TreeView` emitiendo la señal "rows-reordered" utilizando el método de `TreeModel` rows_reordered(). Así, probablemente no es necesario que `GenericTreeModel` implemente la interfaz `TreeSortable`.

De la misma manera, la clase `GenericTreeModel` no necesita implementar las interfaces `TreeDragSource` y `TreeDragDest` puesto que el modelo de árbol personalizado puede implementar sus propias interfaces de arrastrar y soltar y la aplicación puede majenar las señales de `TreeView` adecuadas y llamar a los métodos propios del modelo según sea necesario.

14.11.6. Utilización de GenericTreeModel

La clase `GenericTreeModel` debería usarse como último recurso. Existen mecanismos muy poderosos en el grupo estándar de objetos `TreeView` que deberían ser suficientes para la mayor parte de aplicaciones. Sin duda que existen aplicaciones que pueden requerir el suo de `GenericTreeModel` pero se debería probar antes a utilizar lo siguiente:

Funciones de Datos de Celda Como se ilustra en la sección Función de Datos de Celda, las funciones de datos de celda se pueden usar para modificar e incluso generar los datos de una columna de `TreeView`. Se pueden crear de forma efectiva tantas columnas con datos generados como se precise. Esto proporciona un gran control sobre la presentación de los datos a partir de una fuente de datos subyacente.

Modelo de Árbol Filtrado (TreeModelFilter) En PyGTK 2.4, el `TreeModelFilter` que se describe en la sección TreeModelFilter proporciona un gran nivel de control sobre la visualización de las columnas y filas de un `TreeModel` hijo, incluida la presentación de únicamente las filas hijas de una fila. Las columnas de datos también pueden ser generadas de forma similar al caso de las Funciones de Datos de Celda, pero aquí el modelo parece ser un `TreeModel` con el número y tipo de columnas especificado, mientras que la función de datos de celda deja intactas las columnas del modelo y simplemente modifica la visualización en una vista `TreeView`.

Si se acaba usando un `GenericTreeModel` se debe tener en cuenta que:

- la interfaz `TreeModel` completa debe ser creada y debe funcionar tal como se documentó. Hay sutilezas que pueden conducir a errores. Por el contrario, los `TreeModel`s estándar están muy revisados.

- la gestión de referencias a objetos de Python utilizados por iteradores `TreeIter` puede ser complicada, especialmente en el caso de programas que se ejecuten durante mucho tiempo y con gran variedad de visualizaciones.
- es necesaria la adición de una interfaz para añadir, borrar y cambiar los contenidos de las filas. Hay cierta complicación con la traducción de los `TreeIter` a objetos de Python y a las filas del modelo en esta interfaz.
- la implementación de las interfaces de ordenación y arrastrar y soltar exigen un esfuerzo considerable. La aplicación posiblemente necesite implicarse en hacer que estas interfaces sean completamente funcionales.

14.12. El Visualizador de Celda Genérico (GenericCellRenderer)

Capítulo 15

Nuevos Controles de PyGTK 2.2

El objeto `Clipboard` (portapapeles) ha sido añadido en PyGTK 2.2. GtkClipboard estaba disponible en GTK+ 2.0 pero no era incluido en PyGTK 2.0 debido a que no se trataba de un `GObject` completo. Se añadieron también algunos objetos nuevos al módulo gtk.gdk en PyGTK 2.2 pero no se describirán en este tutorial. Véase el Manual de Referencia de PyGTK 2 para obtener información adicional sobre los objetos `gtk.gdk.Display`, `gtk.gdk.DisplayManager` y `gtk.gdk.Screen`.

15.1. Portapapeles (Clipboard)

Un `Clipboard` (portapapeles) aporta una zona de almacenamiento que permite compartir datos entre procesos o entre distintos controles del mismo proceso. Cada `Clipboard` se identifica con un nombre (una cadena de texto) codificada como un `gdk.Atom`. Es posible utilizar cualquier nombre para identificar un portapapeles (`Clipboard`) y se creará uno nuevo en caso de que no exista. Para que se pueda compartir un objeto `Clipboard` entre distintos procesos, es preciso que cada uno de ellos conozca el nombre del mismo.

Los elementos`Clipboard` están hechos sobre las interfaces de selección y `SelectionData`. El portapapeles usado por defecto por los controles `TextView`, `Label` y `Entry` es "CLIPBOARD". Otros portapapeles comunes son "PRIMARY" y "SECONDARY", que se corresponden con las selecciones primaria y secundaria (Win32 las ignora). Estos objetos pueden igualmente ser especificados utilizando los objetos `gtk.gdk.Atom` siguientes: `gtk.gdk.SELECTION_CLIPBOARD`, `gtk.gdk.SELECTION_PRIMARY` y `gtk.gdk.SELECTION_SECONDARY`. Para mayor información véase la documentación de referencia de gtk.gdk.Atom.

15.1.1. Creación de un objeto Clipboard (Portapapeles)

Los objetos `Clipboard` se crean mediante el constructor:

```
clipboard = gtk.Clipboard(display, selection)
```

donde *`display`* es el `gtk.gdk.Display` asociado con el `Clipboard` denominado por *`selection`*. La función auxiliar que sigue crea un portapapeles (`Clipboard`) haciendo del `gtk.gdk.Display` por defecto:

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

Finalmente, un `Clipboard` también puede crearse utilizando el método de la clase `Widget`:

```
clipboard = widget.get_clipboard(selection)
```

El control widget debe haber sido realizado y ha de formar parte de una ventana de una jerarquía de ventana de primer nivel.

15.1.2. Utilización de Clipboards con elementos Entry, Spinbutton y TextView

Los controles `Entry`, `SpinButton` y `TextView` poseen menús emergentes que proporcionan la capacidad de copiar y pegar el texto seleccionado desde el portapapeles 'CLIPBOARD' así como pegarlo

desde el mismo. Además, se dispone de combinaciones de teclas que sirven de atajos para cortar, copiar y pegar. Cortar se activa con Control+X; copiar con Control+C; y pegar con Control+V.

Esos controles (`Entry` y `SpinButton`) implementan la interfaz `Editable`, de manera que poseen los siguientes métodos para cortar, copiar y pegar de y hacia el portapapeles "CLIPBOARD":

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

Una etiqueta `Label` que sea seleccionable (su propiedad "selectable" es `TRUE`) también permite la copia del texto seleccionado al portapapeles "CLIPBOARD" utilizando un menú emergente o la combinación de teclas rápidas Control+C.

Los `TextBuffer`s poseen métodos simiares, aunque permiten indicar el portapapeles que se ha de usar:

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

El texto de la selección será copiado al portapapeles especificado por *`clipboard`*.

```
textbuffer.cut_clipboard(clipboard, default_editable)
```

El texto seleccionado será copiado a *`clipboard`*. Si *`default_editable`* es `TRUE`, entonces el texto seleccionado será borrado del `TextBuffer`. En otro caso, cut_clipboard() funcionará como el método copy_clipboard().

```
textbuffer.paste_clipboard(clipboard, override_location, default_editable)
```

Si *`default_editable`* es `TRUE`, los contenidos del portapapeles *`clipboard`* serán insertados en el `TextBuffer` en la posición indicada por el iterador `TextIter` *`override_location`*. Si *`default_editable`* es `FALSE`, entonces paste_clipboard() no insertará los contenidos de *`clipboard`*. Si *`override_location`* es `None` los contenidos del portapapeles *`clipboard`* se insertarán en la posición del cursor.

Los `TextBuffer`s también tienen dos métodos para gestionar un conjunto de objetos `Clipboard` que se establecen de forma automática con los contenidos de la selección vigente:

```
textbuffer.add_selection_clipboard(clipboard) # añadir portapapeles de ↩
    selección
textbuffer.remove_selection_clipboard(clipboard) # eliminar portapapeles de ↩
    selección
```

Cuando se añade un `TextBuffer` a una vista de texto (`TextView`) se añade automáticamente el portapapeles "PRIMARY" a los portapapeles de selección. La aplicación puede añadir otros portapapeles a conveniencia (por ejemplo, el portapapeles "CLIPBOARD").

15.1.3. Incorporación de datos en un portapapeles

Se pueden establecer los datos de un portapapeles `Clipboard` de forma programática utilizando los métodos:

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

clipboard.set_text(text, len=-1)
```

El método set_with_data() indica que destinos se soportan para los datos seleccionados y proporciona las funciones (*`get_func`* y *`clear_func`*), que se llaman al solicitar o cuando cambian los datos del portapapeles. *`user_data`* se pasa a *`get_func`* o *`clear_func`* cuando son llamadas. *`targets`* es una lista de tuplas de tres elementos que contiene:

- una cadena que representa un destino soportado por el portapapeles.
- un valor de banderas utilizado para arrastrar y soltar (úsese 0).
- un entero asignado por la aplicación que se pasa como parámetro al manejador de señal para ayudar en la identificación del tipo de destino.

Las signaturas de *`get_func`* y *`clear_func`* son:

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

def clear_func(clipboard, data):
```

donde *clipboard* es el portapapeles de la clase Clipboard, *selectiondata* es un objeto SelectionData en el que poner los datos, *info* es el entero asignado por la aplicación asignado al destino, y *data* son datos de usuario *user_data*.

set_text() es un método auxiliar que usa a su vez el método set_with_data() para asignar los datos de texto en un portapapeles Clipboard con los destinos: "STRING" (cadena), "TEXT" (texto), "COMPOUND_TEXT" (texto compuesto), y "UTF8_STRING" (cadena UTF-8). Usa funciones internas get y clear para gestionar los datos. set_text() equivale a:

```
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
```

Una vez que se introducen datos en un portapapeles, éstos estarán disponibles hasta que se termine el programa o se cambien los datos en el portapapeles.

Para proporcionar el comportamiento típico de cortar al portapapeles la aplicación tendrá que eliminar el texto u objeto correspondiente tras copiarlo al portapapeles.

15.1.4. Obtención de los Contenidos del Portapapeles

Los contenidos de un portapapeles Clipboard se pueden obtener utilizando los siguientes métodos:

```
clipboard.request_contents(target, callback, user_data=None)
```

Los contenidos especificados por *target* son obtenidos de forma asíncrona en la función indicada por *callback*, que es llamada con *user_data*. La signatura de *callback* es:

```
def callback(clipboard, selectiondata, data):
```

donde *selectiondata* es un objeto SelectionData con los contenidos del portapapeles *clipboard*. *data* son datos de usuario *user_data*. El método request_contents() es la forma más general de obtener los contenidos de un portapapeles Clipboard. El siguiente método auxiliar recupera los contenidos de texto de un portapapeles Clipboard:

```
clipboard.request_text(callback, user_data=None)
```

Se devuelve la cadena de texto a la retrollamada en vez de un objeto Selectiondata. Se puede comprobar los destinos disponibles para el portapapeles Clipboard utilizando el método:

```
clipboard.request_targets(callback, user_data=None)
```

Los destinos se devuelven a la función de retrollamada como tuplas de objetos gtk.gdk.Atom.

Se proporcionan dos métodos auxiliares que devuelven los contenidos del portapapeles Clipboard de forma síncrona:

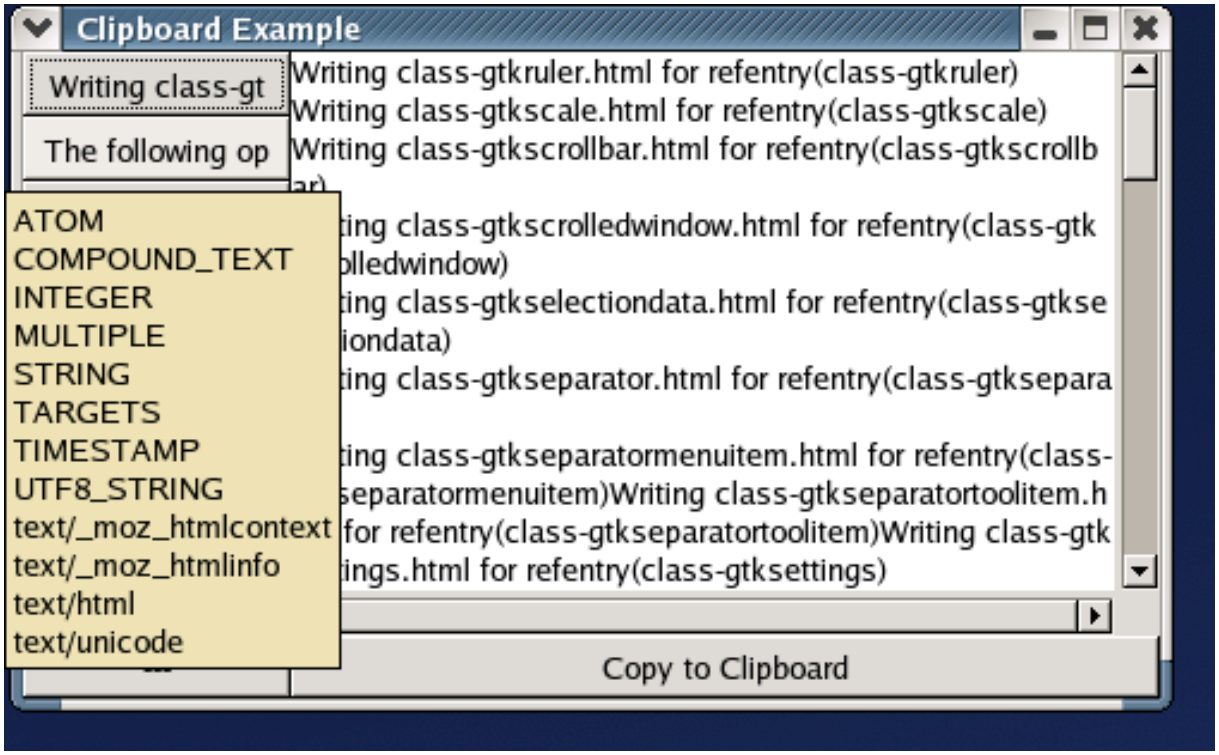
```
selectiondata = clipboard.wait_for_contents(target)

text = clipboard.wait_for_text()
```

15.1.5. Ejemplo de Portapapeles

Para ilustrar el uso de un portapapeles `Clipboard` el programa de ejemplo clipboard.py sigue los elementos de texto que se copian o cortan al cortapapeles "CLIPBOARD" y guarda las últimas diez entradas del cortapapeles. Hay diez botones que proporcionan acceso al texto de las entradas guardadas. La etiqueta del botón muestra los primeros dieciseis caracteres del texto almacenado y las etiquetas de ayuda (pistas) muestran los destinos que tenía la entrada original. Cuando se pulsa un botón de entrada se carga el texto correspondiente, que puede ser editado. A su vez, ell botón bajo la ventana de texto permite guardar los contenidos de la ventana de texto al portapapeles..

Figura 15.1 ilustra el resultado de la ejecución del programa clipboard.py:

Figura 15.1 Programa de ejemplo de Portapapeles



El programa de ejemplo consulta el portapapeles cada 1.5 segundos para ver si los contenidos han cambiado. El programa podría modificarse para duplicar el conjunto completo de destinos disponibles y tomar luego control del portapapeles utilizando el método set_with_data(). Posteriormente, si otro programa fija los contenidos del portapapeles, se llamará a la función `clear_func`, que se puede usar para recargar los contenidos del portapapeles y retomar la propiedad del mismo.

Capítulo 16

Nuevos Controles de PyGTK 2.4

En PyGTK 2.4 se han añadido unos cuantos controles y objetos auxiliares nuevos, incluídos:

- `Action (Acción)`, `RadioAction` (Acción de Exclusión Mútua), `ToggleAction` (Acción Bi-estado) - objetos que representan acciones que puede tomar la usuaria. Las acciones contienen información necesaria para crear controles delegados (proxy) (por ejemplo: iconos, elementos de menú y elementos de barra de herramientas).
- `ActionGroup` (Grupo de Acciones) - un objeto que contiene Actions que están relacionadas de alguna manera, por ejemplo, acciones para abrir, cerrar e imprimir un documento.
- `Border` (Borde) - un objeto que contiene los valores de un borde.
- `ColorButton` (Botón de Color) - un botón usado para laznar un diálogo de selección de Color (ColorSelectionDialog).
- `ComboBox` (Lista desplegable) - un control que proporciona una lista de elementos entre los que elegir. Sustituye al menú de opciones `OptionMenu`.
- `ComboBoxEntry` (Lista con entrada) - un control que proporciona un campo de entrada de texto con una lista desplegable de elementos entre los que elegir. Remplaza al control de lista desplegable `Combo`.
- `EntryCompletion` (Entrada con completado) - un objeto que proporciona completado a un control de entrada `Entry`.
- `Expander` (Elemento de Expansión) - un contenedor que puede mostrar u ocultar sus hijos en respuesta a su pulsación de botón.
- `FileChooser` (Selector de Ficheros) - una interfaz para seleccionar ficheros.
- `FileChooserWidget` (Control de Selección de Ficheros) - un control que implementa la interfaz `FileChooser`. Sustituye al control `FileSelection`.
- `FileChooserDialog` (Diálogo de Selección de Ficheros) - un diálogo utilizado para las acciónes "Archivo/Abrir" y "Archivo/Guardar". Sustituye a `FileSelectionDialog`.
- `FileFilter` (Filtro de Ficheros) - un objeto usado para filtrar archivos en base a un conjunto interno de reglas.
- `FontButton` (Botón de Fuentes) - un botón que lanza el diálogo de selección de fuentes `FontSelectionDialog`.
- `IconInfo` (Información de Icono) - un objeto que contiene información sobre un icono de un tema `IconTheme`.
- `IconTheme` (Tema de Iconos) - un objeto que proporciona búsqueda de iconos por nombre y tamaño.

- `ToolItem` (Elemento genérico), `ToolButton` (Botón), `RadioToolButton` (Botón de exclusión), `SeparatorToolItem` (Separador), `ToggleToolButton` (Botón biestado) - controles que se pueden añadir a una barra de herramientas `Toolbar`. Todos estos sustituyen a los anteriores elementos de `Toolbar`.
- `TreeModelFilter` (Modelo de Árbol Filtrado) - un objeto que proporciona un potente mecanismo para transformar la representación de un modelo `TreeModel` subyacente. Se describe en la sección Sección de TreeModelFilter.
- `UIManager` (Gestor de Interfaz de Usuario) - un objeto que proporciona una manera de construir menús y barras de herramientas a partir de una descripción de la interfaz en XML. También tiene métodos para gestionar la combinación y separación de múltiples descripciones de interfaces de usuario.

16.1. Objetos de Acción (Action) y Grupo de Acciones (ActionGroup)

Los objetos `Action` y `ActionGroup` trabajan de forma conjunta para proveer a las aplicaciones de imágenes, texto, retrollamadas y aceleradores a los menús y barras de herramientas. El objeto `UIManager` utiliza objetos `Action` y `ActionGroup` para construir de forma automática dichas barras de menú y herramientas a partir de su descripción en XML. Así, resulta mucho más sencilla la creación y construcción de menús y barras de herramientas mediante `UIManager` de la manera en la que se describe en una sección posterior. Las secciones que siguen, acerca de los objetos `Action` y `ActionGroup`, describen cómo se pueden utilizar éstos directamente, aunque, en su lugar y siempre que sea posible, se recomienda el uso de la clase `UIManager`.

16.1.1. Acciones (Actions)

Un objeto `Action` (acción) representa una acción que el usuario puede realizar utlizando la interfaz de una aplicación. Contiene información utilizada por los elementos intermedios de la interfaz (por ejemplo, elementos de menú `MenuItem` o elementos de barra de herramientas `Toolbar`) para presentar dicha acción al usuario. Existen dos subclases de la clase `Action`:

ToggleAction (acción conmutable) Es una acción `Action` que permite conmutar entre dos estados.

RadioAction (acción con exclusión) Una acción `Action` que puede agruparse de manera que solamente una de ellas puede estar activa.

Por ejemplo, el elemento de menú estándar Archivo → Salir puede ser representado mediante un icono, un texto mnemónico y un acelerador. Cuando se activa, el elemento de menú dispara una retrollamada que podría cerrar la aplicación. De la misma manera, un botón Salir de una barra de herramientas `Toolbar` podría compartir con aquel el icono, el texto mnemónico y la retrollamada. Ambos elementos de la interfaz podrían ser elementos delegados (proxies) de la misma acción `Action`.

Los controles de botón normal (`Button`), botón conmutado (`ToggleButton`) y botón de exclusión (`RadioButton`) pueden actuar como delegados (proxies) de una acción (`Action`), aunque no existe soporte para ellos en la clase `UIManager`.

16.1.1.1. Creación de acciones

Las acciones `Action` se crean haciendo uso del constructor:

```
action = gtk.Action(name, label, tooltip, stock_id)
```

`name` (nombre) es una cadena que identifica la acción (`Action`) dentro de un grupo (`ActionGroup`) o en la especificación de un gestor de interfaz (`UIManager`). *`label`* y *`tooltip`* son cadenas de texto usadas respectivamente como etiqueta y texto de ayuda en los objetos delegados (proxy). Si *`label`* es `None` entonces *`stock_id`* ha de ser una cadena de texto que especifique un elemento de serie (Stock item) del que se obtendrá la etiqueta. Si *`tooltip`* es `None`, entonces la acción (`Action`) no dispondrá de texto de ayuda.

Como veremos en la sección correspondiente a grupos de acciones (ActionGroups) es mucho más fácil la creación de objetos de acción utilizando algunos métodos de la clase `ActionGroup`:

```
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)
```

Más adelante profundizaremos en éstos, aunque primeramente describiremos cómo usar una acción (`Action`) con un botón (`Button`) para ilustrar las operaciones básicas necesarias para conectar una acción (`Action`) a un objeto delegado.

16.1.1.2. Uso de acciones

El procedimiento básico para usar una acción (`Action`) con un botón (`Button`) como objeto delegado se ilustra en el programa de ejemplo simpleaction.py. El botón (`Button`) se conecta a la acción (`Action`) con el método:

```
action.connect_proxy(proxy)
```

donde proxy es un control delegado del tipo: elemento de menú `MenuItem`, elemento de barra de herramientas `ToolItem` o botón `Button`.

Una acción `Action` tiene una señal, la señal "activate" que se dispara cuando la acción (`Action`) se activa, generalmente como resultado de la activación de un control delegado (por ejemplo, cuando se pulsa sobre un botón de una barra de herramientas `ToolButton`). Simplemente se debe conectar una retrollamada a esta señal para gestionar la activación de cualquiera de los controles delegados.

Este es el código fuente del programa de ejemplo simpleaction.py:

```
 1  #!/usr/bin/env python
 2
 3  import pygtk
 4  pygtk.require('2.0')
 5  import gtk
 6
 7  class SimpleAction:
 8      def __init__(self):
 9          # Creación de la ventana principal
10          window = gtk.Window()
11          window.set_size_request(70, 30)
12          window.connect('destroy', lambda w: gtk.main_quit())
13
14          # Creación de un grupo de aceleradores
15          accelgroup = gtk.AccelGroup()
16          # Añadimos el grupo de aceleradores a la ventana principal
17          window.add_accel_group(accelgroup)
18
19          # Creación de una acción para salir del programa usando un elemento de serie
20          action = gtk.Action('Quit', None, None, gtk.STOCK_QUIT)
21          # Conexión de una retrollamada a la acción
22          action.connect('activate', self.quit_cb)
23
24          # Creación de un grupo ActionGroup llamado SimpleAction
25          actiongroup = gtk.ActionGroup('SimpleAction')
26          # Adición de la acción al grupo con un acelerador
27          # None significa que se utilice el acelerador el elemento de serie
28          actiongroup.add_action_with_accel(action, None)
29
30          # Hacer que la acción use el grupo de aceleradores
31          action.set_accel_group(accelgroup)
32
33          # Conectar el acelerador a la acción
34          action.connect_accelerator()
35
36          # Crear el botón para usarlo como control delegado para la acción
37          quitbutton = gtk.Button()
```

```
38            # añadimos el botón a la ventana principal
39            window.add(quitbutton)
40
41            # Conectamos la acción a su control delegado
42            action.connect_proxy(quitbutton)
43
44            window.show_all()
45            return
46
47        def quit_cb(self, b):
48            print 'Saliendo del programa'
49            gtk.main_quit()
50
51    if __name__ == '__main__':
52        sa = SimpleAction()
53        gtk.main()
```

El ejemplo crea una acción (`Action`) (línea 20) que usa un elemento de serie que aporta el texto de etiqueta, con un mnemónico, un icono, un acelerador y un dominio de traducción. Si no se usa un elemento de serie, sería necesario especificar en su lugar una etiqueta. La línea 22 conecta la señal "activate" del la acción *action* al método self.quit_cb() de forma que sea invocado cuando la acción `Action` es activada por el botón *quitbutton*. La línea 42 conecta *quitbutton* a *action* como control delegado. Cuando se pulsa sobre *quitbutton* éste activará *action* y, por tanto, llamará al método self.quit_cb(). El ejemplo simpleaction.py usa bastante código (líneas 15, 17, 31 y 34 para establecer el acelerador del botón). El proceso es similar para los elementos `MenuItem` y `ToolItem`.

Figura 16.1 muestra el resultado de la ejecución del programa simpleaction.py.

Figura 16.1 Ejemplo Simple de Acción

16.1.1.3. Creación de Controles intermedios (Proxy)

En el apartado anterior previo vimos cómo es posible la conexión de un control existente a una acción (`Action`) como elemento delegado. En éste veremos como es posible crear un control delegado utilizando los métodos de la clase `Action`:

```
menuitem = action.create_menu_item()

toolitem = action.create_tool_item()
```

El ejemplo basicaction.py ilustra una acción `Action` compartida por un elemento de menú (`MenuItem`), un botón de barra de herramientas (`ToolButton`) y un botón ordinario (`Button`). El elemento de menú (`MenuItem`) y el botón de la barra de herramientas (`ToolButton`) se crearán usando los métodos indicados. El código fuente del programa de ejemplo basicaction.py es el siguiente:

```
1   #!/usr/bin/env python
2
3   import pygtk
4   pygtk.require('2.0')
5   import gtk
6
7   class BasicAction:
8       def __init__(self):
9           # Creación de la ventana principal
10          window = gtk.Window()
11          window.connect('destroy', lambda w: gtk.main_quit())
12          vbox = gtk.VBox()
13          vbox.show()
```

```
14          window.add(vbox)
15
16          # Creación de un grupo de aceleradores
17          accelgroup = gtk.AccelGroup()
18          # Adición del grupo a la ventana principal
19          window.add_accel_group(accelgroup)
20
21          # Creación de una acción para salir del programa utilizando un ↩
    elemento de serie
22          action = gtk.Action('Quit', '_Quit me!', 'Quit the Program',
23                              gtk.STOCK_QUIT)
24          action.set_property('short-label', '_Quit')
25          # Conexión de una retrollamada a la acción
26          action.connect('activate', self.quit_cb)
27
28          # Creación de un grupo ActionGroup llamado BasicAction
29          actiongroup = gtk.ActionGroup('BasicAction')
30          # Agregamos la acción al grupo con un acelerador
31          # None significa que se usa el acelerador del elemento de serie
32          actiongroup.add_action_with_accel(action, None)
33
34          # Hacemos que la acción use el grupo de aceleradores
35          action.set_accel_group(accelgroup)
36
37          # Creación de una barra de menú
38          menubar = gtk.MenuBar()
39          menubar.show()
40          vbox.pack_start(menubar, False)
41
42          # Creación de la acción para Archivo y el elemento de menú
43          file_action = gtk.Action('File', '_File', None, None)
44          actiongroup.add_action(file_action)
45          file_menuitem = file_action.create_menu_item()
46          menubar.append(file_menuitem)
47
48          # Creación del menú de Archivo
49          file_menu = gtk.Menu()
50          file_menuitem.set_submenu(file_menu)
51
52          # Creación de un elemento delegado
53          menuitem = action.create_menu_item()
54          file_menu.append(menuitem)
55
56          # Creación de una barra de herramientas
57          toolbar = gtk.Toolbar()
58          toolbar.show()
59          vbox.pack_start(toolbar, False)
60
61          # Creación de un elemento delegado de la barra de herramientas
62          toolitem = action.create_tool_item()
63          toolbar.insert(toolitem, 0)
64
65          # Creación y empaquetado de un control de Etiqueta
66          label = gtk.Label('''
67   Select File->Quit me! or
68   click the toolbar Quit button or
69   click the Quit button below or
70   press Control+q
71   to quit.
72   ''')
73          label.show()
74          vbox.pack_start(label)
75
76          # Creación de un botón para usarlo como otro control delegado
```

```
77            quitbutton = gtk.Button()
78            # add it to the window
79            vbox.pack_start(quitbutton, False)
80
81            # Conexión de la acción a su control delegado
82            action.connect_proxy(quitbutton)
83            # Se establece el texto de ayuda después de que se añade el  ↩
    elemento a la barra
84            action.set_property('tooltip', action.get_property('tooltip'))
85            tooltips = gtk.Tooltips()
86            tooltips.set_tip(quitbutton, action.get_property('tooltip'))
87
88            window.show()
89            return
90
91        def quit_cb(self, b):
92            print 'Saliendo del programa'
93            gtk.main_quit()
94
95    if __name__ == '__main__':
96        ba = BasicAction()
97        gtk.main()
```

Este ejemplo presenta un grupo de acciones `ActionGroup` como contenedor de las acciones `Action` que se usan en el programa. El apartado sobre grupos de acciones ActionGroups se adentrará en mayor detalle en su uso.

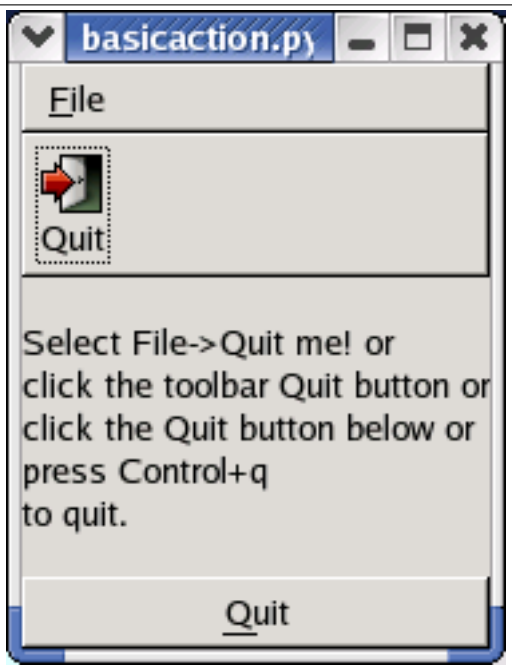
El código de las líneas 9-14 establece una ventana principal que contiene una `VBox`. Las líneas 16-35 establecen la acción `Action` "Quit" (salir) de la misma manera que en el programa de ejemplo simple-action.py y la añaden con el acelerador de serie `gtk.STOCK_QUIT` (línea 32) al grupo de acción `ActionGroup` "BasicAction" (creado en la línea 29). Es de advertir el hecho de que, a diferencia del ejemplo simpleaction.py, aquí no es necesario llamar al método connect_accelerator() para la acción puesto que se hace de forma automática cuando se llama al método create_menu_item() en la línea 53.

Las líneas 38-40 crean una barra de menú (`MenuBar`) y la empaquetan en la `VBox`. Las líneas 43-44 crean una acción `Action` (*`file_action`*) para el menú File y la añaden al grupo *`actiongroup`*. Los elementos de menú File y Quit son creados en las líneas 45 y 53 y añadidos a la barra de menús (*`menubar`*) y al menú (*`file_menu`*) respectivamente en las líneas 46 y 54.

Igualmente se crea una barra de herramientas (`Toolbar`) que se añade a la `VBox` en las líneas 57-59. El elemento de la barra de herramientas `ToolItem` que actúa como delegado se crea y añade a la barra de herramientas en las líneas 62-63. Ha de advertirse que el texto de ayuda de la acción `Action` debe establecerse (línea 84) tras haber añadido el elemento de la barra de herramientas `ToolItem` a la propia barra de herramientas (`Toolbar`) para que pueda ser usado. De la misma manera, el texto de ayuda del botón `Button` debe añadirse de forma manual (líneas 84-86).

Figura 16.2 muestra el programa de ejemplo basicaction.py en acción:

Figura 16.2 Ejemplo Básico de Acción



Es posible desconectar un control delegado (proxy) de una acción (`Action`) usando el método:

```
action.disconnect_proxy(proxy)
```

16.1.1.4. Propiedades de Acción

Una acción `Action` tiene una serie de propiedades que controlan la representación y funcionamiento de sus controles delegados. Las más importantes entre ellas son las propiedades "sensitive" y "visible". La propiedad "sensitive" determina la sensibilidad de los controles delegados. Si "sensitive" es `FALSE` los controles delegados no pueden activarse y normalmente se representarán engrisecidos. De la misma manera, la propiedad "visible" determina si los controles delegados son o no visibles. Si la propiedad "visible" de una acción `Action` es `FALSE` entonces sus controles delegados permanecerán ocultos.

Como veremos en el siguiente apartado, la sesibilidad o visibilidad de una acción `Action` también está controlada por la sensibilidad o visibilidad del grupo de acciones `ActionGroup` al que pertenece. Por lo tanto, para que una acción `Action` sea visible o sensible tanto sus propiedades como las del grupo `ActionGroup` al que pertenece deben concordar. Para establecer la sensibilidad o visibilidad efectiva de una acción `Action` se deben usar los siguientes métodos:

```
result = action.is_sensitive()

result = action.is_visible()
```

El nombre asignado a una acción `Action` se almacena en su propiedad "name", que se establece en el momento en que se crea la acción `Action`. Se puede obtener dicho nombre usando el método:

```
name = action.get_name()
```

Otras propiedades que controlan la visualización de los controles delegados de una acción `Action` son:

"hide-if-empty" Si es `TRUE`, se ocultan los delegados de menú vacíos para esta acción.

"is-important" Si es `TRUE`, los elementos de barra de herramientas `ToolItem` delegados para esta acción muestran su texto en el modo `gtk.TOOLBAR_BOTH_HORIZ`.

"visible-horizontal" Si es `TRUE`, el elemento de barra de herramientas `ToolItem` es visible cuando la barra de herramientas tiene orientación horizontal.

"visible-vertical" Si es `TRUE`, el elemento de la barra de herramientas `ToolItem` es visible cuando la barra de herramientas tiene orientación vertical.

Otras propiedades de interés son:

"label" La etiqueta usada para los elementos de menú y botones que activan esta acción.

"short-label" Una etiqueta más breve que puede usarse en botones de barras de herramientas y en botones.

"stock-id" El Elemento de Serie utilizado para obtener el icono, etiqueta y acelerador que se usará en los controles que representen esta acción.

"tooltip" Un texto de ayuda para esta acción.

Adviértase que el programa de ejemplo basicaction.py anula la etiqueta de `gtk.STOCK_QUIT` sustituyéndola por "_Quit me!" y fija la propiedad "short-label" a "_Quit". La etiqueta corta se usa para las etiquetas del botón de barra de herramientas `ToolButton` y para la del botón `Button`, pero se usa la etiqueta completa en el caso del elemento de menú `MenuItem`. También es de reseñar que no es posible establecer el texto de ayuda de un elemento `ToolItem` hasta que es añadido a su barra de herramientas (`Toolbar`).

16.1.1.5. Acciones y Aceleradores

Una acción `Action` tiene tres métodos usados para establecer un acelerador:

```
action.set_accel_group(accel_group)

action.set_accel_path(accel_path)

action.connect_accelerator()
```

Éstos, además del método gtk.ActionGroup.add_action_with_accel(), deberían ser suficientes en los casos en los que se haya de establecer un acelerador.

Un grupo de aceleración `AccelGroup` debe estabecerse siempre para una acción`Action`. El método set_accel_path() es llamado por el método gtk.ActionGroup.add_action_with_accel(). Si se usa set_accel_path() entonces el camino del acelerador debería utilizar el formato por defecto: "<Actions>/nombre_del_grupo_de_acciones/nom Finalmente, el método connect_accelerator() es llamado para completar la configuración de un acelerador.

NOTA

Una acción `Action` debe tener un grupo `AccelGroup` y un camino de acelerador asociado a él antes de la llamada a connect_accelerator().

Puesto que el método connect_accelerator() puede ser llamado varias veces (por ejemplo, una vez por cada control delegado), el número de llamadas es almacenado, de forma que se produzca un número igual de llamadas al método disconnect_accelerator() antes de que se elimine el acelerador.

Tal como se ilustró en los programas de ejemplo anteriores, un acelerador de una acción `Action` puede ser usado por todos los controles delegados. Una acción `Action` debería ser parte de un grupo `ActionGroup` para poder usar el camino de acelerador por defecto, que tiene el formato: "<Actions>/nombre_del_grupo_de_acciones/nombre_de_la_acción". La forma más sencilla de añadir un acelerador es usar el método gtk.ActionGroup.add_action_with_accel() y el siguiente método general:

- Crear un grupo de aceleradores `AccelGroup` y añadirlo a la ventana principal.
- Crear un nuevo grupo de acciones `ActionGroup`
- Crear una acción `Action` que especifique un elemento de Serie con un acelerador.
- Añadir la acción `Action` al grupo `ActionGroup` utilizando el método gtk.ActionGroup.add_action_with_accel() especificando `None` para usar el acelerador del elemento de Serieo una cadena de acelerador aceptable por la función `gtk.accelerator_parse`().

- Establecer el grupo `AccelGroup` para la acción `Action` usando el método gtk.Action.set_accel_group().
- Completar la configuración del acelerador usando el método gtk.Action.connect_accelerator().

Cualquier control delegado creado por o conectado a la acción `Action` elegida utilizará el acelerador así establecido.

16.1.1.6. Acciones Conmutables

Tal como se mencionó previamente, una acción `ToggleAction` es una subclase de `Action` que permite la conmutación entre dos estados. El constructor de una acción del tipo `ToggleAction` requiere los mismos parámetros que una del tipo `Action`:

```
toggleaction = gtk.ToggleAction(name, label, tooltip, stock_id)
```

Además de los métodos de la clase `Action`, también dispone de los siguientes métodos propios de `ToggleAction`:

```
toggleaction.set_active(is_active)
is_active = toggleaction.get_active()
```

establecer u obtener el valor actual de *`toggleaction`*. *`is_active`* es un valor booleano.
Es posible conectarse a la señal "toggled" especificando una retrollamada con la signatura:

```
def toggled_cb(toggleaction, user_data)
```

La señal "toggled" se emite cuando un elemento de acción `ToggleAction` cambia su estado.
Un control elemento de menú `MenuItem` delegado de una acción `ToggleAction` se mostrará por defecto como un elemento de menú con marca de verificación (`CheckMenuItem`). Para que un elemento delegado `MenuItem` se muestre como un elemento de menú del tipo exclusión (`RadioMenuItem`) se ha de establecer la propiedad "draw-as-radio" como `TRUE` usando el método:

```
toggleaction.set_draw_as_radio(draw_as_radio)
```

Se puede usar el siguiente método para determinar si los elementos de menú `MenuItem` de una acción conmutable `ToggleAction` se mostrarán como elementos de menú del tipo exclusión `RadioMenuItem`:

```
draw_as_radio = toggleaction.get_draw_as_radio()
```

16.1.1.7. Acciones con Exclusión

Una acción con exclusión `RadioAction` es una subclase de una acción conmutable `ToggleAction` que puede ser agrupada de manera que solamente una de las acciones `RadioAction` está activa al mismo tiempo. Los controles delegados correspondiente son `RadioMenuItem` y `RadioToolButton` para barras de menú y de herramientas, respectivamente.
El constructor de una acción `RadioAction` posee los mismos argumentos que una acción de la clase `Action`, con el añadido de un valor entero único que se usa para identificar la acción `RadioAction` dentro de un grupo:

```
radioaction = gtk.RadioAction(name, label, tooltip, stock_id, value)
```

El grupo de una `RadioAction` se puede establecer con el método:

```
radioaction.set_group(group)
```

donde *`group`* es otra acción del tipo `RadioAction` al que se debe agrupar *`radioaction`*. El grupo que contiene una acción `RadioAction` puede obtenerse con el método:

```
group = radioaction.get_group()
```

que devuelve la lista de objetos `RadioAction` que incluye *`radioaction`*.
El valor del miembro actualmente activo del grupo puede obtenerse con el método:

```
active_value = radioaction.get_current_value()
```

Se puede conectar una retrollamada a la señal "changed" para recibir notificación del momento en el que cambia el miembro activo del grupo de acciones `RadioAction`. Nótese que únicamente es necesario conectarse a uno de los objetos `RadioAction` para seguir los cambios. La signatura de la retrollamada es:

```
def changed_cb(radioaction, current, user_data)
```

donde *current* es el elemento `RadioAction` actualmente activo en el grupo.

16.1.1.8. Un ejemplo de Acciones

El programa de ejemplo actions.py ilustra el uso de los objetos de acción (`Action`), acción conmutable (`ToggleAction`) y acción con exclusión (`RadioAction`). Figura 16.3 muestra el programa de ejemplo en funcionamiento:

Figura 16.3 Ejemplo de Acciones

Este ejemplo es suficientemente parecido a basicaction.py como para que no sea precisa una descripción detallada del mismo.

16.1.2. Grupos de Acciones (ActionGroups)

Tal como se mencionó en el apartado anterior, los objetos `Action` relacionados deberían añadirse a un grupo `ActionGroup` de manera que se disponga de un control conjunto sobre su visibilidad y sensibilidad. Por ejemplo, en una aplicación de procesamiento de textos, los elementos de menú y los botones de la barra de herramientas que especifican la justificación del texto podrían estar agrupados en un `ActionGroup`. Es de esperar que una interfaz de usuario tenga múltiples objetos `ActionGroup` que abarquen los diversos aspectos de la aplicación. Por ejemplo, las acciones globales tales como la creación de nuevos documentos, abrir y guardar un documento y salir de la aplicación probablemente formen un grupo `ActionGroup`, mientras que acciones tales como la modificación de la vista del documento formarían otro.

16.1.2.1. Creación de grupos de acciones (ActionGroups)

Un `ActionGroup` se crea mediante el constructor:

```
actiongroup = gtk.ActionGroup(name)
```

donde *name* es un nombre único para el grupo `ActionGroup`. El nombre debería ser único puesto que se usa para construir el camino de acelerador por defecto de sus objetos `Action`.

El nombre de un *ActionGroup* puede obtenerse usando el método:

```
name = actiongroup.get_name()
```

u obteniendo el contenido de la propiedad "name".

16.1.2.2. Adición de Acciones

Tal como se ilustra en el apartado Acciones, se puede añadir una acción `Action` existente a un grupo `ActionGroup` utilizando uno de los métodos siguientes:

```
actiongroup.add_action(action)

actiongroup.add_action_with_accel(action, accelerator)
```

donde *action* es la acción `Action` que se va a añadir y *accelerator* es una especificación de cadena de acelerador que resulte aceptable a la función `gtk.accelerator_parse()`. Si *accelerator* es `None` se usará el acelerador (si existe) asociado con la propiedad "stock-id" de *action*. Tal como se indicó anteriormente, el método add_action_wih_accel() es el preferible si se desea utilizar aceleradores.

El grupo `ActionGroup` ofrece tres métodos para hacer más sencilla la creación y adición de objetos de acción (`Action`) a un grupo de acción (`ActionGroup`):

```
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)
```

El parámetro *entries* es una secuencia de tuplas de entradas de acciones que aportan la información necesaria para crear las acciones que se añaden al grupo `ActionGroup`. El objeto `RadioAction` con el valor *value* se fija inicialmente como activo. *on_change* es una retrollamada que está conectada a la señal "changed" del primer objeto `RadioAction` del grupo. La signatura de *on_changed* es:

```
def on_changed_cb(radioaction, current, user_data)
```

Las tuplas de entradas de objetos `Action` contienen:

- El nombre de la acción. Debe especificarse.
- Un identificador de elemento estándar (stock id) para la acción. Opcionalmente puede tener el valor por defecto `None` si se especifica una etiqueta.
- La etiqueta para la acción. Opcionalmente puede tener el valor por defecto `None` si se ha especificado un identificador de elemento estándar (stock id).
- El acelerador para la acción, en el formato comprensible para la función `gtk.accelerator_parse()`. Opcionalmente puede tener el valor por defecto de `None`.
- El texto de ayuda de la acción. Opcionalmente puede tener el valor por defecto de `None`.
- La función de retrollamada invocada cuando se activa la acción activated. Opcionalmente puede tener el valor por defecto `None`.

Como mínimo se debe especificar un valor para el campo *name* y un valor bien en el campo *stock id* o el campo *label*. Si se especifica una etiqueta entonces se puede indicar `None` como identificador de elemento estándar (stock id) si no se usa uno. Por ejemplo, la siguiente llamada al método:

```
actiongroup.add_actions([('quit', gtk.STOCK_QUIT, '_Quit me!', None,
                          'Quit the Program', quit_cb)])
```

añade una acción a *actiongroup* para salir del programa.

Las tuplas de entradas para los objetos `ToggleAction` son similares a las tuplas de entradas para objetos `Action` salvo que existe un campo optativo adicional *flag* que contiene un valor booleano que indica si la acción está activa. El valor por defecto del campo *flag* es `FALSE`. Por ejemplo, la siguiente llamada:

```
actiongroup.add_toggle_actions([('mute, None, '_Mute', '<control>m',
                                 'Mute the volume', mute_cb, True)])
```

añade un objeto `ToggleAction` al grupo *actiongroup* y lo configura inicialmente para que esté activo.

Las tuplas de entradas para objetos `RadioAction` son semejantes a las de los objetos `Action` pero incorporan un campo *value* en vez del campo *callback*:

- El nombre de la acción. Debe ser especificado.
- Un identificador de elemento estándar (stock id) para la acción. Opcionalmente puede tener el valor por defecto `None` si se especifica una etiqueta.
- La etiqueta para la acción. Opcionalmente puede tener el valor por defecto `None` si se ha especificado un identificador de elemento estándar (stock id).
- El acelerador para la acción, en el formato comprensible para la función `gtk.accelerator_parse()`. Opcionalmente puede tener el valor por defecto de `None`.
- El texto de ayuda de la acción. Opcionalmente puede tener el valor por defecto de `None`.
- El valor (*value*) que se debe tener el objeto de acción con exclusión. Opcionalmente tiene el valor por defecto de `0`. Siempre se debería especificar en las aplicaciones.

Por ejemplo, el siguiente fragmento de código:

```
radioactionlist = [('am', None, '_AM', '<control>a', 'AM Radio', 0)
                   ('fm', None, '_FM', '<control>f', 'FM Radio', 1)
                   ('ssb', None, '_SSB', '<control>s', 'SSB Radio', 2)]
actiongroup.add_radio_actions(radioactionlist, 0, changed_cb)
```

crea tres objetos `RadioAction` y establece 'am' como la acción inicialmente activa y `changed_cb` como la retrollamada empleada cuando cualquiera de las acciones se active.

16.1.2.3. Obtención de iconos

Para obtener un objeto de acción (`Action`) de un grupo `ActionGroup` se utiliza el método:

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

Es posible otener una lista de objetos `Action` contenidos en un `ActionGroup` con el método:

```
actionlist = actiongroup.list_actions()
```

16.1.2.4. Control de las Acciones

La sensibilidad y visibilidad de todos los objetos `Action` de un grupo de acciones `ActionGroup` puede controlarse estableciendo los valores de los atributos correspondientes. Los siguientes métodos facilitan la obtención y establecimiento de los valores de las propiedades:

```
is_sensitive = actiongroup.get_sensitive()
actiongroup.set_sensitive(sensitive)

is_visible = actiongroup.get_visible()
actiongroup.set_visible(visible)
```

Finally you can remove an `Action` from an `ActionGroup` using the method:

```
actiongroup.remove_action(action)
```

16.1.2.5. Un ejemplo de grupo de acciones (ActionGroup)

El programa de ejemplo actiongroup.py reproduce la barra de menú y barra de herramientas del programa de ejemplo actions.py utilizando los métodos de `ActionGroup`. Además el programa incluye botones para controlar la sensibilidad y visibilidad de los elementos de menú y los elementos de la barra de herramientas. Figura 16.4 muestra el programa el funcionamiento:

Figura 16.4 Ejemplo de ActionGroup

16.1.2.6. Señales de los grupos de acciones (ActionGroup)

Una aplicación puede rastrear la conexión y eliminación de controles delegados en objetos del tipo `Action` dentro de un grupo `ActionGroup` utilizando las señales "connect-proxy" y "diconnect-proxy". Las signaturas de las retrollamadas deben ser:

```
def connect_proxy_cb(actiongroup, action, proxy, user_params)

def disconnect_proxy_cb(actiongroup, action, proxy, user_params)
```

Esto permite que se puedan rastrear cambios que permiten hacer modificaciones adicionales al nuevo control delegado en el momento en que es conectado o para actualizar alguna otra parte de la interfaz de usuario cuando se desconecta uno de los controles delegados.

Las señales "pre-activate" y "post-activate" permiten a las aplicaciones hacer un proceso adicional inmediatamente antes o después de que se haya activado una acción. Las signaturas de las retrollamadas son:

```
def pre_activate_cb(actiongroup, action, user_params)

def post_activate_cb(actiongroup, action, user_params)
```

Estas señales son utilizadas fundamentalmente por la clase `UIManager` para proveer notificación global para todos los objetos `Action` en objetos `ActionGroup` utilizados por ella.

16.2. Controles de Lista Desplegable (ComboBox) y Lista Desplegable con Entrada (ComboBoxEntry)

16.2.1. Controles ComboBox

El control `ComboBox` sustituye el obsoleto `OptionMenu` con un control potente que utiliza un `TreeModel` (generalmente un `ListStore`) que proporciona los elementos de la lista que se mostrarán. El `ComboBox` implementa la interfaz `CellLayout`, que proporcina diversos métodos para gestionar la visualización de los elementos de la lista. Uno o más methods for managing the display of the list items. One or more `CellRenderer`s se pueden empaquetar en un `ComboBox` para personalizar la visualización de los elementos de la lista.

16.2.1.1. Uso Básico de ComboBox

La forma sencilla de crear y poblar un `ComboBox` es utilizar la función auxiliar:

```
combobox = gtk.combo_box_new_text()
```

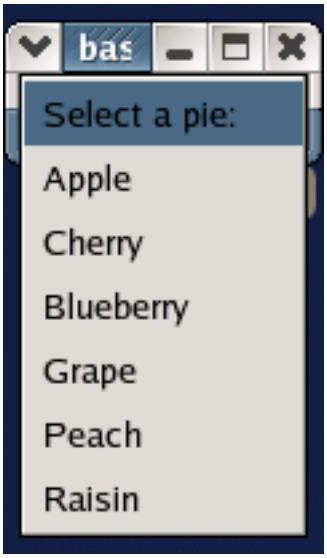
Esta función crea una `ComboBox` y su almacén `ListStore` asociado y lo empaqueta con un `CellRendererText`. Los siguientes métodos auxiliares se usan para poblar o eliminar los contenidos de la `ComboBox` y su `ListStore`:

```
combobox.append_text(text)
combobox.append_text(text)
combobox.insert_text(position, text)
combobox.remove_text(position)
```

donde *text* es la cadena que se añadirá a la ComboBox y *position* es el índice donde se insertará o eliminará el texto *text*. En la mayoría de los casos las funciones y métodos auxiliares es todo lo que se necesitará.

El programa de ejemplo comboboxbasic.py demuestra el uso de las anteriores funciones y métodos. Figura 16.5 ilustra el programa en ejecución:

Figura 16.5 ComboBox Básica



Para obtener el texto activo se puede usar el método:

```
texto = combobox.get_active_text()
```

Sin embargo, hasta la versión 2.6 no se proporciona en GTK+ un método cómodo para obtener el texto activo. Para ello se podría usar una implementación similar a:

```
def get_active_text(combobox):
    model = combobox.get_model()
    active = combobox.get_active()
    if active < 0:
        return None
    return model[active][0]
```

El índice del elemento activo se obtiene a través del método:

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

El elemento activo se puede establecer con el método:

```
combobox.set_active(index)
```

donde *index* es un entero mayor que -2. Si *index* es -1 no hay elemento activo y el control ComboBox estará en blanco. Si *index* es menor que -1, la llamada será ignorada. Si *index* es mayor que -1 el elemento de la lista con dicho índice será mostrado.

Se puede conectar a la señal "changed" de un ComboBox para recibir notificación del cambio del elemento activo. La signatura del manejador de "changed" es:

```
def changed_cb(combobox, ...):
```

donde ... representa cero o más argumentos pasados al método GObject.connect().

16.2.1.2. Uso Avanzado de ComboBox

La creación de una lista `ComboBox` mediante la función `gtk.combo_box_new_text()` es aproximadamente equivalente al siguiente código:

```
liststore = gtk.ListStore(str)
combobox = gtk.ComboBox(liststore)
cell = gtk.CellRendererText()
combobox.pack_start(cell, True)
combobox.add_attribute(cell, 'text', 0)
```

Para sacar partido de la potencia de las variadas clases de objetos `TreeModel` y `CellRenderer` es necesario construir una `ComboBox` utilizando el constructor:

```
combobox = gtk.ComboBox(model=None)
```

donde *`model`* es un modelo `TreeModel`. Si se crea una lista `ComboBox` sin asociarle un `TreeModel` es posible añadirlo a posteriori utilizando el método:

```
combobox.set_model(model)
```

Se puede obtener el `TreeModel` asociado con el método:

```
model = combobox.get_model()
```

Algunas de las cosas que se pueden hacer con una `ComboBox` son:

- Compartir el mismo `TreeModel` con otras `ComboBox`es y `TreeViews`.
- Mostrar imágenes y texto en los elementos de la `ComboBox`.
- Utilizar un `TreeStore` o `ListStore` existente como modelo para los elementos de la lista de la `ComboBox`.
- Utilizar un `TreeModelSort` para disponer de una lista de `ComboBox` ordenada.
- Utilizar un `TreeModelFilter` para usar un subárbol de un `TreeStore` como fuente de elementos de la lista de la `ComboBox`.
- Usar un `TreeModelFilter` para utilizar un subconjunto de las filas de un `TreeStore` o `ListStore` como elementos de la lista de la `ComboBox`.
- Utilizar una función de datos de celda para modificar o sintetizar la visualización de los elementos de la lista.

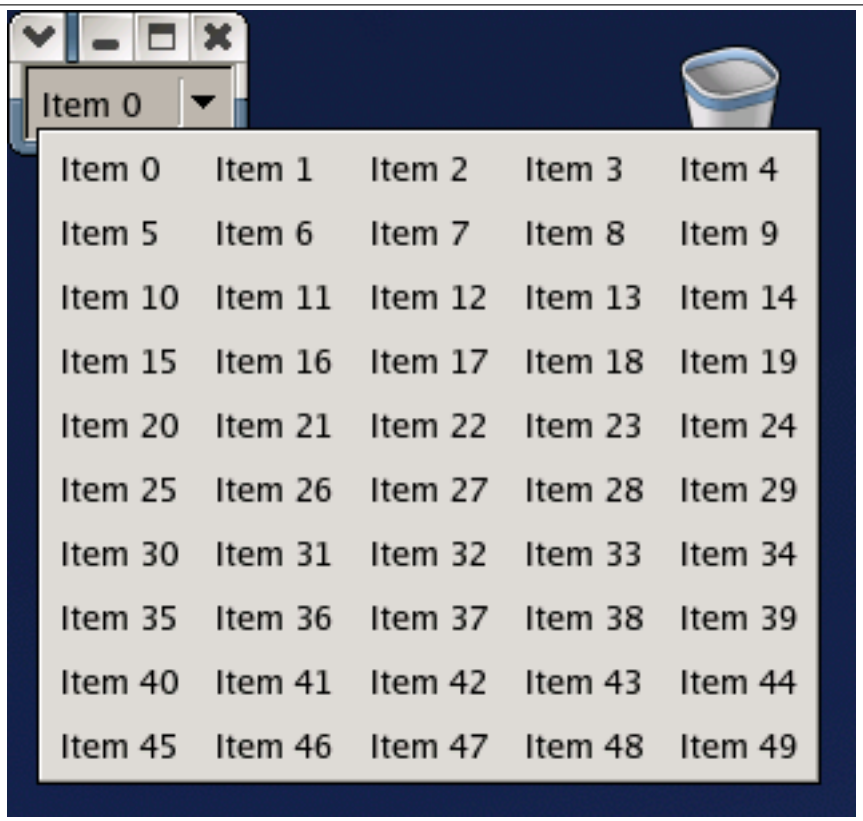
El uso de los objetos `TreeModel` y `CellRenderer` se detalla en el capítulo de Controles de Vista de Árbol.

Los elementos de la lista de la `ComboBox` se pueden mostrar en una tabla si se tiene un gran número de elementos que visualizar. En otro caso, la lista tendrá flechas de desplazamiento si la lista no puede ser mostrada en su totalidad. El siguiente método se usa para determinar el número de columnas que se mostrarán:

```
combobox.set_wrap_width(width)
```

donde *`width`* es el número de columnas de la tabla que muestra los elementos de la lista. Por ejemplo, el programa comboboxwrap.py muestra una lista de 50 elementos en 5 columnas. Figura 16.6 ilustra el programa en acción:

Figura 16.6 ComboBox con una Disposición Asociada



Con un gran número de elementos, digamos, por ejemplo, 50, el uso del método set_wrap_width() tendrá un mal rendimiento debido al cálculo de la disposición en tabla. Para tener una noción del efecto se puede modificar el programa comboboxwrap.py en su línea 18 de forma que muestre 150 elementos.

```
        for n in range(150):
```

Ejecute el programa para obtener una idea aproximada del tiempo de inicialización. Luego modifíquelo comentando la línea 17:

```
        #combobox.set_wrap_width(5)
```

Ejecute y cronometre de nuevo. Debería ejecutarse significativamente más rápido. Unas 20 veces más rápido.

Además del método get_active() antes descrito, se puede obtener un iterador `TreeIter` que señala la fila activa mediante el método:

```
  iter = combobox.get_active_iter()
```

También se puede establecer el elemento de la lista activa usando un iterador `TreeIter` con el método:

```
  combobox.set_active_iter(iter)
```

Los métodos set_row_span_column() y set_column_span_column() permiten la especificación de un número de columna de un `TreeModel` que contiene el número de filas o columnas que debe abarcar el elemento de la lista en una disposición de tabla. Desgraciadamente, en GTK+ 2.4 estos métodos funcionan mal.

Puesto que `ComboBox` implementa la interfaz `CellLayout`, que tiene capacidades similares a las de una `TreeViewColumn` (véase la sección de TreeViewColumn para más información). En resumen, la interfaz proporciona:

```
  combobox.pack_start(cell, expand=True)
  combobox.pack_end(cell, expand=True)
  combobox.clear()
```

Los dos primeros métodos empaquetan un `CellRenderer` en la `ComboBox` y el método clear() elimina todos los atributos de todos los `CellRenderers`.

Los siguientes métodos:

```
comboboxentry.add_attribute(cell, attribute, column)

comboboxentry.set_attributes(cell, ...)
```

establecen los atributos del `CellRenderer` indicado por *`cell`*. El método add_attribute() toma una cadena de nombre de atributo *`attribute`* (p.e. 'text') y un número entero de columna *`column`* de la columna en el `TreeModel` usado, para fijar el atributo *`attribute`*. Los argumentos restante del método set_attributes() son pares `atributo=columna` (p.e. text=1).

16.2.2. Controles ComboBoxEntry

El control `ComboBoxEntry` sustituye al control `Combo`. Es una subclase del control `ComboBox` y contiene un control hijo de entrada `Entry` que obtiene sus contenidos seleccionando un elemento en la lista desplegable o introduciendo texto directamente en la entrada bien desde el teclado o pegándolo desde un portapapeles `Clipboard` o una selección.

16.2.2.1. Uso Básico de ComboBoxEntry

Como la `ComboBox`, la `ComboBoxEntry` se puede crear con la función auxiliar:

```
comboboxentry = gtk.combo_box_entry_new_entry()
```

Una `ComboBoxEntry` se debe rellenar utilizando los métodos auxiliares de `ComboBox` descritos en Uso Básico de ComboBox.

Puesto que un control `ComboBoxEntry` es un control `Bin`, su control hijo de entrada `Entry` está disponible utilizando el atributo "child" o el método get_child():

```
entry = comboboxentry.child
entry = comboboxentry.get_child()
```

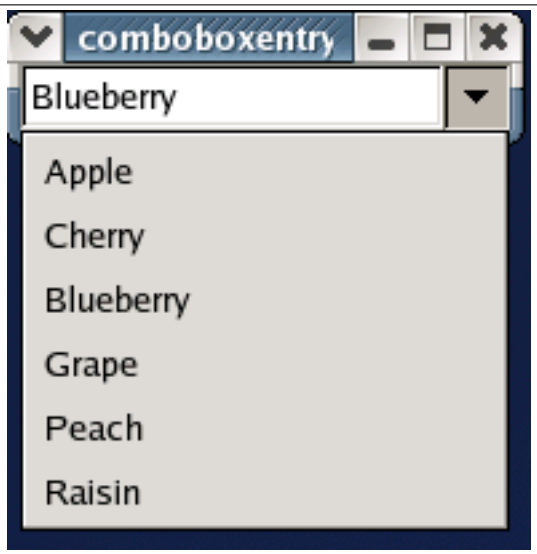
Se puede obtener el texto de la entrada `Entry` utilizando su método get_text().

Al igual que `ComboBox`, es posible seguir los cambios del elemento activo de la lista conectándose a la señal "changed". Desgraciadamente, esto no permite seguir los cambios en la entrada `Entry` que se hacen por entrada directa. Cuando se hace una entrada directa al control `Entry` se emite la señal "changed", pero el índice devuelto por el método get_active() será -1. Para seguir todos los cambios del texto de la entrada `Entry` text, será necesario utilizar la señal "changed" de la entrada `Entry`. Por ejemplo:

```
def changed_cb(entry):
    print entry.get_text()

comboboxentry.child.connect('changed', changed_cb)
```

imprimirá el texto tras cada cambio en el control hijo `Entry`. Por ejemplo, el programa comboboxentrybasic.py muestra el uso de la API auxiliar. Figura 16.7 ilustra la ejecución del programa:

Figura 16.7 ComboBoxEntry Básica



Obsérvese que cuando se modifica el texto de la entrada `Entry` debido a la selección de un elemento de la lista desplegable se llama dos veces al manejador de la señal "changed": una vez cuando se elimina el texto, y, otra cuando el texto se establece desde el elemento seleccionado de la lista.

16.2.2.2. Uso Avanzado de ComboBoxEntry

El constructor de una ComboBoxEntry es:

```
comboboxentry = gtk.ComboBoxEntry(model=None, column=-1)
```

donde *model* es un `TreeModel` y *column* es el número de la columna en el modelo *model* que se usará para fijar los elementos de la lista. Si no se indica la columna el valor predeterminado es -1 que significa que el texto de la columna no está especificado.

La creación de una `ComboBoxEntry` utilizando la función auxiliar `gtk.combo_box_entry_new_text()` es equivalente al siguiente código:

```
liststore = gtk.ListStore(str)
comboboxentry = gtk.ComboBoxEntry(liststore, 0)
```

La `ComboBoxEntry` añade un par de métodos que se usan para establecer y recuperar el número de columna del `TreeModel` que se usará para fijar las cadenas de los elementos de la lista:

```
comboboxentry.set_text_column(text_column)
text_column = comboboxentry.get_text_column()
```

La columna de texto también se puede obtener y especificar utilizando la propiedad "text-column". Véase la Sección de Uso Avanzado de ComboBox para más información sobre el uso avanzado de una `ComboBoxEntry`.

NOTA

La aplicación debe establecer la columna de texto para que la `ComboBoxEntry` fije los contenidos de la entrada `Entry` desde la lista desplegable. La columna de texto únicamente puede determinarse una vez, bien utilizando el constructor o utilizando el método set_text_column().

Al crear una `ComboBoxEntry` ésta se empaqueta con un nuevo `CellRendererText` que no es accesible. El atributo 'text' del `CellRendererText` se establece como un efecto colateral de la determinación de la columna de texto utilizando el método set_text_column(). Se pueden empaquetar `CellRenderers` adicionales en una `ComboBoxEntry` para la visualización en la lista desplegable. Véase la Sección de Uso Avanzado de ComboBox para más información.

16.3. Controles Botón de Color y de Fuente (ColorButton y FontButton)

16.3.1. Control Botón de Color (ColorButton)

El control `ColorButton` proporciona una forma cómoda de mostrar un color en un botón. Éste, al ser pulsado, abre un diálogo de selección de color (`ColorSelectionDialog`). Resulta útil para mostrar y establecer los colores en un diálogo de preferencias de usuario. Un botón `ColorButton` se encarga de configurar, mostrar y obtener el resultado del diálogo `ColorSelectionDialog`. El control `ColorButton` se crea con el constructor:

```
colorbutton = gtk.ColorButton(color=gtk.gdk.Color(0,0,0))
```

El color inicial se puede especificar con el parámetro *color*, aunque se puede determinar posteriormente con el método:

```
colorbutton.set_color(color)
```

El título del diálogo `ColorSelectionDialog` mostrado al pulsar el botón se puede determinar y obtener con los métodos:

```
colorbutton.set_title(title)

title = colorbutton.get_title()
```

La opacidad del color se determina utilizando el canal alpha. Los siguientes métodos obtienen y fijan la opacidad del color en un rango de 0 (transparente) a 65535 (opaco):

```
alpha = colorbutton.get_alpha()

colorbutton.set_alpha(alpha)
```

De forma predeterminada se ignora el valor alpha, dado que la propiedad "use_alpha" es `FALSE`. El valor de dicha propiedad "use_alpha" se puede alterar y obtener con los métodos:

```
colorbutton.set_use_alpha(use_alpha)

use_alpha = colorbutton.get_use_alpha()
```

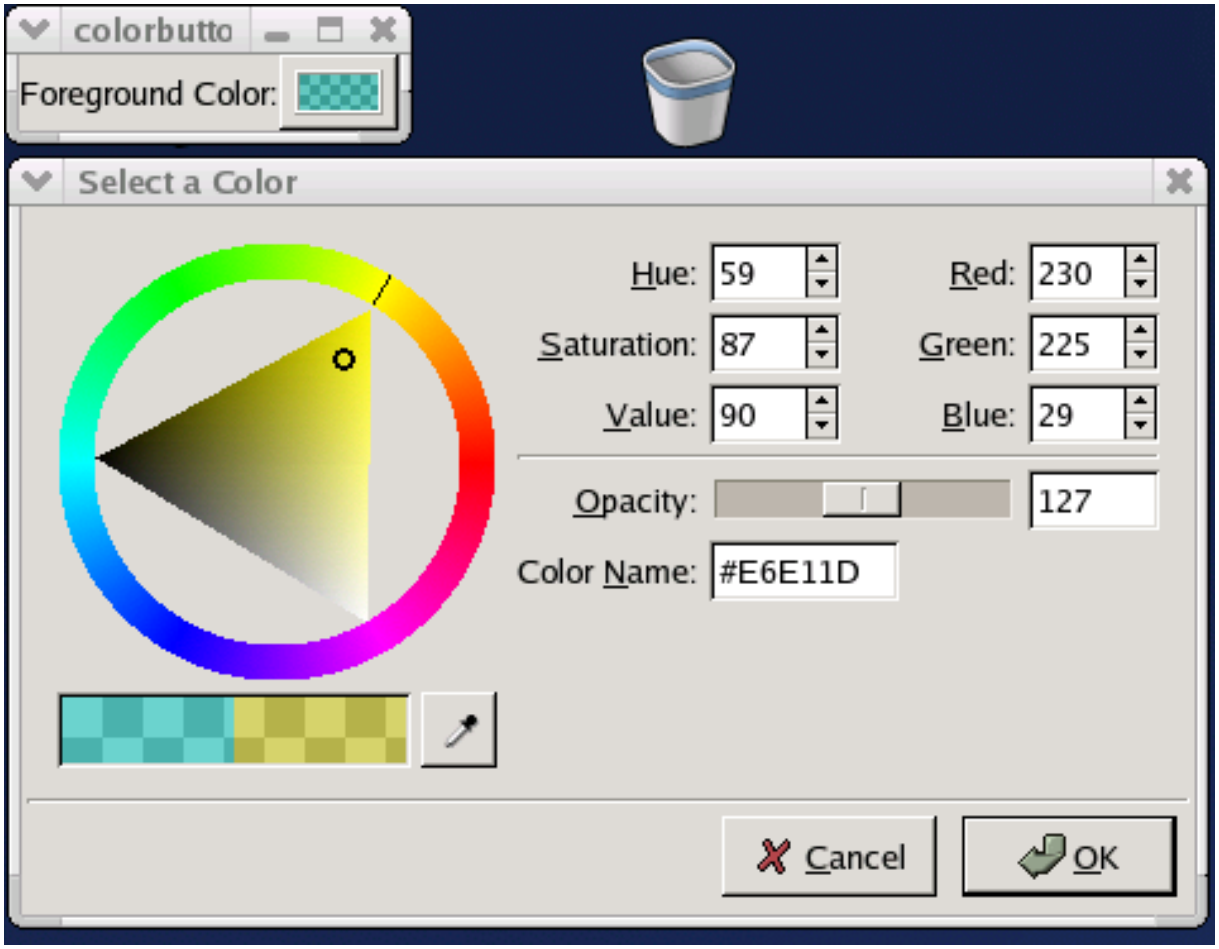
Si "use_alpha" es `TRUE` (verdadero), entonces el diálogo `ColorSelectionDialog` muestra un deslizador que modifica la opacidad y muestra el color sobre un fondo en damero.

Es posible detectar los cambios en el color seleccionado conectándose a la señal "color-set", que se emite cada vez que la usuaria modifica el color. La signatura de la retrollamada es:

```
def color_set_cb(colorbutton, user_data):
```

El programa de ejemplo colorbutton.py ilustra el uso del botón a `ColorButton`. Figura 16.8 muestra el programa en ejecución.

Figura 16.8 Ejemplo de Botón de Color - ColorButton



16.3.2. Control Botón de Fuente (FontButton)

Al igual que `ColorButton`, el control Botón de fuente `FontButton` es un control auxiliar que proporciona una visualización de la fuente actualmente seleccionada y, cuando se pulsa sobre él, abre un diálogo de selección de fuente `FontSelectionDialog`. El botón `FontButton` se encarga de configurar, mostrar y obtener el resultado de la llamada al diálogo `FontSelectionDialog`. Este control se crea con el constructor:

```
fontbutton = gtk.FontButton(fontname=None)
```

donde *`fontname`* es una cadena que especifica la fuente actual del diálogo `FontSelectionDialog`. Por ejemplo, el nombre de la fuente podría ser 'Sans 12', 'Sans Bold 14', o Monospace Italic 14'. Como mínimo es necesario indicar la familia y tamaño de la fuente.

La fuente actual también se puede modificar y obtener con los métodos:

```
result = fontbutton.set_font_name(fontname)

fontname = fontbutton.get_font_name()
```

donde *`result`* devuelve `TRUE` o `FALSE` para indicar si se pudo cambiar la fuente con éxito. El control `FontButton` posee una serie de métodos y propiedades asociadas que modifican la visualización de la fuente actual en el botón `FontButton`. Las propiedades "show-size" y "show-style" contienen valores booleanos que controlan si se muestra el tamaño y estilo de la fuente en la etiqueta del botón. Los métodos siguientes modifican y obtienen el valor de estas propiedades:

```
fontbutton.set_show_style(show_style)
show_style = fontbutton.get_show_style()

fontbutton.set_show_size(show_size)
```

```
show_size = fontbutton.get_show_size()
```

De forma alternativa, es posible usar el tamaño y estilo actuales en la etiqueta del botón para ilustrar inmediatamente la selección de fuente. Para ello tenemos las propiedades "use-size" y "use-font" y sus métodos asociados:

```
fontbutton.set_use_font(use_font)
use_font = fontbutton.get_use_font()

fontbutton.set_use_size(use_size)
use_size = fontbutton.get_use_size()
```

El uso de la fuente actual en la etiqueta resulta útil a pesar de los cambios inevitables que produce en el tamaño del botón, sin embargo, no ocurre lo mismo con el tamaño de la fuente, especialmente si se usan tamaños muy grandes o muy pequeños. Obsérvese además que, si se cambian las propiedades "use-font" o "use-size" a `TRUE` y posteriormente se vuelven a cambiar a `FALSE`, se retiene el último valor de fuente y tamaño visible. Por ejemplo, si "use-font" y "use-size" son `TRUE` y la fuente actual es `Monospace Italic 20`, entonces la etiqueta de `FontButton` se muestra usando la fuente `Monospace Italic 20`; si entonces cambiamos "use-font" y "use-size" a `FALSE` y la fuente actual a `Sans 12` la etiqueteta todavía mostrará la fuente `Monospace Italic 20`. Use el programa de ejemplo fontbutton.py para ver cómo funciona todo esto.

Finalmente, el título del diálogo de selección de fuente `FontSelectionDialog` se puede modificar y obtener con los métodos:

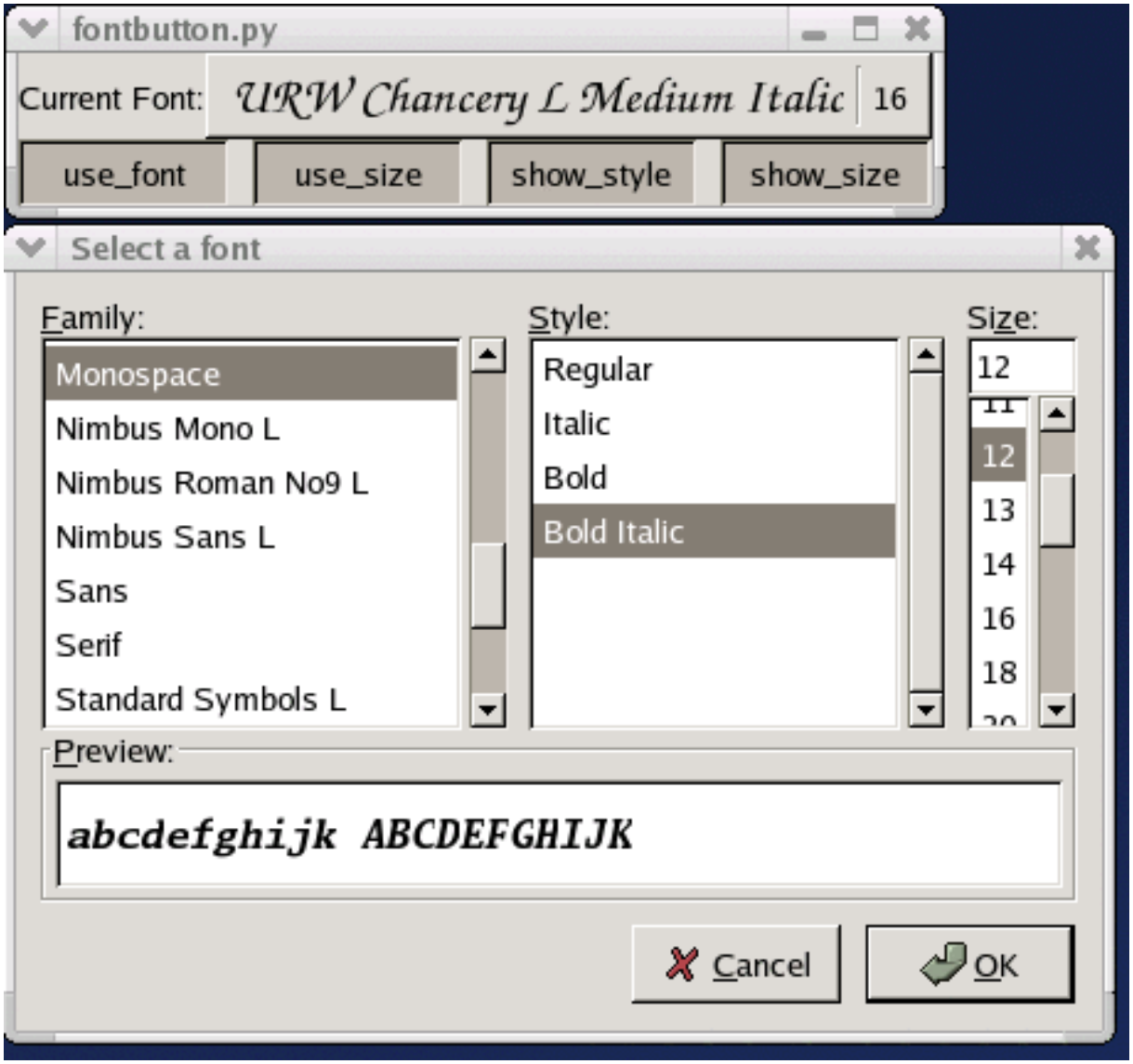
```
fontbutton.set_title(title)

title = fontbutton.get_title()
```

Al igual que un botón `ColorButton`, es posible seguir los cambios en la fuente actual conectándose a la señal "font-set" que se emite cuando la usuaria modifica la fuente. La signatura de la función de retrollamada es la que sigue:

```
def font_set_cb(fontbutton, user_data):
```

El programa de ejemplo fontbutton.py ilustra el uso del control `FontButton`. En él se pueden modificar las propiedades "use-font", "use-size", "show-size" y "show-style" mediante botones biestado. Figura 16.9 muestra el programa en ejecución.

Figura 16.9 Ejemplo de Botón de Fuente - FontButton



16.4. Controles de Entrada con Completado (EntryCompletion)

Un control `EntryCompletion` es un objeto que se usa con un control de entrada `Entry` para proporcionar la funcionalidad de completado. Cuando la usuaria escribe en la entrada `Entry`, el `EntryCompletion` mostrará una ventana con un conjunto de cadenas que coinciden con el texto parcial del `Entry`.

Un `EntryCompletion` se crea con el constructor:

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

Se puede usar el método `Entry`. set_completion() para asociar un `EntryCompletion` a una entrada `Entry`:

```
entry.set_completion(completion)
```

Las cadenas usadas por el `EntryCompletion` para buscar coincidencias se obtienen desde un `TreeModel` (generalmente un almacén de lista `ListStore`), que debe ser asignado usando el método:

```
completion.set_model(model)
```

El `EntryCompletion` implementa la interfaz `CellLayout`, al igual que la `TreeViewColumn`, para manejar la visulización de los datos del `TreeModel`. El siguiente método configura una `EntryCompletion` de la manera más habitual (una lista de cadenas):

```
completion.set_text_column(column)
```

Este método es equivalente a:

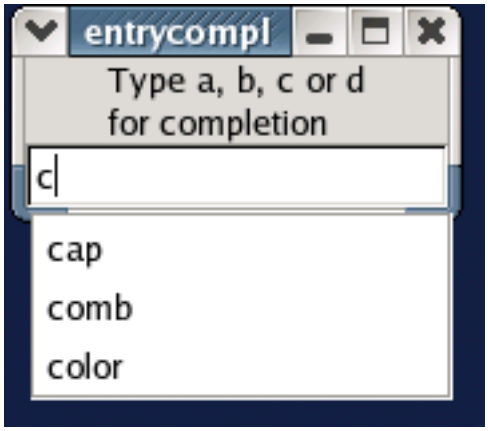
```
cell = CellRendererText()
completion.pack_start(cell)
completion.add_attribute(cell, 'text', column)
```

Para establecer el número de caracteres que deben ser introducidos antes de que `EntryCompletion` comience a mostrar coincidencias se puede usar el método:

```
completion.set_minimum_key_length(length)
```

El programa de ejemplo entrycompletion.py demuestra el uso de `EntryCompletion`. Figura 16.10 ilustra el programa en ejecución.

Figura 16.10 Entrada con Completado (EntryCompletion)



El programa comienza con un pequeño número de cadenas para el completado que puede ser aumentado escribiendo en el campo de entrada y presionando la tecla **Enter**. Si la cadena es única entonce se agrega a la lista de cadenas de completado.

La función de coincidencias preconstruida no diferencia entre mayúsculas y minúsculas. Si se necesita una función más especializada, se puede usar el siguiente método para instalar una función propia:

```
completion.set_match_func(func, user_data)
```

La signatura de *func* es:

```
def func(completion, key_string, iter, data):
```

donde *key_string* contiene el texto actual de la entrada `Entry`, *iter* es un iterador `TreeIter` que señala la fila del modelo `TreeModel`, y *data* son datos de usuario *user_data*. *func* debe devolver `TRUE` si la cadena de completado de la fila tiene que ser desplegada.

El fragmento de código mostrado a continuación usa una función de coincidencia para desplegar los nombres de completado que comienzan con el contenido de la entrada y tienen el sufijo dado, en este caso, un nombre terminado en `.png` para un archivo `PNG`.

```
...
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)
...
```

Por ejemplo, si el usuario teclea 'foo' y el modelo de completado contiene cadenas como 'foobar.png', 'smiley.png', 'foot.png' y 'foo.tif', las cadenas 'foobar.png' y 'foot.png' deberían mostrarse como alternativas.

16.5. Controles de Expansión (Expander)

El control `Expander` es un contenedor bastante simple que permite mostrar u ocultar su control hijo haciendo clic en un triángulo similar al de un `TreeView`. Se crean nuevos `Expander` con el constructor:

```
expander = gtk.Expander(label=None)
```

donde *`label`* es una cadena de texto utilizada como etiqueta del expansor. Si *`label`* es `None` o no se especifica, no se crea ninguna etiqueta. Alternativamente, se puede usar la función:

```
expander = gtk.expander_new_with_mnemonic(label=None)
```

que establece el carácter de la etiqueta precedido por un guión bajo como atajo de teclado mnemónico.

El control `Expander` usa la API de `Container` para añadir y eliminar su control hijo:

```
expander.add(widget)

expander.remove(widget)
```

El control hijo se puede obtener utilizando el atributo de `Bin` "child" o el método get_child().

La opción que controla la interpretación de los guiones bajos de la etiqueta se puede obtener y cambiar con los métodos:

```
use_underline = expander.get_use_underline()

expander.set_use_underline(use_underline)
```

Si se desea usar etiquetas de marcado de Pango (véase la Referencia de Marcas de Pango para más detalles) en la cadena de la etiqueta se usan los siguientes métodos para establecer y obtener el estado de la propiedad "use-markup":

```
expander.set_use_markup(use_markup)

use_markup = expander.get_use_markup()
```

Finalmente, se puede utilizar cualquier control como control de etiqueta utilizando el método siguiente:

```
expander.set_label_widget(label_widget)
```

Que permite, por ejemplo, poder utilizar una `HBox` empaquetada con una imagen y un texto de etiqueta.

Se puede obtener y establecer el estado del `Expander` utilizando los métodos:

```
expanded = expander.get_expanded()

expander.set_expanded(expanded)
```

Si *`expanded`* es `TRUE` entonces e muestra el control hijo.

En la mayoría de los casos `Expander` hace automáticamente lo que se desea, al revelar y ocultar el control hijo. En algunos casos la aplicación puede necesitar la creación de un control hijo en el momento de la expanción. Se puede usar la señal "notify::expanded" para seguir los cambios en el estado de triángulo expansor. El manejador de la señal puede entonces crear o modificar el control hijo según se necesite.

El programa de ejemplo expander.py muestra el uso de `Expander`. Figura 16.11 ilustra la ejecución del programa:

Figura 16.11 Control de Expansión

El programa crea una etiqueta `Label` que contiene la hora actual y la muestra cuando se expande el expansor.

16.6. Selecciones de Archivos mediante el uso de Controles basados en el Selector de Archivos FileChooser

El nuevo modo de seleccionar archivos en PyGTK 2.4 es el uso de variantes del control `FileChooser`. Los dos objetos que implementan esta nueva interfaz en PyGTK 2.4 son el control de Selección de Archivo `FileChooserWidget` y el diálogo de Selección de Archivo `FileChooserDialog`. Este último es el diálogo completo con la ventana y botones fácilmente definidos. El primero es un control útil para embeber en otro control.

Tanto el `FileChooserWidget` como el `FileChooserDialog` poseen los medios necesarios para navegar por el árbol del sistema de archivos y seleccionar ficheros. El aspecto de los controles depende de la acción utilizada para abrir el control.

Para crear un nuevo diálogo de selección de archivos y seleccionar un archivo existente (como en la opción de una aplicación típica Archivo → Abrir), se usa:

```
chooser = gtk.FileChooserDialog(title=None,action=gtk.FILE_CHOOSER_ACTION_OPEN,
                                buttons=(gtk.STOCK_CANCEL,gtk.RESPONSE_CANCEL, ↩
                                    gtk.STOCK_OPEN,gtk.RESPONSE_OK))
```

Para crear un nuevo diálogo de selección de archivos para seleccionar un nuevo nombre de archivo (como en la opción de una aplicación típica Archivo → Guardar), se usa:

```
chooser = gtk.FileChooserDialog(title=None,action=gtk.FILE_CHOOSER_ACTION_SAVE,
                                buttons=(gtk.STOCK_CANCEL,gtk.RESPONSE_CANCEL, ↩
                                    gtk.STOCK_OPEN,gtk.RESPONSE_OK))
```

En los ejemplos anteriores, los dos botones (los elementos de serie Cancelar y Abrir) se crean y conectan a sus respuestas respectivas (las respuestas estándar Cancelar y OK).

Para fijar la carpeta que se mostrará en el selector de archivos se usa el método:

```
chooser.set_current_folder(pathname)
```

Para establecer el nombre de archivo sugerido, tal como lo introduciría un usuario (la típica situación Archivo → Guardar como), se usa el método:

```
chooser.set_current_name(name)
```

El método anterior no necesita que exista el nombre de archivo. Si se quiere seleccionar un archivo existente concreto (tal como en la situación Archivo → Abrir), se debe usar el método:

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

Para obtener el nombre que ha seleccionado la usuaria o sobre la que ha hecho clic se usa el método:

```
filename = chooser.get_filename()
```

Es posible permitir selecciones múltiples (únicamente para la acción `gtk.FILE_CHOOSER_ACTION_OPEN`) utilizando el método:

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

donde *`select_mutiple`* debe ser `TRUE` para permitir selecciones múltiples. En este caso, se necesitará utilizar el método siguiente para obtener una lista de los nombres de archivo seleccionados:

```
filenames = chooser.get_filenames()
```

Una característica importante de todos los selectores de archivos es la capacidad de añadir filtros de selección de archivos. El filtro se añade con el método:

```
chooser.add_filter(filter)
```

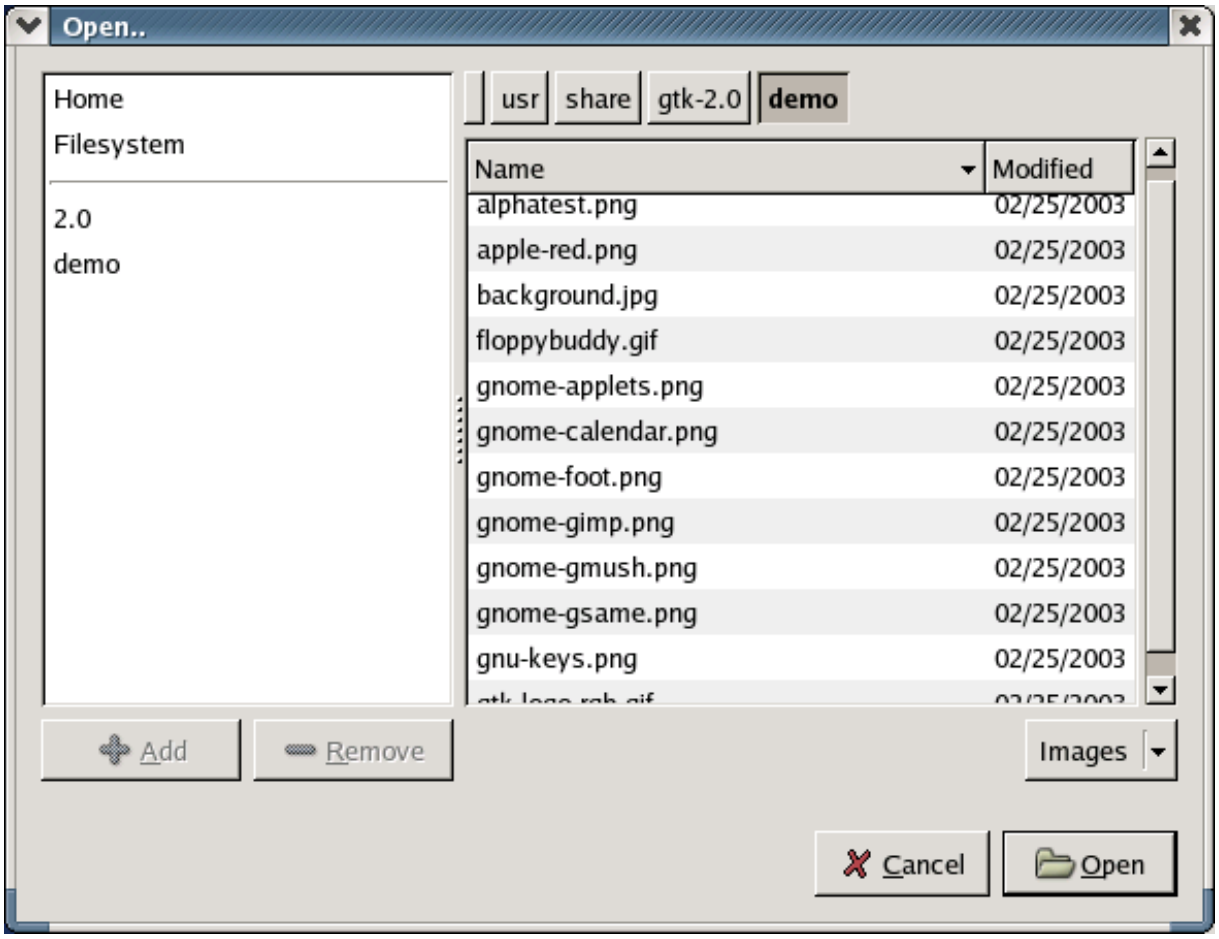
En el ejemplo anterior, *filter* debe ser una instancia de la clase `FileFilter`.

El panel izquierdo del selector de archivos da una lista de atajos a algunas carpetas tales como Inicio, Filesystem, CDROM, etc. Es posible añadir una carpeta a la lista de estos atajos y eliminarla de ella con los siguientes métodos:

```
chooser.add_shortcut_folder(folder)
chooser.remove_shortcut_folder(folder)
```

donde *folder* es la ruta de la carpeta. El programa de ejemplo filechooser.py ilustra el uso del control de selección de archivos. Figura 16.12 muestra el resultado de la ejecución:

Figura 16.12 Ejemplo de Selección de Archivos



El código fuente del programa de ejemplo filechooser.py es:

```
1  #!/usr/bin/env python
2
3  # ejemplo filechooser.py
4
5  import pygtk
6  pygtk.require('2.0')
7
8  import gtk
9
```

```
10   # Comprobamos la presencia del nuevo pygtk: esta es una clase nueva de ↩
    PyGtk 2.4
11   if gtk.pygtk_version < (2,3,90):
12      print "PyGtk 2.3.90 or later required for this example"
13      raise SystemExit
14
15   dialog = gtk.FileChooserDialog("Open..",
16                                  None,
17                                  gtk.FILE_CHOOSER_ACTION_OPEN,
18                                  (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
19                                   gtk.STOCK_OPEN, gtk.RESPONSE_OK))
20   dialog.set_default_response(gtk.RESPONSE_OK)
21
22   filter = gtk.FileFilter()
23   filter.set_name("All files")
24   filter.add_pattern("*")
25   dialog.add_filter(filter)
26
27   filter = gtk.FileFilter()
28   filter.set_name("Images")
29   filter.add_mime_type("image/png")
30   filter.add_mime_type("image/jpeg")
31   filter.add_mime_type("image/gif")
32   filter.add_pattern("*.png")
33   filter.add_pattern("*.jpg")
34   filter.add_pattern("*.gif")
35   filter.add_pattern("*.tif")
36   filter.add_pattern("*.xpm")
37   dialog.add_filter(filter)
38
39   response = dialog.run()
40   if response == gtk.RESPONSE_OK:
41       print dialog.get_filename(), 'selected'
42   elif response == gtk.RESPONSE_CANCEL:
43       print 'Closed, no files selected'
44   dialog.destroy()
```

16.7. El gestor de Interfaces de Usuario UIManager

16.7.1. Perspectiva general

`UIManager` proporciona una manera de crear menús y barras de herramientas a partir de una descripción similar a XML. `UIManager` usa objetos `ActionGroup` para gestionar los objetos `Action` que proporcionan la estructura de control de los elementos del menú y la barra de herramientas.

Al usar el gestor `UIManager` se pueden introducir y eliminar dinámicamente múltiples acciones y descripciones de la Interfaz de Usuario. Ello permite modificar los menús y las barras de herramientas con los cambios de modo de la aplicación (por ejemplo, si cambia de edición de texto a edición de imágenes), o en el caso de que se añadan o eliminen características accesorias de la aplicación.

Se puede usar `UIManager` para crear los menús y barras de herramientas de la interfaz de usuario de una aplicación así:

- Se crear una instancia de `UIManager`
- Se extrae el grupo de atajos `AccelGroup` del `UIManager` y se añade a la ventana de nivel superior
- Se crean las instancias de Grupo de Acciones (`ActionGroup`) y se rellenan pertinentemente con instancias de acciones `Action`.
- Se añaden las instancias de `ActionGroup` al `UIManager` en el orden en el que se desee que aparezcan las instancias de acciones (`Action`).

- Se añaden las descripciones XML de la interfaz al gestor `UIManager`. Es necesario asegurarse de que todas las acciones `Action` referenciadas por las descripciones XML están disponibles en las instancias de los grupos `ActionGroup` del `UIManager`.

- Se extraen, por nombre, las referencias a los controles de barra de menús, menú y barra de herramientas para utilizarlas en la construcción de la interfaz de usuario.

- Se modifica dinámicamente la interfaz de usuario añadiendo y eliminando las descripciones de Interfaz de Usuario y añadiendo, reordenando y eliminando las instancias asociadas de grupos de acciones (`ActionGroup`).

16.7.2. Creación de un gestor UIManager

Las instancias de `UIManager` se crean con el constructor:

```
uimamager = gtk.UIManager()
```

Los nuevos `UIManager` se crean con un grupo de atajos (`AccelGroup`) asociado, que puede obtenerse con el método:

```
accelgroup = uimanager.get_accel_group()
```

El grupo de atajos (`AccelGroup`) debería añadirse a la ventana de nivel superior de la aplicación para que las usuarias de la aplicación puedan usar los atajos de la acción (`Action`). Por ejemplo:

```
window = gtk.Window()
...
uimanager = gtk.UIManager()
accelgroup = uimanager.get_accel_group()
window.add_accel_group(accelgroup)
```

16.7.3. Adición y Eliminación de Grupos de Acciones (ActionGroups)

Tal como se describe en Sección 16.1.2, los grupos de acciones `ActionGroups` pueden llenarse con acciones (`Actions`) utilizando los métodos auxiliares add_actions(), add_toggle_actions() y add_radio_actions(). Los grupos de acciones (`ActionGroup`) se pueden usar desde un gestor `UIManager` una vez que han sido añadidos a su lista de grupo de acciones (`ActionGroup`) con el método:

```
uimanager.insert_action_group(action_group, pos)
```

donde *pos* es el índice de la posición en la que debería insertarse *action_group*. Un gestor de interfaces `UIManager` puede contener varios grupos de acciones (`ActionGroups`) con nombres de acciones (`Action`) repetidos. Por ello es importante el orden de los objetos `ActionGroup`, puesto que la búsqueda de una acción (`Action`) finaliza cuando se encuentra la primera acción (`Action`) con el nombre dado. Ello implica que las acciones que están en objetos `ActionGroup` que están situados antes ocultan a los que están colocados después.

Las acciones referenciadas en una descripción XML de la interfaz de usuario deben añadirse al `UIManager` antes que la propia descripción.

Se puede eliminar un grupo de acciones `ActionGroup` de un gestor de interfaz de usuario `UIManager` con el método:

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

Se puede obtener la lista de los objetos `ActionGroup` asociados con un determinado `UIManager` con el método:

```
actiongrouplist = uimanager.get_action_groups()
```

16.7.4. Descripciones de la Interfaz de Usuario

Las descripciones de la Interfaz de Usuario aceptadas por `UIManager` son simples definiciones XML con los siguientes elementos:

ui El elemento raíz de una descripción de una Interfaz de Usuario. Se puede omitir. Puede contener elementos **menubar**, **popup**, **toolbar** y **accelerator**.

menubar Elemento de nivel superior que describe una estructura de barra de menú (`MenuBar`) y puede contener elementos **MenuItem**, **separator**, **placeholder** y **menu**. Tiene un atributo opcional *name* (nombre) que, si se omite, toma el valor "menubar".

popup Elemento de nivel superior que describe una estructura de menú emergente (`Menu`) y que puede contener elementos **menuitem**, **separator**, **placeholder**, y **menu**. Tiene un atributo opcional *name* (nombre) que, si se omite, toma el valor "popup".

toolbar Elemento de nivel superior que describe una estructura de barra de herramientas (`Toolbar`) y que puede contener otros elementos **toolitem**, **separator** y **placeholder**. Posee un atributo opcional *name* (nombre) que, si se omite, toma el valor "toolbar".

placeholder Elemento que identifica una posición dentro de un **menubar**, **toolbar**, **popup** o **menu**. Este elemento puede contener otros elementos **menuitem**, **separator**, **placeholder**, y **menu**. Los elementos **Placeholder** se usan al incorporar descripciones de Interfaz de Usuario para permitir, por ejemplo, la construcción de un menú a partir de descripciones de Interfaz de usuario utilizando nombres de contenedores **placeholder** compartidos. Posee un atributo opcional *name* (nombre), que, si se omite, toma el valor "placeholder".

menu Elemento que describe una estructura de menú `Menu` y puede contener otros elementos **menuitem**, **separator**, **placeholder** y **menu**. Un elemento **menu** tiene un atributo obligatorio *action* que denomina un objeto de acción `Action` que se usará en la creación del `Menu`. Opcionalmente puede incluir los atributos *name* (nombre) y *position* (posición). Si no se especifica *name* se usa el valor correspondiente al elemento *action* como nombre. El atributo *position* puede tomar el valor "top" (superior) o "bottom" (inferior), usándose éste último si no se especifica *position*.

menuitem Elemento que escribe un elemento de menú `MenuItem`. El elemento **menuitem** posee un atributo obligatorio *action* que denomina un objeto de acción `Action` que se usa para crear el elemento de menú `MenuItem`. También posee los atributos optativos *name* (nombre) y *position* (posición). Si no se indica *name* entonces se usa el nombre correspondiente al elemento *action*. El atributo *position* puede tomar el valor "top" (superior) o el valor "bottom" (inferior), usándose éste último si no se especifica *position*.

toolitem Elemento que describe un elemento de barra de herramientas `ToolItem`. El elemento **toolitem** posee un atributo obligatorio *action* que denomina un objeto de acción `Action` que se usa para crear la barra de herramientas `Toolbar`. También posse los atributos optativos *name* (nombre) y *position* (posición). Si no se indica *name* entonces se usa el nombre correspondiente al elemento *action*. El atributo *position* puede tomar el valor "top" (superior) o el valor "bottom" (inferior), usándose éste último si no se especifica *position*.

separator Elemento que describe un separador de elemento de menú `SeparatorMenuItem` o un separador de elemento de barra de herramientas `SeparatorToolItem` según corresponda.

accelerator Elemento que describe un atajo de teclado (acelerador). El elemento **accelerator** posee un atributo obligatorio *action* que denomina un objeto de acción `Action` que define la combinación de teclas del atajo y que se activa con el atajo. También tiene un atributo opcional de nombre *name*. Si no se especifica *name* se usa el nombre de *action* como nombre.

Como ejemplo, una descripción de Interfaz de Usuario que se podría usar para la creación de una interfaz similar a la de Figura 16.4 es:

```
<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>
```

Obsérvese que esta descripción simplemente usa los nombres de los elementos **action** como nombres de la mayoría de los elementos, en lugar de especificar sus atributos *name*. También es desaconsejable el uso del elemento **ui** puesto que parece innecesario.

La jerarquía de objetos que se crea al usar una descripción de Interfaz de Usuario es muy parecida a la jerarquía de elementos XML exceptuando que los elementos contenedores **placeholder** se introducen en sus elementos padre.

Se puede acceder a un control de la jerarquía creada por una descripción de Interfaz de Usuario mediante su ruta, que se compone del nombre del elemento de control y sus elementos anteriores separados por barras inclinadas ("/"). Por ejemplo, si se usa la descripción anterior, estas serían rutas válidas de algunos controles:

```
/MenuBar
/MenuBar/File/Quit
/MenuBar/RadioBand/SSB
/Toolbar/Mute
/Toolbar/RadioBandItems/FM
```

Hay que observar que el nombre de los contenedores **placeholder** deben incluirse en la ruta. Generalmente simplemente se accederá a los controles de nivel superior (por ejemplo, "/MenuBar" y "/Toolbar") pero es posible que sea necesario acceder a otros controles de nivel inferior para, por ejemplo, cambiar una propiedad.

16.7.5. Adición y Eliminación de Descripciones de Interfaz de Usuario

Una vez que se configura un gestor de Interfaz de Usuario `UIManager` con un grupo de acciones `ActionGroup` entonces es posible añadir una descripción de Interfaz de Usuario e integrarla con la interfaz existente mediante los siguientes métodos:

```
merge_id = uimanager.add_ui_from_string(buffer)

merge_id = uimanager.add_ui_from_file(filename)
```

donde *`buffer`* es una cadena que contiene una descripción de Interfaz de Usuario y *`filename`* es el archivo que contiene una descripción de Interfaz de Usuario. Ambos métodos devuelven un identificador *`merge_id`* que consiste en un valor entero único. Si falla el método se emite la excepción `GError`. El identificador *`merge_id`* puede usarse para eliminar la descripción de Interfaz de Usuario del gestor `UIManager` con el método:

```
uimanager.remove_ui(merge_id)
```

Los mismos métodos se pueden usar más de una vez para añadir descripciones adicionales, que se incorporarán para obtener una descripción XML de Interfaz de Usuario combinada. Se hablará más de las Interfaces de Usuario combinadas de forma más detallada en la sección Sección 16.7.8.

Se puede añadir un elemento sencillo de Interfaz de Usuario a la descripción existente con el método:

```
uimanager.add_ui(merge_id, path, name, action, type, top)
```

donde *merge_id* es un valor entero único, *path* es la ruta donde se añadirá el nuevo elemento, *action* es el nombre de una acción `Action` o `None` para añadir un elemento **separator**, *type* es el tipo de elemento que se ha de añadir y *top* (superior) es un valor booleano. Si *top* es `TRUE` el elemento se añadirá antes que sus elementos hermanos, y después en caso contrario.

merge_id se obtendría con el método:

```
merge_id = uimanager.new_merge_id()
```

Los valores enteros devueltos por el método new_merge_id() son monótonamente crecientes.

path (ruta) es una cadena compuesta por el nombre del elemento y los nombres de sus ancestros unidos por una barra inclinada ("/") pero sin incluir el nudo optativo raíz "/ui". Por ejemplo, "/MenuBar/RadioBand" es la ruta del elemento **menu** llamado "RadioBand" de la siguiente descripción de Interfaz de Usuario:

```
<menubar name="MenuBar">
  <menu action="RadioBand">
  </menu>
</menubar>
```

El valor de *type* (tipo) debe ser uno de los siguientes:

gtk.UI_MANAGER_AUTO El tipo del elemento de Interfaz de Usuario (menuitem, toolitem o separator) se establece en función del contexto.

gtk.UI_MANAGER_MENUBAR Una barra de menú.

gtk.UI_MANAGER_MENU Un menú.

gtk.UI_MANAGER_TOOLBAR Una barra de herramientas.

gtk.UI_MANAGER_PLACEHOLDER Un contenedor (placeholder).

gtk.UI_MANAGER_POPUP Un menú emergente.

gtk.UI_MANAGER_MENUITEM Un elemento de menú.

gtk.UI_MANAGER_TOOLITEM Un elemento de barra de herramientas.

gtk.UI_MANAGER_SEPARATOR Un separador.

gtk.UI_MANAGER_ACCELERATOR Un atajo o acelerador.

add_ui() falla sin aviso si el elemento no es añadido. El uso de add_ui() es de un nivel tan bajo que se deberían usar siempre en su lugar los métodos auxiliares add_ui_from_string() y add_ui_from_file().

La adición de un elemento o una descripción de Interfaz de Usuario provoca la actualización de la jerarquía de controles en una función ociosa (idle). Se puede asegurar que la jerarquía de controles se ha actualizado antes de acceder a ella utilizando el método:

```
uimanager.ensure_update()
```

16.7.6. Acceso a los Controles de la Interfaz de Usuario

Se accede a un control de la jerarquía de controles de la Interfaz de Usuario mediante el método:

```
widget = uimanager.get_widget(path)
```

donde *path* (ruta) es una cadena que contiene el nombre del elemento control y sus ancestros de la forma que se describe en Sección 16.7.4.

Por ejemplo, dada la siguiente descripción de Interfaz de Usuario:

```
<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>
```

que haya sido añadida al gestor `UIManager` *`uimanager`*, es posible acceder a la barra de menús (`MenuBar`) y barra de herramientas (`Toolbar`) para su uso en una ventana de aplicación (`Window`) utilizando el código que sigue:

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

De la misma forma, se accede a los controles de los niveles inferiores mediante sus rutas. Por ejemplo, se accede al elemento de la clase `RadioToolButton` llamado "SSB" así:

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

Para facilitar las cosas, se pueden obtener todos los controles de nivel superior de un determinado tipo mediante el método:

```
toplevels = uimanager.get_toplevels(type)
```

donde *`type`* especifica el tipo de los controles que se devolverán usando una combinación de las banderas: `gtk.UI_MANAGER_MENUBAR`, `gtk.UI_MANAGER_TOOLBAR` y `gtk.UI_MANAGER_POPUP`. Se puede usar el método gtk.Widget.get_name() para determinar qué control de nivel superior se tiene.

Se puede obtener la acción `Action` que usa el control auxiliar asociado con un elemento de Interfaz de Usuario usando el método:

```
action = uimanager_get_action(path)
```

donde *`path`* (ruta) es una cadena que contiene la ruta a un elemento de la Interfaz de Usuario de *`uimanager`*. Si el elemento no posee una acción `Action` asociada entonces se devuelve `None`.

16.7.7. Ejemplo sencillo de Gestor de Interfaz UIManager

uimanager.py es un ejemplo sencillo de programa que ilustra el uso de un gestor de interfaz de usuario `UIManager`. Figura 16.13 muestra el programa en funcionamiento.

Figura 16.13 Programa sencillo de Gestor de Interfaz de Usuario UIManager

El programa de ejemplo uimanager.py usa la descripción XML de Sección 16.7.6. El texto de las dos etiquetas se cambian como respuesta a la activación de la acción biestado (`ToggleAction`) "Mute" y de las acciones de exclusión mútua (`RadioAction`) "AM", "FM" and "SSB". Todas las acciones están contenidas en un único grupo de acciones (`ActionGroup`) que permite que se pueda conmutar la sensibilidad y visibilidad de todos los controles auxiliares de las acciones mediante los botones biestado "Sensitive" y "Visible". El uso del elemento contenedor **placeholder** se describe posteriormente en Sección 16.7.8.

16.7.8. Combinación de Descripciones de Interfaz de Usuario

La combinación de descripciones de Interfaz de Usuario se hace en función del nombre de los elementos XML. Tal como se indicó más arriba, los elementos individuales de la jerarquía pueden ser accedidos usando un nombre de ruta que está formado por el nombre del elemento y los nombres de sus ancestros. Por ejemplo, si usamos la descripción de Interfaz de Usuario de Sección 16.7.4, el elemento **toolitem** "AM" tiene la ruta "/Toolbar/RadioBandItems/AM" mientras que el elemento **menuitem** "FM" tiene la ruta "/MenuBar/RadioBand/FM".

Si se combina una descripción de Interfaz de Usuario con esa descripción de Interfaz entonces sus elementos se añaden como elementos del mismo nivel que los elementos existentes. Por ejemplo, si la descripción de Interfaz de Usuario:

```
<menubar name="MenuBar">
  <menu action="File">
    <menuitem action="Save" position="top"/>
    <menuitem action="New" position="top"/>
  </menu>
  <menu action="Sound">
    <menuitem action="Loudness"/>
  </menu>
  <menu action="RadioBand">
    <menuitem action="CB"/>
    <menuitem action="Shortwave"/>
  </menu>
</menubar>
<toolbar name="Toolbar">
  <toolitem action="Save" position="top"/>
  <toolitem action="New" position="top"/>
  <separator/>
  <toolitem action="Loudness"/>
  <separator/>
  <placeholder name="RadioBandItems">
    <toolitem action="CB"/>
    <toolitem action="Shortwave"/>
  </placeholder>
</toolbar>
```

se añade a nuestra descripción de Interfaz de Usuario de ejemplo:

```
<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>
```

se crearía la siguiente descripción de Interfaz de Usuario combinada:

```
<menubar name="MenuBar">
  <menu name="File" action="File">
    <menuitem name="New" action="New"/>
    <menuitem name="Save" action="Save"/>
    <menuitem name="Quit" action="Quit"/>
  </menu>
  <menu name="Sound" action="Sound">
    <menuitem name="Mute" action="Mute"/>
    <menuitem name="Loudness" action="Loudness"/>
  </menu>
  <menu name="RadioBand" action="RadioBand">
    <menuitem name="AM" action="AM"/>
    <menuitem name="FM" action="FM"/>
    <menuitem name="SSB" action="SSB"/>
    <menuitem name="CB" action="CB"/>
    <menuitem name="Shortwave" action="Shortwave"/>
  </menu>
</menubar>
<toolbar name="Toolbar">
  <toolitem name="New" action="New"/>
  <toolitem name="Save" action="Save"/>
  <toolitem name="Quit" action="Quit"/>
  <separator/>
  <toolitem name="Mute" action="Mute"/>
  <separator name="sep1"/>
  <placeholder name="RadioBandItems">
    <toolitem name="AM" action="AM"/>
    <toolitem name="FM" action="FM"/>
    <toolitem name="SSB" action="SSB"/>
    <toolitem name="CB" action="CB"/>
    <toolitem name="Shortwave" action="Shortwave"/>
  </placeholder>
  <separator/>
  <toolitem name="Loudness" action="Loudness"/>
  <separator/>
</toolbar>
```

Si se examina el XML resultante se puede ver que los elemnentos **menuitem** "New" y "Save" se han combinado antes que el elemento "Quit" como resultado de haber fijado el atributo "position" a "top", que significa que el elemento debe ser antepuesto. De la misma forma, los elementos **toolitem** "New" y "Save" se han antepuesto a "Toolbar". Obsérvese que los elementos "New" y "Save" son invertidos en el proceso de combinación.

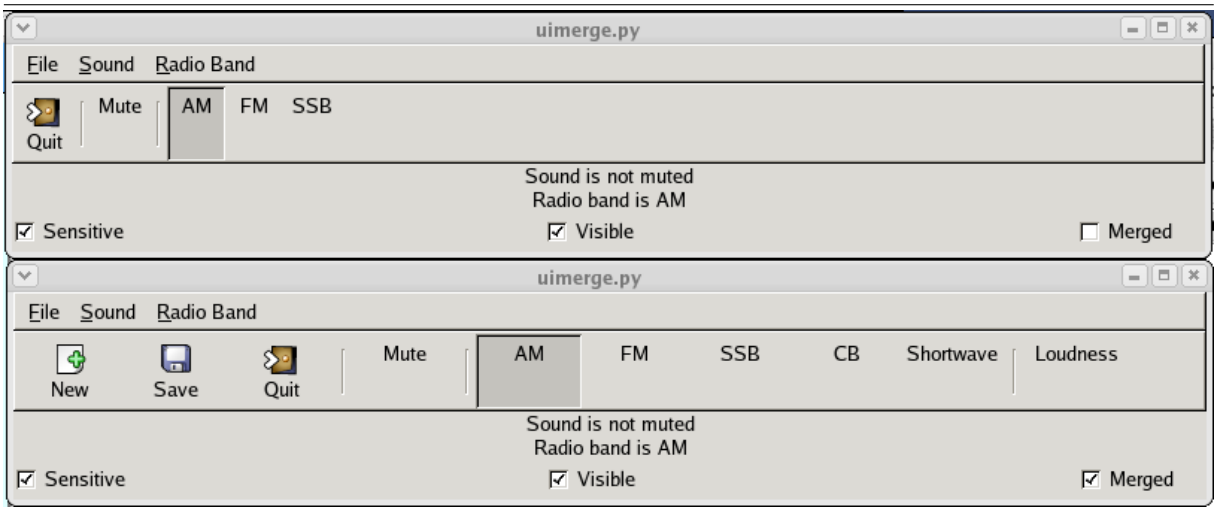
El elemento **toolitem** "Loudness" se añade tras los elementos "Toolbar" y aparece el último en la descripción combinada de la Interfaz de Usuario, aunque no sea así en su propia descripción de Interfaz de Usuario. El elemento **placeholder** "RadioBandItems" en ambas Interfaces de Usuario combina los elementos **toolitem** "CB" y "Shortwave" con los elementos "AM", "FM", and "SSB". Si no se hubiese usado el elemento **placeholder** "RadioBandItems" entonces los elementos "CB" y "Shortwave" se habrían situado tras el elemento "Loudness".

Se puede obtener una representación de la descripción de Interfaz de Usuario utilizada por un gestor `UIManager` con el método:

```
uidesc = uimanager.get_ui()
```

El programa de ejemplo uimerge.py muestra la combinación de las descripciones de Interfaz de Usuario anteriores. Figura 16.14 ilustra la Interfaz original y la combinada:

Figura 16.14 Ejemplo UIMerge



El programa de ejemplo usa tres objetos `ActionGroup`:

- Objetos `Action` para los menús "File", "Sound" y "Radio Band"
- Objetos `Action` para los menús "Quit", "Mute", "AM", "FM", "SSB" y "Radio Band"
- Objetos `Action` para los elementos "Loudness", "CB" y "Shortwave"

Los controles de botón biestado (`ToggleButton`) "Sensitive" y Visible" controlan la sensibilidad y visibilidad de únicamente el segundo grupo de acciones (`ActionGroup`).

16.7.9. Señales de UIManager

`UIManager` posee una par de señales interesantes a las que se puede conectar una aplicación. La señal "actions-changed" se emite cuando se añade o elimina un grupo de acciones `ActionGroup` de un gestor `UIManager`. La signatura de la retrollamada es:

```
def callback(uimanager, ...)
```

La señal "add-widget" se emite cuando se crea un control auxiliar `MenuBar` o `Toolbar`. La signatura de la retrollamada es:

```
def callback(uimanager, widget, ...)
```

donde *`widget`* es el control recién creado.

Capítulo 17

Controles sin documentar

¡Todos estos controles necesitan voluntarios! :) Por favor, considere contribuir a mejorar este tutorial.

Si se necesita usar alguno de estos controles que están sin documentar, es muy recomendable echar un vistazo a los archivos *.c de la distribución de PyGTK. Los nombres de los métodos de PyGTK son muy descriptivos y, una vez que se comprende cómo funcionan las cosas, no es muy difícil averiguar como usar un control simplemente echando un vistazo a la definición de sus métodos. Esto, junto con la lectura de algunos ejemplos del código de otros, debería bastar para no encontrar mayor problema.

En cuanto realmente se entiendan todos los métodos de un control nuevo, que todavía se encuentra sin documentar, por favor, piense en escribir un fragmento de tutorial sobre él para que otros puedan beneficiarse del tiempo invertido.

17.1. Etiqueta de Aceleración (Atajo)

17.2. Menú de Opciones

17.3. Elementos de Menú

17.3.1. Elemento de Menú de Activación

17.3.2. Elemento de Menú de Exclusión Mútua

17.3.3. Elemento de Menú de Separación

17.3.4. Elemento de Menú de Cascada

17.4. Curvas

17.5. Diálogo de Mensaje

17.6. Curva Gamma

Capítulo 18

Establecimiento de Atributos de Controles

Este capítulo describe los métodos usados para modificar controles (y objetos) y poder cambiar su estilo, espaciamiento, tamaño, etc.

El método:

```
widget.activate()
```

hace que el control emita la señal "activate".

El método:

```
widget.set_sensitive(sensitive)
```

cambia la sensibilidad del control (es decir, si reacciona a eventos). Si *sensitive* es TRUE (verdadero) el control recibirá eventos; si es FALSE (falso) el control no recibirá eventos. Un control que está insensible se visualiza normalmente en un tono gris.

El método:

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

establece el tamaño del control de forma que tenga el ancho dado por el parámetro *width* y la altura dada por el parámetro *height*.

18.1. Métodos de Banderas de los Controles

Los métodos:

```
widget.set_flags(flags)

widget.unset_flags(flags)

flags = widget.flags()
```

ponen, quitan y leen las banderas de los objetos gtk.Object y gtk.Widget . Las *flags* (banderas) pueden ser cualquiera de las banderas estándar:

```
IN_DESTRUCTION    # en destrucción
FLOATING          # flotannte
RESERVED_1        # reservada 1
RESERVED_2        # reservada 2
TOPLEVEL          # de nivel superior
NO_WINDOW         # sin ventana
REALIZED          # realizado
MAPPED            # mapeado
VISIBLE           # visible
SENSITIVE         # sensible
PARENT_SENSITIVE  # padre sensible
CAN_FOCUS         # puede recibir el foco
```

```
HAS_FOCUS         # tiene el foco
CAN_DEFAULT       # puede ser el control predeterminado
HAS_DEFAULT       # es el control predeterminado
HAS_GRAB          # tiene la exclusividad de los eventos
RC_STYLE          # estilo rc
COMPOSITE_CHILD   # hijo compuesto
NO_REPARENT       # no reparentado
APP_PAINTABLE     # aplicación pintable
RECEIVES_DEFAULT  # recibe predeterminado
DOUBLE_BUFFERED   # tiene doble buffer
```

El método:

```
widget.grab_focus()
```

permite a un control adquirir el foco en caso de que tenga la bandera `CAN_FOCUS` activada.

18.2. Métodos de Visualización de Controles

Los métodos:

```
widget.show()

widget.show_all()

widget.hide()

widget.hide_all()

widget.realize()

widget.unrealize()

widget.map()

widget.unmap()
```

controlan la visualización del control *`widget`*.

El método show() (mostrar) hace que el control se visualice llamando los métodos realize() y map().

El método hide() (ocultar) hace que el control se oculte y, si es necesario, también lo "desmapea" usando el método unmap().

Los métodos show_all() (mostrar todos) y hide_all() (ocultar todos) hacen que el control y todos sus controles hijos se muestren o se oculten.

El método realize() ("realizar") hace que se reserven los recursos que necesita el control, incluida su ventana.

El método unrealize() ("desrealizar") libera la ventana del control y otros recursos asociados al mismo. "Desrealizar" un control también lo oculta y "desmapea".

El método map() ("mapear") hace que se reserve espacio en el display (pantalla) para el control; ésto sólo se aplica a los controles que necesitan ser manipulados por el gestor de ventanas. Mapear un control también lo realiza si es necesario.

El método unmap() (desmapear) elimina un control del display (pantalla) y también lo oculta si es necesario.

18.3. Atajos de Teclado de los Controles

Los siguientes métodos:

```
widget.add_accelerator(accel_signal, accel_group, accel_key, accel_mods,  ↩
    accel_flags)

widget.remove_accelerator(accel_group, accel_key, accel_mods)
```

añaden y eliminan atajos de teclado a un Grupo de Atajos de Teclado (`gtk.AcceleratorGroup`) que, asociado al control de nivel más alto de un conjunto de ellos, hace que se gestionen los atajos de teclado de éstos.

El parámetro *accel_signal* es una señal que puede emitir el control .

El parámetro *accel_key* es la tecla que se usará como atajo de teclado.

El parámetro *accel_mods* es un grupo de modificadores que se añaden a la tecla *accel_key* (por ejemplo **Shift** (mayúsculas), **Control**, etc.):

```
SHIFT_MASK
LOCK_MASK
CONTROL_MASK
MOD1_MASK
MOD2_MASK
MOD3_MASK
MOD4_MASK
MOD5_MASK
BUTTON1_MASK
BUTTON2_MASK
BUTTON3_MASK
BUTTON4_MASK
BUTTON5_MASK
RELEASE_MASK
```

El parámetro *accel_flags* es un conjunto de opciones sobre cómo se muestra la información del atajo de teclado. Los valores posibles son:

```
ACCEL_VISIBLE          # mostrar la tecla de aceleración en el control

ACCEL_LOCKED           # no permitir que la ventana del atajo de teclado cambie
```

Un grupo de atajos de teclado se crea con la función:

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

El grupo *accel_group* se añade a una ventana de nivel superior con el siguiente método:

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

Ejemplo de cómo añadir un atajo de teclado:

```
menu_item.add_accelerator("activate", accel_group,
                          ord('Q'), gtk.gdk.CONTROL_MASK, gtk.ACCEL_VISIBLE)
```

18.4. Métodos relacionados con el Nombre de los Controles

Los siguientes métodos cambian y leen el nombre de un control:

```
widget.set_name(name)

name = widget.get_name()
```

El parámetro *name* es la cadena de caracteres que se asocia al control *widget*. Es útil para definir estilos usados en controles particulares de una aplicación. El nombre del control puede usarse para limitar la aplicación del estilo en vez de usar la clase del control. En el capítulo cómo usar los ficheros rc de GTK+ se profundiza más en el uso de estilos de esta manera.

18.5. Estilo de los Controles

Los siguientes métodos cambian y leen el estilo asociado a un control:

```
widget.set_style(style)

style = widget.get_style()
```

La siguiente función:

```
style = get_default_style()
```

obtiene el estilo predeterminado.

Un estilo contiene la información gráfica que necesita un control para dibujarse a sí mismo en sus diferentes estados:

```
STATE_NORMAL        # El estado durante la operación normal
STATE_ACTIVE        # El control está activado, como cuando se pulsa un botón
STATE_PRELIGHT      # El puntero del ratón está sobre el control
STATE_SELECTED      # El control está seleccionado
STATE_INSENSITIVE   # El control está desactivado
```

Un estilo contiene los siguientes atributos:

```
fg          # una lista de 5 colores de primer plano, uno para cada estado
bg          # una lista de 5 colores de fondo
light       # una lista de 5 colores (claros), creados en el método set_style()
dark        # una lista de 5 colores (oscuros), creados en el método set_style ↩
    ()
mid         # una lista de 5 colores (medios), creados en el método set_style()
text        # una lista de 5 colores para texto
base        # una lista de 5 colores para bases
text_aa     # una lista de 5 colores a medio camino entre text/base

black       # el color negro
white       # el color blanco
font_desc   # la descripción de fuente pango predeterminada

xthickness  #
ythickness  #

fg_gc       # una lista de 5 contextos gráficos (primer plano) - creados en el  ↩
    método set_style()
bg_gc       # una lista de 5 contextos gráficos (fondo) - creados en el método  ↩
    set_style()
light_gc    # una lista de 5 contextos gráficos (claros) - creados en el método ↩
     set_style()
dark_gc     # una lista de 5 contextos gráficos (oscuros) - creados en el  ↩
    método set_style()
mid_gc      # una lista de 5 contextos gráficos (medios) - creados en el método ↩
     set_style()
text_gc     # una lista de 5 contextos gráficos (texto) - creados en el método  ↩
    set_style()
base_gc     # una lista de 5 contextos gráficos (base) - creados en el método  ↩
    set_style()
black_gc    # una lista de 5 contextos gráficos (negro) - creados en el método  ↩
    set_style()
white_gc    # una lista de 5 contextos gráficos (blanco) - creados en el método ↩
     set_style()

bg_pixmap   # una lista de 5 pixmaps (de fondo)
```

Cada atributo se puede leer directamente de manera similar a *style.black* y `style.fg_gc[gtk.STATE_NORMAL]`. Todos los atributos son de sólo lectura excepto `style.black`, `style.white`, `style.black_gc` y `style.white_gc`.

Se puede copiar un estilo existente para su modificación posterior con el método:

```
new_style = style.copy()
```

que copia los atributos del estilo salvo las listas de contextos gráficos y las listas de colores light, dark y mid (claros, oscuros y medios).

El estilo actual se puede obtener con:

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

Para cambiar el estilo de un control (por ejemplo, para cambiar el color de primer plano), se deben usar los siguientes métodos de un control:

```
widget.modify_fg(state, color) # parámetros: estado, color
widget.modify_bg(state, color) # parámetros: estado, color
widget.modify_text(state, color) # parámetros: estado, color
widget.modify_base(state, color) # parámetros: estado, color
widget.modify_font(font_desc) # parámetro: descripción de la fuente
widget.set_style(style) # parámetro: estilo
```

Cuando se establece el estilo *style* se reservan los colores del estilo y se crean los contextos gráficos. La mayoría de los controles se redibujan también automáticamente tras cambiar de estilo. Si el estilo *style* es None entonces el control regresará al estilo predeterminado.

No todos los cambios del estilo afectan al control. Por ejemplo, el cambio del color de fondo de una etiqueta Label no tendrá efecto dado que el control Label no tiene una ventana propia gtk.gdk.Window. El color de fondo de una Etiqueta depende del color de fondo de su control padre. Pero es posible meter la Etiqueta en un control EventBox que añade una ventana y permite así cambiar su color de fondo. En la sección EventBox se proporciona un ejemplo de esta técnica.

Capítulo 19

Temporizadores, Entrada/Salida y Funciones de Inactividad

19.1. Temporizadores

Puede que estes preguntándote cómo hacer que GTK haga algo útil mientras está dentro de la función `main`(). Bien, tienes varias opciones. Usando la siguiente función puedes crear un temporizador que será llamado en intervalos regulares (en milisegundos).

```
source_id = gobject.timeout_add(interval, function, ...)
```

El argumento *`interval`* es el número de milisegundos entre llamadas sucesivas a tu función. El argumento *`function`* es la operación que deseas que se llame. Cualquier argumento tras el segundo se pasará a tu función como datos. El valor de retorno *`source_id`* es un entero, que se puede utilizar para eliminar el temporizador llamando a:

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

También se puede parar el temporizador devolviendo cero o FALSE (falso) desde tu función. Obviamente esto significa que si quieres que el temporizador se siga llamando, debes devolver un valor distinto de cero, como TRUE (verdadero).

Tu función será algo parecido a:

```
def timeout_callback(...):
```

El número de argumentos a tu función debe coincidir con el número de argumentos de datos especificados en `timeout_add`().

19.2. Monitorizar la Entrada/Salida

Puedes comprobar si hay algo para leer o escribir a un fichero (bien a un fichero Python o a un fichero de más bajo nivel del Sistema Operativo) y automáticamente invocar una función. Ésto es especialmente útil para aplicaciones de red. La función:

```
source_id = gobject.io_add_watch(source, condition, callback)
```

donde el primer argumento (*`source`*) es el fichero abierto (objeto de fichero de Python o descriptor de fichero de más bajo nivel) que quieres monitorizar. La función `gobject.io_add_watch`() usa internamente el descriptor de fichero de bajo nivel, pero la función lo podrá extraer usando el método fileno() cuando sea necesario. El segundo argumento (*`condition`*) especifica qué es lo que se quiere monitorizar. Puede ser cualquiera de:

```
gobject.IO_IN - Llama a tu función cuando en el fichero hay datos disponibles ↩
    para leer.

gobject.IO_OUT - Llama a tu función cuando el fichero está listo para escritura ↩
    .
```

```
gobject.IO_PRI - Llama a tu función cuando el fichero tiene datos urgentes para ↩
     leer.

gobject.IO_ERR - Llama a tu función cuando se da una condición de error.

gobject.IO_HUP - Llama a tu función cuando se ha producido un "cuelgue" (se ha  ↩
    roto la conexión, de uso para pipes y sockets generalmente).
```

Estas constantes se definen en el módulo gobject module. Supongo que ya te has dado cuenta que el tercer argumento, *callback*, es la función que quieres que se llame cuando las condiciones anteriores se cumplen.

El valor de retorno, *source_id* se puede usar para parar de monitorizar el fichero usando la siguiente función:

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

Tu función será algo parecido a:

```
def input_callback(source, condition):
```

donde *source* y *condition* son los que especificaste antes. El valor de source será el descriptor de fichero de bajo nivel y no el objeto fichero Python (es decir, el valor que devuelve el método de fichero de Python `fileno`()).

También se puede parar la monitorización devolviendo cero o FALSE (falso) desde tu función. Obviamente esto significa que si quieres que el monitorización se siga llamando, debes devolver un valor distinto de cero, como TRUE (verdadero).

19.3. Funciones de Inactividad

¿Qué pasa si quieres que se llame a una función cuando no esté pasando nada? Usa la función:

```
source_id = gobject.idle_add(callback, ...)
```

Cualquier argumento tras el primero (indicados con ...) se pasan a la función *callback* en orden. El valor de retorno *source_id* se utiliza como una referencia al manejador.

Esta función hace que GTK llame a la función especificada cuando no está pasando nada más.

Y la función ha ser llamada debe ser parecida a:

```
def callback(...):
```

donde los argumentos pasados a `callback` son los mismos especificados en la función `gobject.idle_add()`. Al igual que en otras funciones, devolviendo FALSE (falso) dejará de ser llamada de nuevo, y devolviendo TRUE (verdadero) se la seguirá llamando la próxima ocasión que haya un tiempo de inactividad.

Se puede eliminar una función de inactividad de la cola llamando la función siguiente:

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

siendo *source_id* el valor devuelto por la función `gobject.idle_add`().

Capítulo 20

Procesamiento Avanzado de Eventos y Señales

20.1. Métodos de Señales

Los métodos de señales son métodos de la clase `gobject.GObject` que se heredan en los `gtk.Objects` incluyendo todos los controles GTK+.

20.1.1. Conectar y Desconectar Manejadores de Señal

```
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)
```

Los primeros cuatro métodos conectan un manejador de señales (*cb*) a un objeto `gtk.Object` (*object*) para la señal especificada por *name*, y devuelven un valor *handler_id* que identifica la conexión. *cb_args* son cero o más argumentos que serán pasados al final de la llamada al manejador *cb*. Los métodos connect_after() y connect_object_after() harán que se llame a sus manejadores después de haber llamado a todos los demás manejadores (incluyendo los manejadores predeterminados) que estén conectados a ese mismo objecto y señal. Cada manejador de señales de un objeto tiene su propio conjunto de argumentos. Resulta útil consultar la documentación de GTK+ para averiguar qué argumentos se deben usar en cada manejador de señales, aunque se proporciona información sobre los controles más comunes en el apéndice Señales de GTK+. El manejador de señales genérico es similar a este:

```
def signal_handler(object, ...., cb_args):
```

Los manejadores de señales que se definen como métodos de una clase Python (especificados en los métodos connect() como *self.cb*) tendrán un argumento adicional como primer argumento, la instancia del objeto *self*:

```
signal_handler(self, object, ...., cb_args)
```

Los métodos connect_object() y connect_object_after() llaman al manejador de señales con el *slot_object* en lugar del *object* como primer argumento:

```
def signal_handler(slot_object, ...., func_args):

def signal_handler(self, slot_object, ...., func_args):
```

El método disconnect() elimina la conexión entre un manejador y la señal de un objeto. El argumento *handler_id* especifica la conexión que se eliminará.

20.1.2. Bloqueo y Desbloqueo de Manejadores de Señal

Los siguientes métodos:

```
object.handler_block(handler_id)

object.handler_unblock(handler_id)
```

bloquean y desbloquean el manejador de señal especificado en el argumento *handler_id*. Cuando un manejador de señal está bloqueado no es invocado cuando se produce la señal.

20.1.3. Emisión y Parada de Señales

Los siguientes métodos:

```
object.emit(name, ...)

object.emit_stop_by_name(name)
```

emiten y paran, respectivamente, la señal especificada en el argumento *name*. La emisión de la señal hace que se ejecuten el manejador predeterminado y los definidos por el usuario. El método emit_stop_by_name() abortará la emisión de señales actual.

20.2. Emisión y Propagación de Señales

La emisión de señales es el proceso por el cual GTK+ ejecuta todos los manejadores de una señal y un objeto específicos.

Debe tenerse en cuenta en primer lugar que el valor de retorno de una emisión de señal es el valor de retorno del último manejador ejecutado. Como las señales de eventos son todas del tipo `RUN_LAST`, este valor será el del manejador predeterminado (dado por GTK+) a menos que se use el método connect_after().

La forma en la que se trata un evento (por ejemplo "button_press_event") es:

- Empezar con el control donde se produjo el evento.
- Se emite para él la señal genérica "event". Si ese manejador devuelve un valor `TRUE` (verdadero) se detiene el procesamiento.
- En otro caso, se emite una señal específica "button_press_event". Si ésta devuelve `TRUE` (verdadero) se detiene el procesamiento.
- En caso contrario, se pasa al padre del control y se repiten los dos pasos anteriores.
- Se continúa hasta que algún manejador devuelve `TRUE`, o hasta que se alcanza al control de más alto nivel.

Algunas consecuencias de lo anterior son:

- El valor de retorno de tu manejador no tendrá efecto si hay un manejador predeterminado, a menos que lo conectes con connect_after().
- Para evitar que se llame al manejador predeterminado, tienes que usar el método connect() y utilizar emit_stop_by_name() - el valor de retorno sólo afecta a si la señal se propaga, pero no a la emisión actual.

Capítulo 21

Tratamiento de Selecciones

21.1. Descripción General de la Selección

Uno de los mecanismos de comunicación entre procesos que está disponible en X y GTK+ es las selecciones. Una selección identifica una porción de datos (p.e. un trozo de texto), seleccionado por el usuario de alguna manera (p.e. arrastrando con el ratón). Sólo una aplicación en un display (Nota del traductor: esto es terminología X) (el dueño) puede poseer una selección particular en un momento dado, por lo que cuando una aplicación solicita una selección, el dueño anterior debe indicarle al usuario que la selección ha sido cedida. Otras aplicaciones pueden solicitar los contenidos de una selección con diferentes formatos, a los que se llama objetivos. Puede haber múltiples selecciones, pero la mayoría de las aplicaciones X sólo tratarán una, la selección primaria.

En la mayoría de los casos, no es necesario que una aplicación PyGTK maneje directamente las selecciones. Los controles estándar, como el control Entry, ya tienen la capacidad de solicitar la selección cuando sea necesario (por ejemplo, cuando el usuario arrastra el ratón sobre el texto), y de recuperar el contenido de la selección de otros controles o aplicaciones (por ejemplo, cuando el usuario hace clic en el segundo botón del ratón). Sin embargo, puede haber casos en los que se quiera dar a los controles la capacidad de entregar la selección, o puede que se quieran obtener objetivos no disponibles en principio.

Un concepto fundamental necesario para comprender el manejo de selecciones es el "atom" (átomo). Un átomo es un número entero que identifica inequívocamente una cadena de caracteres (en un display concreto). Algunos átomos se encuentran predefinidos por parte del servidor X y por GTK+.

21.2. Recuperar la Selección

Recuperar la selección es un proceso asíncrono. Para iniciar el proceso es necesario llamar a:

```
result = widget.selection_convert(selection, target, time=0)
```

Que convierte la selección *selection* en el formato especificado por el objetivo *target*. La selección *selection* es un átomo correspondiente al tipo de selección. Las selecciones comunes son las cadenas de texto:

```
PRIMARY    # primaria

SECONDARY  # secundaria
```

Si es posible, el campo tiempo *time* debería ser el momento en el que se produjo el evento que ocasionó la selección. Esto ayuda a cerciorarse que los eventos ocurren en el orden en el que el usuario los solicita. Sin embargo, si no está disponible (por ejemplo, si la conversión se produjo en una señal "clicked"), entonces se puede utilizar 0, que significa el momento actual. El resultado será `TRUE` (verdadero) si la conversión tuvo éxito, o `FALSE` (falso) en caso contrario.

Cuando el dueño de la selección responde a la petición, una señal "selection_received" (selección_recibida) se envia a la aplicación. El manejador de esta señal recibe un objeto de tipo `gtk.SelectionData`, que tiene los siguientes atributos:

```
selection # selección
target    # objetivo
```

```
type      # tipo
format    # formato
data      # datos
```

selection y *target* son los valores que se dieron en el método selection_convert() .

type es un átomo que identifica el tipo de datos que ha devuelto el dueño de la selección. Algunos valores posibles son "STRING", una cadena de caracteres latin-1, "ATOM", una serie de átomos, "INTEGER", un entero, "image/x-xpixmap", etc. La mayoría de los objetivos sólo pueden devolver un tipo.

La lista de átomos estándar en X y GTK+ es:

```
PRIMARY
SECONDARY
ARC
ATOM
BITMAP
CARDINAL
COLORMAP
CURSOR
CUT_BUFFER0
CUT_BUFFER1
CUT_BUFFER2
CUT_BUFFER3
CUT_BUFFER4
CUT_BUFFER5
CUT_BUFFER6
CUT_BUFFER7
DRAWABLE
FONT
INTEGER
PIXMAP
POINT
RECTANGLE
RESOURCE_MANAGER
RGB_COLOR_MAP
RGB_BEST_MAP
RGB_BLUE_MAP
RGB_DEFAULT_MAP
RGB_GRAY_MAP
RGB_GREEN_MAP
RGB_RED_MAP
STRING
VISUALID
WINDOW
WM_COMMAND
WM_HINTS
WM_CLIENT_MACHINE
WM_ICON_NAME
WM_ICON_SIZE
WM_NAME
WM_NORMAL_HINTS
WM_SIZE_HINTS
WM_ZOOM_HINTS
MIN_SPACE
NORM_SPACE
MAX_SPACE  END_SPACE,
SUPERSCRIPT_X
SUPERSCRIPT_Y
SUBSCRIPT_X
SUBSCRIPT_Y
UNDERLINE_POSITION
UNDERLINE_THICKNESS
STRIKEOUT_ASCENT
STRIKEOUT_DESCENT
ITALIC_ANGLE
```

```
X_HEIGHT
QUAD_WIDTH
WEIGHT
POINT_SIZE
RESOLUTION
COPYRIGHT
NOTICE
FONT_NAME
FAMILY_NAME
FULL_NAME
CAP_HEIGHT
WM_CLASS
WM_TRANSIENT_FOR
CLIPBOARD
```

format da la longitud de las unidades (por ejemplo caracteres) en bits. Normalmente se puede obviar este parámetro al recibir datos.

data contiene los datos devueltos en forma de cadena de texto.

PyGTK empaqueta todos los datos recibidos en una cadena de texto. Esto hace que sea fácil manipular objetivos de cadenas de texto. Para obtener objetivos de otros tipos (como ATOM o INTEGER) el programa debe extraer la información de la cadena de texto devuelta. PyGTK proporciona dos métodos para obtener texto y una lista de objetivos a partir de los datos de la selección:

```
text = selection_data.get_text()

targets = selection_data.get_targets()
```

donde *text* es una cadena de texto que contiene el texto de la selección y *targets* es una lista de los objetivos que acepta la selección.

Dado un `gtk.SelectionData` que contiene una lista de objetivos, el método:

```
has_text = selection_data.targets_include_text()
```

devolverá `TRUE` (verdadero) si uno o más de los objetivos pueden proporcionar texto.

El programa de ejemplo getselection.py enseña cómo recibir un objetivo "STRING" o "TARGETS" desde la selección primaria para luego imprimir los datos correspondientes en la consola al hacer clic en el botón asociado. La figura Figura 21.1 muestra la ventana del programa:

Figura 21.1 Ejemplo de Obtención de la Selección

El código fuente de getselection.py es:

```
1    #!/usr/bin/env python
2
3    # ejemplo getselection.py
4
5    import pygtk
6    pygtk.require('2.0')
7    import gtk
8
9    class GetSelectionExample:
10       # Gestor de la señal que se llama cuando la usuaria
11       # pulsa el botón "Get String Target"
12       def get_stringtarget(self, widget):
13           # Se solicita el objetivo "STRING" para la selección primaria
```

```
14            ret = widget.selection_convert("PRIMARY", "STRING")
15            return
16
17        # Gestor de señal que se invoca cuando la usuaria pulsa el botón " ↩
    Get Targets"
18        def get_targets(self, widget):
19            # Se solicita el objetivo "TARGETS" para la selección primaria
20            ret = widget.selection_convert("PRIMARY", "TARGETS")
21            return
22
23        # Gestor de señal llamado cuando el propietario de la selección  ↩
    devuelve los datos
24        def selection_received(self, widget, selection_data, data):
25            # Nos aseguramos de que los datos se reciben en el formato  ↩
    esperado
26            if str(selection_data.type) == "STRING":
27                # Mostramos la cadena recibida
28                print "STRING TARGET: %s" % selection_data.get_text()
29
30            elif str(selection_data.type) == "ATOM":
31                # Mostramos la lista de objetivos recibida
32                targets = selection_data.get_targets()
33                for target in targets:
34                    name = str(target)
35                    if name != None:
36                        print "%s" % name
37                    else:
38                        print "(bad target)"
39            else:
40                print "Selection was not returned as \"STRING\" or \"ATOM ↩
    \"!"
41
42            return gtk.FALSE
43
44
45        def __init__(self):
46            # Creamos la ventana principal
47            window = gtk.Window(gtk.WINDOW_TOPLEVEL)
48            window.set_title("Get Selection")
49            window.set_border_width(10)
50            window.connect("destroy", lambda w: gtk.main_quit())
51
52            vbox = gtk.VBox(gtk.FALSE, 0)
53            window.add(vbox)
54            vbox.show()
55
56            # Creamos un botón que al pulsarlo se obtiene la cadena de  ↩
    objetivo
57            button = gtk.Button("Get String Target")
58            eventbox = gtk.EventBox()
59            eventbox.add(button)
60            button.connect_object("clicked", self.get_stringtarget, eventbox ↩
    )
61            eventbox.connect("selection_received", self.selection_received)
62            vbox.pack_start(eventbox)
63            eventbox.show()
64            button.show()
65
66            # Creamos un botón que devuelve los objetivos al pulsarlo
67            button = gtk.Button("Get Targets")
68            eventbox = gtk.EventBox()
69            eventbox.add(button)
70            button.connect_object("clicked", self.get_targets, eventbox)
71            eventbox.connect("selection_received", self.selection_received)
```

```
72            vbox.pack_start(eventbox)
73            eventbox.show()
74            button.show()
75
76            window.show()
77
78    def main():
79        gtk.main()
80        return 0
81
82    if __name__ == "__main__":
83        GetSelectionExample()
84        main()
```

Las lineas 30-38 se encargan de la obtención de los datos de selección de "TARGETS" (objetivos) e imprime la lista de los nombres de los objetivos. Los botones están insertados en sus propias cajas de eventos puesto que una selección debe estar asociada a una `gtk.gdk.Window` y los botones son controles sin ventana propia en GTK+2.0.

21.3. Proporcionar la Selección

Proporcionar la selección es un poco más complicado. Se deben registrar los manejadores que se llamarán cuando se solicite la selección. Para cada par selección-objetivo a manejar, se debe hacer una llamada a:

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

widget (control), *selection* (selección), y *target* (objetivo) identifican las peticiones que gestionará este manejador. Cuando llegue una petición para una selección, se llamará a la señal "selection_get". *info* es un entero que se puede usar como identificador para el objetivo específico dentro de la retrollamada.

La retrollamada tiene la siguiente signatura:

```
def selection_get(widget, selection_data, info, time):
```

El argumento `gtk.SelectionData` es el mismo que antes, pero en esta ocasión se deben rellenar los campos *type*, *format* y *data*. (El campo *format* es importante aquí, ya que el servidor X lo usa para saber si el campo *data* necesita que se le cambie el orden de sus bytes o no. Normalmente será 8 - un caracter - ó 32 - un entero). Esto se hace llamando al método:

```
selection_data.set(type, format, data)
```

Este método de PyGTK sólo puede tratar datos de cadenas de caracteres, por lo que *data* debe cargarse en una cadena Python pero *format* tendrá el tamaño apropiado para los datos (por ejemplo 32 para átomos y enteros, 8 para cadenas). Los módulos Python `struct` o `StringIO` pueden servir para convertir datos que no son cadenas de caracteres a cadenas de caracteres. Por ejemplo, se puede convertir una lista de enteros en una cadena y ponerlos en el campo *selection_data* así:

```
ilist = [1, 2, 3, 4, 5]

data = apply(struct.pack, ['%di'%len(ilist)] + ilist)

selection_data.set("INTEGER", 32, data)
```

El siguiente método fija los datos de la selección a partir de dicha cadena:

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

Cuando la usuaria lo solicite, se debe reclamar la posesión de la selección llamando a:

```
result = widget.selection_owner_set(selection, time=0L)
```

result será TRUE (verdadero) si el programa reclama la selección *selection* con éxito. Si otra aplicación reclama la posesión de *selection*, entonces llegará un evento "selection_clear_event".

Como ejemplo de proporcionar la selección, el programa setselection.py añade la funcionalidad de selección a un botón biestado que está dentro de una gtk.EventBox. (Se necesita una gtk.Eventbox porque la selección debe asociarse a una gtk.gdk.Window y un gtk.Button es un control sin ventana en GTK+ 2.0.). Cuando se pulsa el botón biestado el programa reclama la selección primaria. El único objetivo soportado (aparte de algunos objetivos que proporciona la propia GTK+ como "TARGETS"), es el objetivo "STRING". Al solicitar este objetivo se devuelve una representación en cadena de caracteres de la hora actual. La figura Figura 21.2 muestra la ventana del programa cuando éste ha conseguido la posesión de la selección primaria:

Figura 21.2 Ejemplo de Fijar la Selección

El código fuente de setselection.py es:

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

# ejemplo setselection.py

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

class SetSelectionExample:
    # Retrollamada cuando la usuaria conmuta la selección
    def selection_toggled(self, widget, window):
        if widget.get_active():
            self.have_selection = window.selection_owner_set("PRIMARY")
            # si la solicitud de la selección falla, se devuelve el botón al
            # estado inactivo
            if not self.have_selection:
                widget.set_active(gtk.FALSE)
        else:
            if self.have_selection:
                # No es posible "liberar" la selección en PyGTK
                # simplemente se señala que no se posee
                self.have_selection = gtk.FALSE
            return

    # Llamada cuando otra aplicación reclama la selección
    def selection_clear(self, widget, event):
        self.have_selection = gtk.FALSE
        widget.set_active(gtk.FALSE)
        return gtk.TRUE

    # Proporciona la hora actual como selección
    def selection_handle(self, widget, selection_data, info, time_stamp):
        current_time = time.time()
        timestr = time.asctime(time.localtime(current_time))

        # Al devolver una cadena única no debe terminar en valor nulo
        # Esto se hace automáticamente
        selection_data.set_text(timestr, len(timestr))
```

```
40            return
41
42        def __init__(self):
43            self.have_selection = gtk.FALSE
44            # Creamos la ventana principal
45            window = gtk.Window(gtk.WINDOW_TOPLEVEL)
46            window.set_title("Set Selection")
47            window.set_border_width(10)
48            window.connect("destroy", lambda w: gtk.main_quit())
49            self.window = window
50            # Creamos una caja de eventos que contenga el botón, ya que no ↩
    tiene su
51            # propia gtk.gdk.Window
52            eventbox = gtk.EventBox()
53            eventbox.show()
54            window.add(eventbox)
55
56            # Creamos un botón biestado que actúe como selección
57            selection_button = gtk.ToggleButton("Claim Selection")
58            eventbox.add(selection_button)
59
60            selection_button.connect("toggled", self.selection_toggled, ↩
    eventbox)
61            eventbox.connect_object("selection_clear_event", self. ↩
    selection_clear,
62                                    selection_button)
63
64            eventbox.selection_add_target("PRIMARY", "STRING", 1)
65            eventbox.selection_add_target("PRIMARY", "COMPOUND_TEXT", 1)
66            eventbox.connect("selection_get", self.selection_handle)
67            selection_button.show()
68            window.show()
69
70    def main():
71        gtk.main()
72        return 0
73
74    if __name__ == "__main__":
75        SetSelectionExample()
76        main()
```

Capítulo 22

Arrastrar y Soltar

PyGTK posee un conjunto de funciones de alto nivel de comunicación entre procesos a través del sistema arrastrar-y-soltar. PyGTK puede realizar arrastrar-y-soltar sobre los protocolos de bajo nivel Xdnd y arrastrar-y-soltar Motif.

22.1. Descripción General de Arrastrar y Soltar

Una aplicación con la capacidad de arrastrar-y-soltar primero define y configura el/los control/es para arrastrar-y-soltar. Cada control puede ser una fuente y/o destino para arrastrar-y-soltar. Debe tenerse en cuenta que estos controles han de tener una ventana X asociada.

Los controles fuente pueden enviar datos de arrastrar, permitiendo así al usuario arrastrar cosas desde ellos, mientras que los controles destino pueden recibir datos de arrastrar. Los destinos de arrastrar-y-soltar pueden limitar quiénes pueden enviarles datos, por ejemplo, la propia aplicación o cualquier aplicación (incluyéndose a sí misma).

Para enviar y recibir datos se utilizan señales. Soltar un elemento en un control destino requiere una petición de datos (para el control fuente) y un manejador de datos recibidos (para el control destino). Se pueden conectar manejadores de señal adicionales si se quiere saber cuándo empieza el usuario a arrastrar (en el mismo momento en el que empieza), cuándo se realiza el soltar, y cuándo finaliza el proceso completo de arrastrar-y-soltar (con éxito o no).

La aplicación debe proporcionar los datos a los controles origen cuando se le sean solicitados, lo cual implica tener un manejador de señal para la solicitud de datos. Los controles destino han de tener un manejador de señales de datos recibidos.

Por tanto, un ciclo habitual de arrastrar-y-soltar sería así:

- Se empieza a arrastrar. El control fuente puede recibir la señal "drag-begin". Puede configurar el icono de arrastrar, etc.
- Se mueve lo arrastrado sobre el área de soltar. El control destino puede recibir la señal "drag-motion".
- Se suelta el botón. El control destino puede recibir la señal "drag-drop". El control destino debería solicitar los datos fuente.
- Se solicitan los datos de arrastrar (al soltar el botón). El control fuente puede recibir la señal "drag-data-get".
- Se reciben los datos (puede ser en la misma o en otra aplicación). El control destino puede recibir la señal "drag-data-received".
- Se borran los datos de arrastrar (si el arrastre fue un movimiento). El control fuente puede recibir la señal "drag-data-delete".
- El proceso arrastrar-y-soltar termina. El control fuente puede recibir la señal "drag-end".

Hay algunos pasos intermedios adicionales pero se verán en detalle un poco más tarde.

22.2. Propiedades de Arrastrar y Soltar

Los datos de arrastrar tienen las siguientes propiedades:

- Tipo de acción de arrastrar (por ejemplo `ACTION_COPY` (acción copiar), `ACTION_MOVE` (acción mover)).
- Tipo de arrastrar-y-soltar específico del cliente (un par de nombre y número).
- Tipo de formato de los datos enviados y recibidos.

Las acciones de arrastrar son bastante obvias, especifican si el control puede arrastrar con la/s acción/es especificada/s, por ejemplo `gtk.gdk.ACTION_COPY` y/o `gtk.gdk.ACTION_MOVE`. Una acción `gtk.gdk.ACTION_COPY` sería el típico arrastrar y soltar sin que la fuente se elimine mientras que una acción `gtk.gdk.ACTION_MOVE` sería exactamente igual, pero se 'sugiere' que se borren los datos orígen tras la llamada a la señal de recepción. Hay más acciones como `gtk.gdk.ACTION_LINK` que se pueden investigar en cuanto se adquiera un poco más de destreza con el mecanismo de arrastrar-y-soltar.

El tipo de arrastrar-y-soltar especificado por el cliente es mucho más flexible, porque será la aplicación la que lo defina y compruebe. Se tendrán que configurar los controles destino para que reciban ciertos tipos de arrastrar-y-soltar, especificando un nombre y/o un número. Es más fiable el uso de un nombre ya que otra aplicación puede estar usando el mismo número con un significado completamente diferente.

Los tipos de emisión y recepción de datos (*objetivo de selección*) entran en juego sólo en los propios manejadores de datos solicitados y recibidos. El término *objetivo de selección* es un poco confuso. Es un término adaptado de la selección GTK+ (cortar/copiar y pegar). Lo que *selection target* realmente significa es el tipo de formato de datos (por ejemplo `gtk.gdk.Atom`, entero, o cadena de caracteres) que se está enviando o recibiendo. El manejador de datos solicitados tiene que especificar el tipo (*selection target*) de datos que está enviando y el manejador de datos recibidos tiene que manejar el tipo de datos recibidos (*selection target*).

22.3. Métodos de Arrastrar y Soltar

22.3.1. Configuración del Control Origen

El método drag_source_set() especifica un conjunto de tipos objetivo para una operación de arrastrar en un control.

```
widget.drag_source_set(start_button_mask, targets, actions)
```

Los parámetros significan lo siguiente:

- *`widget`* especifica el control fuente
- *`start_button_mask`* especifica una máscara de bits de los botones que pueden empezar a arrastrar (por ejemplo `BUTTON1_MASK`).
- *`targets`* especifica una lista de los tipos de datos objetivos que se manejarán.
- *`actions`* especifica un máscara de bits de las acciones posibles para arrastrar desde esta ventana.

El parámetro *`targets`* es una lista de tuplas similar a:

```
(target, flags, info)
```

`target` especifica una cadena de caracteres que representa el tipo de arrastre.

`flags` restringe la aplicación del arrastre. *`flags`* puede ser 0 (sin limitación del ámbito) o las siguientes constantes:

```
  gtk.TARGET_SAME_APP    # El objetivo sólo se puede seleccionar para arrastres
dentro de una única aplicación.

  gtk.TARGET_SAME_WIDGET # El objetivo sólo se puede seleccionar para arrastres
dentro del mismo control.
```

info es un identificador entero asignado por la aplicación.

Si no es necesario que un control siga siendo el orígen de operaciones arrastrar-y-soltar, el método drag_source_unset() se puede usar para eliminar un conjunto de tipos de objetivos arrastrar-y-soltar.

```
widget.drag_source_unset()
```

22.3.2. Señales en el Control Fuente

Las siguientes señales se envian al control fuente durante una operación de arrastrar-y-soltar.

Cuadro 22.1 Señales del Control Fuente

drag_begin (comienzo de arrastre)	def drag_begin_cb(widget, drag_context, data):
drag_data_get (obtención de datos de arrastre)	def drag_data_get_cb(widget, drag_context, selection_data, info, time, data):
drag_data_delete (eliminación de datos de arrastre)	def drag_data_delete_cb(widget, drag_context, data):
drag_end (fin de arrastre)	def drag_end_cb(widget, drag_context, data):

El manejador de señal "drag-begin" se puede usar para configurar algunas condiciones iniciales tales como el icono de arrastre usando para ello uno de los siguientes métodos del la clase `Widget`: drag_source_set_icon(), drag_source_set_icon_pixbuf(), drag_source_set_icon_stock(). El manejador de señal "drag-end" puede usarse para deshacer las acciones del manejador de señal "drag-begin".

El manejador de señal "drag-data-get" debería devolver los datos de arrastre que coincidan con el objetivo especificado por *info*. Rellena `gtk.gdk.SelectionData` con los datos de arrastre.

El manejador de señal "drag-delete" se usa para borrar los datos de arrastre de una acción `gtk.gdk.ACTION_MOVE` tras haberlos copiado.

22.3.3. Configuración de un Control Destino

drag_dest_set() especifica que este control puede ser destino de operaciones arrastrar-y-soltar y define los tipos que admite.

drag_dest_unset() especifica que el control no puede recibir más operaciones arrastrar-y-soltar.

```
widget.drag_dest_set(flags, targets, actions)

widget.drag_dest_unset()
```

flags especifica qué acciones debe realizar GTK+ por parte del control cuando se suelte algo en él. Los valores posibles son:

gtk.DEST_DEFAULT_MOTION Si está activado para un control, GTK+ comprobará si el arrastre se corresponde con algún objetivo y acción del control cuando se arrastre por encima de él. Entonces GTK+ llamará a drag_status() según corresponda.

gtk.DEST_DEFAULT_HIGHLIGHT Si está activado para un control, GTK+ resaltará el control siempre que el arrastre esté encima del mismo y el formato y la acción sean aceptables.

gtk.DEST_DEFAULT_DROP Si está activado para un control, GTK+ comprobará si el arrastre se corresponde con algún objetivo y acción de dicho control. Si es así, GTK+ llamará a drag_data_get() de parte del control. No importa si el soltar tiene éxito o no, GTK+ llamará a drag_finish(). Si la acción fue mover y el arrastre tuvo éxito, entonces se pasará `TRUE` como argumento *delete* (borrar) a drag_finish().

gtk.DEST_DEFAULT_ALL Si está activo, especifica que todas las acciones anteriores deben ejecutarse.

targets es una lista de tuplas de información de objetivos tal y como se describe más arriba.

actions es una máscara de bits de las posibles acciones que realizar cuando se arrastre sobre este control. Los valores posibles se pueden componer con la operación OR y son los siguientes:

```
gtk.gdk.ACTION_DEFAULT # acción predeterminada
gtk.gdk.ACTION_COPY    # acción copiar
gtk.gdk.ACTION_MOVE    # acción mover
gtk.gdk.ACTION_LINK    # acción enlazar
gtk.gdk.ACTION_PRIVATE # acción privada
gtk.gdk.ACTION_ASK     # acción preguntar
```

targets y *actions* son ignoradas si *flags* no contiene `gtk.DEST_DEFAULT_MOTION` o `gtk.DEST_DEFAULT_DROP`. En este caso la aplicación debe manejar las señales "drag-motion" y "drag-drop".

El manejador de señal "drag-motion" debe determinar si los datos de arrastre son apropiados comparando para ello los objetivos del destino con los objetivos `gtk.gdk.DragContext` y opcionalmente examinando los datos de arrastre llamando el método drag_get_data() method. Se debe llamar al método `gtk.gdk.DragContext`. drag_status() para actualizar el estado de *drag_context*.

El manejador de señal "drag-drop" debe determinar el objetivo que corresponda usando el método del control drag_dest_find_target() y después solicitar los datos de arrastre mediante el método del control drag_get_data(). Los datos estarán disponibles en el manejador "drag-data-received".

El programa dragtargets.py muestra todos los posibles objetivos de una operación de arrastre en un control de tipo etiqueta (label):

```
 1  #!/usr/local/env python
 2
 3  import pygtk
 4  pygtk.require('2.0')
 5  import gtk
 6
 7  def motion_cb(wid, context, x, y, time):
 8      context.drag_status(gtk.gdk.ACTION_COPY, time)
 9      return True
10
11  def drop_cb(wid, context, x, y, time):
12      l.set_text('\n'.join([str(t) for t in context.targets]))
13      return True
14
15  w = gtk.Window()
16  w.set_size_request(200, 150)
17  w.drag_dest_set(0, [], 0)
18  w.connect('drag_motion', motion_cb)
19  w.connect('drag_drop', drop_cb)
20  w.connect('destroy', lambda w: gtk.main_quit())
21  l = gtk.Label()
22  w.add(l)
23  w.show_all()
24
25  gtk.main()
```

El programa crea una ventana que se configura como destino de arrastre para ningún objetivo y ninguna acción poniendo las banderas (flags) a cero. Se conectan los manejadores `motion_cb`() y `drop_cb`() a las señales "drag-motion" y "drag-drop" respectivamente. El manejador `motion_cb`() configura el estado de arrastre para que se permita soltar. `drop_cb`() modifica el texto de la etiqueta a una cadena de caracteres que contenga los objetivos de arrastre e ignora los datos de tal modo que la acción de soltar no se completa en ningún caso..

22.3.4. Señales en el Control Destino

Durante una operación arrastrar-y-soltar se envian las siguientes señales al control destino.

El programa de ejemplo dragndrop.py muestra el uso de arrastrar y soltar en una aplicación. Un botón con un icono xpm (en gtkxpm.py) es el origen del arrastre y proporciona tanto texto como datos xpm. Un control de disposición es el destino para soltar el xpm mientras que un botón es el destino para soltar el texto. La figura Figura 22.1 ilustra la ventana del programa tras soltar el xpm en el control de disposición y el texto en el botón:

Cuadro 22.2 Señales del Control Destino

drag_motion (movimiento de arrastre)	def drag_motion_cb(widget, drag_context, x, y, time, data):
drag_drop (arrastre soltado)	def drag_drop_cb(widget, drag_context, x, y, time, data):
drag_data_received (datos recibidos)	def drag_data_received_cb(widget, drag_context, x, y, selection_data, info, time, data):

Figura 22.1 Ejemplo de Arrastrar y Soltar

El código fuente de dragndrop.py es:

```
1    #!/usr/bin/env python
2
3    # ejemplo dragndrop.py
4
5 import pygtk
6 pygtk.require('2.0')
7 import gtk
8 import string, time
9
10   import gtkxpm
11
12   class DragNDropExample:
13       HEIGHT = 600
14       WIDTH = 600
15       TARGET_TYPE_TEXT = 80
16       TARGET_TYPE_PIXMAP = 81
17       fromImage = [ ( "text/plain", 0, TARGET_TYPE_TEXT ),
18                     ( "image/x-xpixmap", 0, TARGET_TYPE_PIXMAP ) ]
19       toButton = [ ( "text/plain", 0, TARGET_TYPE_TEXT ) ]
20       toCanvas = [ ( "image/x-xpixmap", 0, TARGET_TYPE_PIXMAP ) ]
```

```
21
22        def layout_resize(self, widget, event):
23            x, y, width, height = widget.get_allocation()
24            if width > self.lwidth or height > self.lheight:
25                self.lwidth = max(width, self.lwidth)
26                self.lheight = max(height, self.lheight)
27                widget.set_size(self.lwidth, self.lheight)
28
29        def makeLayout(self):
30            self.lwidth = self.WIDTH
31            self.lheight = self.HEIGHT
32            box = gtk.VBox(gtk.FALSE,0)
33            box.show()
34            table = gtk.Table(2, 2, gtk.FALSE)
35            table.show()
36            box.pack_start(table, gtk.TRUE, gtk.TRUE, 0)
37            layout = gtk.Layout()
38            self.layout = layout
39            layout.set_size(self.lwidth, self.lheight)
40            layout.connect("size-allocate", self.layout_resize)
41            layout.show()
42            table.attach(layout, 0, 1, 0, 1, gtk.FILL|gtk.EXPAND,
43                         gtk.FILL|gtk.EXPAND, 0, 0)
44            # se crean las barras de desplazamiento que se empaquetan en la  ↩
    tabla
45            vScrollbar = gtk.VScrollbar(None)
46            vScrollbar.show()
47            table.attach(vScrollbar, 1, 2, 0, 1, gtk.FILL|gtk.SHRINK,
48                         gtk.FILL|gtk.SHRINK, 0, 0)
49            hScrollbar = gtk.HScrollbar(None)
50            hScrollbar.show()
51            table.attach(hScrollbar, 0, 1, 1, 2, gtk.FILL|gtk.SHRINK,
52                         gtk.FILL|gtk.SHRINK,
53                         0, 0)
54            # indicamos que las barras de desplazamiento usen los ajustes  ↩
    del control disposición
55            vAdjust = layout.get_vadjustment()
56            vScrollbar.set_adjustment(vAdjust)
57            hAdjust = layout.get_hadjustment()
58            hScrollbar.set_adjustment(hAdjust)
59            layout.connect("drag_data_received", self.receiveCallback)
60            layout.drag_dest_set(gtk.DEST_DEFAULT_MOTION |
61                                 gtk.DEST_DEFAULT_HIGHLIGHT |
62                                 gtk.DEST_DEFAULT_DROP,
63                                 self.toCanvas, gtk.gdk.ACTION_COPY)
64            self.addImage(gtkxpm.gtk_xpm, 0, 0)
65            button = gtk.Button("Text Target")
66            button.show()
67            button.connect("drag_data_received", self.receiveCallback)
68            button.drag_dest_set(gtk.DEST_DEFAULT_MOTION |
69                                 gtk.DEST_DEFAULT_HIGHLIGHT |
70                                 gtk.DEST_DEFAULT_DROP,
71                                 self.toButton, gtk.gdk.ACTION_COPY)
72            box.pack_start(button, gtk.FALSE, gtk.FALSE, 0)
73            return box
74
75        def addImage(self, xpm, xd, yd):
76            hadj = self.layout.get_hadjustment()
77            vadj = self.layout.get_vadjustment()
78            style = self.window.get_style()
79            pixmap, mask = gtk.gdk.pixmap_create_from_xpm_d(
80                self.window.window, style.bg[gtk.STATE_NORMAL], xpm)
81            image = gtk.Image()
82            image.set_from_pixmap(pixmap, mask)
```

```
 83            button = gtk.Button()
 84            button.add(image)
 85            button.connect("drag_data_get", self.sendCallback)
 86            button.drag_source_set(gtk.gdk.BUTTON1_MASK, self.fromImage,
 87                                   gtk.gdk.ACTION_COPY)
 88            button.show_all()
 89            # ajustamos según el desplazamientos de la disposición - la posición del evento
 90            # es relativo a la zona visible, no al tamaño del control disposición
 91            self.layout.put(button, int(xd+hadj.value), int(yd+vadj.value))
 92            return
 93
 94        def sendCallback(self, widget, context, selection, targetType, eventTime):
 95            if targetType == self.TARGET_TYPE_TEXT:
 96                now = time.time()
 97                str = time.ctime(now)
 98                selection.set(selection.target, 8, str)
 99            elif targetType == self.TARGET_TYPE_PIXMAP:
100                selection.set(selection.target, 8,
101                              string.join(gtkxpm.gtk_xpm, '\n'))
102
103        def receiveCallback(self, widget, context, x, y, selection, targetType,
104                            time):
105            if targetType == self.TARGET_TYPE_TEXT:
106                label = widget.get_children()[0]
107                label.set_text(selection.data)
108            elif targetType == self.TARGET_TYPE_PIXMAP:
109                self.addImage(string.split(selection.data, '\n'), x, y)
110
111        def __init__(self):
112            self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
113            self.window.set_default_size(300, 300)
114            self.window.connect("destroy", lambda w: gtk.main_quit())
115            self.window.show()
116            layout = self.makeLayout()
117            self.window.add(layout)
118
119    def main():
120        gtk.main()
121
122    if __name__ == "__main__":
123        DragNDropExample()
124        main()
```

Capítulo 23

Ficheros rc de GTK+

GTK tiene su propia forma de tratar las opciones predeterminadas de las aplicaciones, mediante ficheros rc. Con ellos se pueden cambiar los colores de cualquier control, y también se pueden usar para poner un dibujo de fondo en algunos controles.

23.1. Funciones para Ficheros rc

Cuando comience tu aplicación, debes incluir una llamada a:

```
rc_parse(filename)
```

En donde *filename* contiene el nombre de un fichero rc. Entonces GTK+ analiza el fichero y usa los valores de estilo a los tipos de controles que se definan en él.

Si se desea tener un conjunto especial de controles que puedan tener un estilo diferente a los demás, o cualquier otra división lógica de controles, se debe usar una llamada a:

```
widget.set_name(name)
```

Se asignará el nombre especificado en el parámetro *name* al control *widget*. Esto permitirá modificar los atributos de este control en el fichero rc usando su nombre.

Si se usa una llamada parecida a:

```
button = gtk.Button("Special Button")

button.set_name("special button")
```

Entonces se le dará el nombre "special button" al botón *button* lo que permite localizarlo en el fichero rc como "special button.GtkButton". [--- Verifíquese!]

El ejemplo de fichero rc de más abajo, modifica las propiedades de la ventana principal y permite a todos sus controles hijos heredar el estilo descrito como el estilo "main button" (botón principal). El código usado por la aplicación es:

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

window.set_name("main window")
```

Y el estilo se define entonces en el fichero rc así:

```
widget "main window.*GtkButton*" style "main_button"
```

Que aplica a todos los controles Button de la "main window" (ventana principal) el estilo "main_buttons" tal como se define en el fichero rc.

Como se puede ver, éste es un sistema bastante potente y flexible. Use su imaginación para sacarle el mejor provecho.

23.2. Formato de los Ficheros rc de GTK+

El formato del fichero rc se muestra en el siguiente ejemplo. Éste es el fichero `testgtkrc` de la distribución GTK+, pero se le han añadido unos algunos comentarios y cosas varias. Se puede incluir esta explicación en los programas para permitir al usuario afinar su aplicación.

Existen varias directivas para modificar los atributos de un control.

- *fg* - Cambia el color de frente de un control.
- *bg* - Cambia el color de fondo de un control.
- *bg_pixmap* - Cambia el fondo de un control aun mosaico según el pixmap dado.
- *font* - Cambia la fuente que usará el control.

Además de estas propiedades también existen varios estados en los que puede encontrarse un control y se pueden cambiar los diferentes colores, pixmaps y fuentes de cada estado. Estos estados son:

NORMAL (Normal) El estado normal de un control, sin que el ratón esté encima de él, sin que haya sido pulsado, etc.

PRELIGHT (Preiluminado) Cuando el ratón está encima del control, los colores definidos en este estado tendrán efecto.

ACTIVE (Activo) Cuando el control está presionado o se ha hecho clic en él estará activo y los atributos asignados con este valor tendrán efecto.

INSENSITIVE (Insensible) Cuando un control está insensible, y no puede ser activado, tendrá estos atributos.

SELECTED (Seleccionado) Cuando un objeto está seleccionado se usan estos atributos.

Cuando se usan las palabras "fg" y "bg" para modificar los colores de los controles, el formato es:

```
fg[<STATE>] = { Rojo, Verde, Azul }
```

Donde `STATE` es uno de los estados anteriores (PRELIGHT, ACTIVE, etc), y *Rojo*, *Verde* y *Azul* son valores en el rango de 0 - 1.0, siendo { 1.0, 1.0, 1.0 } el blanco. Deben estar en formato float (decimal), o se les asignará 0. Por tanto, un simple "1" no es correcto, debe ser "1.0". Un simple "0" funciona, ya que los valores no reconocidos se establecen como 0.

bg_pixmap es muy similar al anterior excepto que los colores se sustituyen por un nombre de fichero.

pixmap_path es una lista de rutas separadas por ":". Cuando se especifique un pixmap, se buscará en esta lista.

La directiva "font" (fuente) es muy sencilla:

```
font = "<font name>"
```

Lo único dificil es saber la cadena de la fuente *font*. Y, para ello, el programa **xfontsel** o una utilidad similar puede ser de gran ayuda.

La directiva "widget_class" determina el estilo de una clase de controles. Estas clases se enumeran en el resumen general de controles en la jerarquía de clases.

La directiva "widget" determina el estilo de un conjunto de controles con un nombre específico, reemplazando cualquier otro estilo de la clase del control en cuestión. Estos controles se registran en la aplicación usando el método set_name() . Esto permite cambiar los atributos de un control para cada control concreto, en vez de especificar los atributos de toda su clase. Se recomienda documentar todos los controles especiales para que los usuarios puedan personalizarlos.

Si se usa la palabra *parent* (padre) como atributo el control usará los atributos de su padre en la aplicación.

Al definir un estilo, se pueden asignar los atributos de un estilo definido previamente al estilo actual.

```
style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

Este ejemplo usa el estilo "button", y crea un nuevo estilo "main_button" cambiando simplemente la fuente y el color de fondo del estado preiluminado del estilo "button" (botón).

Por supuesto, muchos de estos atributos no funcionan con todos los controles. Es sólo cuestión de sentido común. Todo lo que podría ser de aplicación, debería funcionar.

23.3. Ejemplo de fichero rc

```
# 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>

# Aquí sigue una lista de todos los estados posibles. Obsérvese que algunos
# de ellos no pueden aplicarse aciertos controles.
#
# NORMAL - El estado normal de un control, sin que el ratón esté encima
# de él, sin que haya sido pulsado, etc.
#
# PRELIGHT (Preiluminado) - Los colores definidos en este estado tendrán efecto
# cuando el ratón esté encima del control.
#
# ACTIVE (Activo) - Cuando el control está presionado o se ha hecho clic en
# él estará activo y los atributos asignados con este valor tendrán efecto.
#
# INSENSITIVE (Insensitivo) - Un control tendrá estos atributos cuando está
# insensible y no puede ser activado.
#
# SELECTED (Seleccionado) - Cuando un objeto está seleccionado se
# usan estos atributos.
#
# Dados estos estados podemos establecer los atributos de los controles en cada ↩
    uno
# de ellos utilizando las directivas siguientes:
#

# fg - Cambia el color de frente o primer plano de un control.
# bg - Cambia el color de fondo de un control.
# bg_pixmap - Cambia el fondo de un control a un mosaico con
#             el pixmap dado.
# font - Cambia la fuente que usará el control.
#

# El siguiente fragmento crea un estilo llamado "button" (botón). El nombre no es
# realmente importante, pues se asigna a los controles presentes al final
# del archivo.

style "window"
{
  # Fija el margen (padding) en torno a la ventana al pixmap especificado.
  # bg_pixmap[<STATE>] = "<pixmap filename>"
  bg_pixmap[NORMAL] = "warning.xpm"
}

style "scale"
```

```
{
  # Pone el color de frente (color de la fuente) a rojo cuando el estado
  # es "NORMAL"

  fg[NORMAL] = { 1.0, 0, 0 }

  # Pone el dibujo de fondo de este control igual al que tenga su padre.
  bg_pixmap[NORMAL] = "<parent>"
}

style "button"
{
  # Esto muestra todos los posibles estados para un botón. El único que no
  # se aplica es el estado SELECTED (seleccionado).

  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 }
}

# En este ejemplo, para crear el nuevo estilo "main_button" (botón principal),
# heredamos los atributos del estilo "button" y luego reemplazamos la
# fuente y el color de fondo del estado "preiluminado".

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 }

  # Esto pone la imagen de fondo del toggle_button a la misma que tenga
  # su padre (según se defina en la aplicación).
  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"

# Lo siguiente hace que los diferentes tipos de controles citados usen
# los estilos definidos arriba.
# Estas clases se enumeran en el resumen general de controles en la
# jerarquía de clases, y probablemente también en este documento, como
# referencia del usuario.
```

```
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"

# Esto hace que todos los hijos de "main window" tengan el estilo main_button.
# Debiera ser documentado para poder sacarle partido.
widget "main window.*GtkButton*" style "main_button"
```

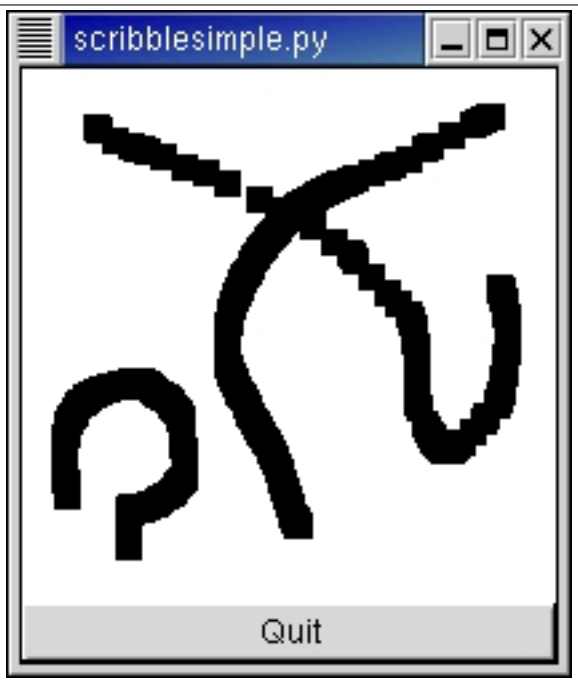
Capítulo 24

Scribble: Un Ejemplo Sencillo de Programa de Dibujo

24.1. Perspectiva General de Scribble

En esta sección, construiremos un programa sencillo de dibujo. En el proceso examinaremos cómo manejar eventos de ratón, cómo dibujar en una ventana y cómo dibujar mejor usando un pixmap de fondo.

Figura 24.1 Ejemplo de Programa de Dibujo Scribble



24.2. Manejo de Eventos

Las señales GTK+ que hemos explicado sirven para acciones de alto nivel, tal como la selección de un elemento de menú. Sin embargo, a veces es útil tener información sobre eventos de más bajo nivel, como cuándo se mueve un ratón o se pulsa una tecla. También hay señales para estos eventos de bajo nivel. Los manejadores de estas señales tienen un parámetro adicional, que es un objeto `gtk.gdk.Event`, y contiene información sobre el evento. Por ejemplo, a los manejadores de eventos de movimiento se les pasa un objeto `gtk.gdk.Event` que contiene información de EventMotion, que (parcialmente) contiene atributos como:

```
  type        # tipo
```

```
  window     # ventana
  time       # tiempo
  x
  y
    ...
  state      # estado
    ...
```

window es la ventana en la que ha ocurrido el evento.
x e *y* proporcionan las coordenadas del evento.
type informa del tipo de evento, en este caso MOTION_NOTIFY. Estos tipos (incluidos en el módulo gtk.gdk) son:

```
NOTHING                código especial para indicar un evento nulo.

DELETE                 el manejador de ventanas ha pedido que se oculte o ↩
    destruya la ventana de más alto nivel, normalmente cuando el usuario hace ↩
    clic en un icono especial de la barra de título.

DESTROY                la ventana ha sido destruida.

EXPOSE                 toda o parte de la ventana se ha hecho visible y necesita ↩
    redibujarse.

MOTION_NOTIFY          el puntero (normalmente el ratón) se ha movido.

BUTTON_PRESS           se ha presionado un botón del ratón.

_2BUTTON_PRESS         se ha hecho doble clic en un botón del ratón (2 veces ↩
    dentro de un corto periodo de tiempo). Nótese que cada clic genera también un ↩
     evento BUTTON_PRESS.

_3BUTTON_PRESS         se ha hecho clic 3 veces seguidas dentro de un corto ↩
    periodo de tiempo en un botón del ratón. Nótese que cada clic genera también ↩
    un evento BUTTON_PRESS.

BUTTON_RELEASE         se ha soltado un botón del ratón.

KEY_PRESS              se ha pulsado una tecla.

KEY_RELEASE            se ha soltado una tecla.

ENTER_NOTIFY           el puntero ha entrado en la ventana.

LEAVE_NOTIFY           el puntero ha salido de la ventana.

FOCUS_CHANGE           el foco del teclado ha entrado o dejado la ventana.

CONFIGURE              el tamaño, posición u orden de apilamiento de la ventana ↩
    ha cambiado. Obsérvese que GTK+ no usa estos eventos en ventanas hijas ( ↩
    GDK_WINDOW_CHILD).

MAP                    se han reservado recursos para la ventana.

UNMAP                  se han liberado recursos para la ventana.

PROPERTY_NOTIFY        se ha borrado o cambiado una propiedad de la ventana.

SELECTION_CLEAR        la aplicación ha perdido la propiedad de una selección.

SELECTION_REQUEST      otra aplicación ha solicitado una selección.

SELECTION_NOTIFY       se ha recibido una selección.
```

```
PROXIMITY_IN          se ha hecho contacto en una superficie sensible de un ↩
    dispositivo de entrada (por ejemplo, una tableta gráfica o pantalla sensible ↩
    al tacto).

PROXIMITY_OUT         se ha perdido el contacto de una superfice sensible.

DRAG_ENTER            el ratón ha entrado en la ventana mientras se estaba ↩
    arrastrando algo.

DRAG_LEAVE            el ratón ha salido de la ventana mientras se estaba ↩
    arrastrando algo.

DRAG_MOTION           el ratón se ha movido por la ventana mientras se estaba ↩
    arrastrando algo.

DRAG_STATUS           el estado de la operación de arrastre iniciada por la ↩
    ventana ha cambiado.

DROP_START            se ha iniciado una operación de soltar en la ventana.

DROP_FINISHED         la operación de soltar iniciada por la ventana ha ↩
    terminado.

CLIENT_EVENT          se ha recibido un mensaje desde otra aplicación.

VISIBILITY_NOTIFY     la visibilidad de la ventana ha cambiado.

NO_EXPOSE             indica que la región fuente estaba disponible ↩
    completamente cuando partes de un dibujable fueron copiadas. No es muy útil.

SCROLL                ?

WINDOW_STATE          ?

SETTING               ?
```

state especifica el modificador de estado cuando ocurre el evento (es decir, especifica que teclas auxiliares y botones del ratón estaban presionados). Es una combinación con el operador OR de algunas de las siguientes constantes (incluidas en el módulo gtk.gdk):

```
  SHIFT_MASK     # máscara de mayúsculas
  LOCK_MASK      # máscara de bloqueo
  CONTROL_MASK   # máscara de control
  MOD1_MASK      # máscara del modificador 1
  MOD2_MASK      # máscara del modificador 2
  MOD3_MASK      # máscara del modificador 3
  MOD4_MASK      # máscara del modificador 4
  MOD5_MASK      # máscara del modificador 5
  BUTTON1_MASK   # máscara del botón 1
  BUTTON2_MASK   # máscara del botón 2
  BUTTON3_MASK   # máscara del botón 3
  BUTTON4_MASK   # máscara del botón 4
  BUTTON5_MASK   # máscara del botón 5
```

Como con las demás señales, para determinar qué sucede cuando ocurre un evento se llama al método connect(). Pero también es necesario indicar a GTK+ qué eventos queremos tratar. Para ello llamamos al método:

```
  widget.set_events(events)
```

El argumento *events* especifica los eventos en los que se está interesado. Es una combinación con el operador OR de constantes que especifican los distintos tipos de eventos. Los tipos de eventos (del módulo gtk.gdk) son:

```
  EXPOSURE_MASK
```

```
  POINTER_MOTION_MASK
  POINTER_MOTION_HINT_MASK
  BUTTON_MOTION_MASK
  BUTTON1_MOTION_MASK
  BUTTON2_MOTION_MASK
  BUTTON3_MOTION_MASK
  BUTTON_PRESS_MASK
  BUTTON_RELEASE_MASK
  KEY_PRESS_MASK
  KEY_RELEASE_MASK
  ENTER_NOTIFY_MASK
  LEAVE_NOTIFY_MASK
  FOCUS_CHANGE_MASK
  STRUCTURE_MASK
  PROPERTY_CHANGE_MASK
  VISIBILITY_NOTIFY_MASK
  PROXIMITY_IN_MASK
  PROXIMITY_OUT_MASK
  SUBSTRUCTURE_MASK
```

Hay un par de cuestiones que es necesario tener en cuenta al llamar al método set_events() . En primer lugar, que debe ser llamado antes de que se cree la ventana X del control PyGTK (en la práctica esto significa que es preciso llamarlo inmediatamente después de crear el control). En segundo lugar, que el control debe tener una ventana X asociada. Por eficiencia, la mayoria de los controles no tienen su propia ventana sino que se dibujan en la ventana del control padre. Entre estos controles se incluyen:

```
  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
```

Para capturar eventos en estos controles se debe usar un control EventBox. Véase la sección del control EventBox para más detalles.

Los atributos de los eventos que PyGTK usa para cada tipo de evento son:

```
todos los eventos       type              # tipo
                        window            # ventana
                        send_event        # evento enviado

NOTHING
DELETE
DESTROY                                   # sin atributos adicionales

EXPOSE                  area              # área
                        count             # cuenta

MOTION_NOTIFY           time              # tiempo
                        x                 # x
                        y                 # y
                        pressure          # presión
                        xtilt             # inclinación x
```

```
                        ytilt               # inclinación y
                        state               # estado
                        is_hint             # es pista
                        source              # fuente
                        deviceid            # identificador de dispositivo
                        x_root              # x raíz
                        y_root              # y raíz

BUTTON_PRESS
_2BUTTON_PRESS
_3BUTTON_PRESS
BUTTON_RELEASE          time                # tiempo
                        x                   # x
                        y                   # y
                        pressure            # presión
                        xtilt               # inclinación x
                        ytilt               # inclinación y
                        state               # estado
                        button              # botón
                        source              # fuente
                        deviceid            # identificador de dispositivo
                        x_root              # x raíz
                        y_root              # y raíz

KEY_PRESS
KEY_RELEASE             time                # tiempo
                        state               # estado
                        keyval              # valor de tecla
                        string              # cadena de caracteres

ENTER_NOTIFY
LEAVE_NOTIFY            subwindow           # subventana
                        time                # tiempo
                        x                   # x
                        y                   # y
                        x_root              # x raíz
                        y_root              # y raíz
                        mode                # modo
                        detail              # detalle
                        focus               # foco
                        state               # estado

FOCUS_CHANGE            _in                 # dentro

CONFIGURE               x                   # x
                        y                   # y
                        width               # ancho
                        height              # alto

MAP
UNMAP                                       # sin atributos adicionales

PROPERTY_NOTIFY         atom                # átomo
                        time                # tiempo
                        state               # estado

SELECTION_CLEAR
SELECTION_REQUEST
SELECTION_NOTIFY        selection           # selección
                        target              # objetivo
                        property            # propiedad
                        requestor           # solicitante
                        time                # tiempo
```

```
PROXIMITY_IN
PROXIMITY_OUT          time                # tiempo
                       source              # fuente
                       deviceid            # identificador de dispositivo

DRAG_ENTER
DRAG_LEAVE
DRAG_MOTION
DRAG_STATUS
DROP_START
DROP_FINISHED          context             # contexto
                       time                # tiempo
                       x_root              # x raíz
                       y_root              # y raíz

CLIENT_EVENT           message_type        # tipo de mensaje
                       data_format         # formato de los datos
                       data                # datos

VISIBILTY_NOTIFY       state               # estado

NO_EXPOSE                                  # sin atributos adicionales
```

24.2.1. Scribble - Manejo de Eventos

En nuestro programa de dibujo querremos saber en qué momento se pulsa el botón del ratón y cuándo se mueve. Por tanto especificamos `POINTER_MOTION_MASK` y `BUTTON_PRESS_MASK`. También querremos saber cuándo hay que redibujar la ventana, y por ello especificamos `EXPOSURE_MASK`. Aunque deseamos que se nos notifique con un evento de configuración cuando el tamaño de la ventana cambie, no es necesario especificar el correspondiente `STRUCTURE_MASK`, puesto que se hace automáticamente para todas las ventanas.

Sin embargo, resulta problemático especificar solamente `POINTER_MOTION_MASK`, puesto que haría que el servidor añadiese un nuevo evento de moviento a la cola de eventos cada vez que el usuario mueve el ratón. Imaginemos que se tardan 0.1 segundos en tratar un evento de movimiento, pero el servidor X encola un nuevo evento de movimiento cada 0.05 segundos. Pronto nos encontraremos muy por detrás de lo que el usuario está dibujando. Si el usuario está dibujando durante 5 segundos, ¡se tardarían otros 5 segundos en recuperarse después de que suelte el botón del ratón!. Lo que se puede hacer es tratar un evento de movimiento por cada evento que procesemos. Para hacer esto debemos especificar `POINTER_MOTION_HINT_MASK`.

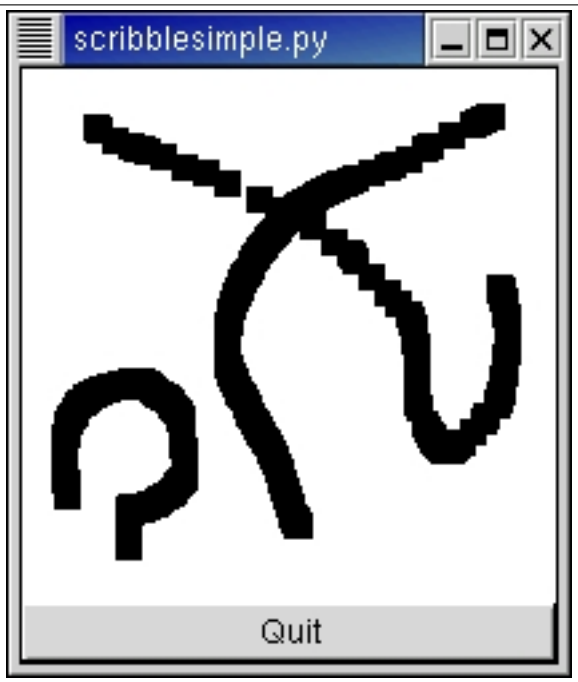
Cuando se especifica `POINTER_MOTION_HINT_MASK`, el servidor manda un evento de movimiento la primera vez que el puntero se mueve tras entrar en la ventana o después de que se pulse o suelte el botón. Los movimientos posteriores se suprimen hasta que se solicite explícitamente la posición del puntero con el siguiente método de `gtk.gdk.Window`:

```
x, y, mask = window.get_pointer()
```

window es un objeto `gtk.gdk.Window`. *x* e *y* son las coordenadas del puntero y *mask* es la máscara de modificación para detectar qué teclas están pulsadas. (Hay un método de `gtk.Widget`, get_pointer(), que devuelve la misma información que el método gtk.gdk.Window.get_pointer() pero sin la información de máscara).

El programa de ejemplo scribblesimple.py demuestra el uso básico de eventos y manejadores de eventos. La figura Figura 24.2 muestra el programa en acción:

Figura 24.2 Ejemplo sencillo - Scribble



Los manejadores de eventos se conectan a la drawing_area (área de dibujo) en las siguientes líneas:

```
92        # Señales para gestionar el mapa de píxeles de respaldo
93        drawing_area.connect("expose_event", expose_event)
94        drawing_area.connect("configure_event", configure_event)
95
96        # señales de evento
97        drawing_area.connect("motion_notify_event", motion_notify_event)
98        drawing_area.connect("button_press_event", button_press_event)
99
100       drawing_area.set_events(gtk.gdk.EXPOSURE_MASK
101                               | gtk.gdk.LEAVE_NOTIFY_MASK
102                               | gtk.gdk.BUTTON_PRESS_MASK
103                               | gtk.gdk.POINTER_MOTION_MASK
104                               | gtk.gdk.POINTER_MOTION_HINT_MASK)
```

Los manejadores de eventos `button_press_event()` y `motion_notify_event()` en scribblesimple.py son:

```
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 gtk.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 gtk.TRUE
```

Los manejadores `expose_event()` y `configure_event()` se describirán más adelante.

24.3. El Control del Área de Dibujo, y Dibujar

Ahora toca dibujar en la pantalla. El control que se usa para éso es la DrawingArea (Área de dibujo). Un control de área de dibujo es básicamente una ventana X; sin más. Es un lienzo en blanco en el que se puede pintar lo que se desee. Un área de dibujo se crea usando la llamada:

```
darea = gtk.DrawingArea()
```

Se puede especificar un tamaño predeterminado para el control usando:

```
darea.set_size_request(width, height)
```

Este tamaño predeterminado se puede cambiar, como en todos los controles, llamando al método set_size_request(), y también se puede modificar si el usuario cambia manualmente el tamaño de la ventana que contiene el área de dibujo.

Hay que aclarar que, cuando creamos un control DrawingArea, se adquiere toda la responsabilidad de dibujar su contenido. Si la ventana se tapa y luego se muestra, se recibe un evento de exposición y se debe redibujar lo que antes se ocultó.

Tener que recordar lo que había dibujado en la pantalla para que podamos redibujarlo es, cuanto menos, una inconveniencia. Además, puede resultar molesto visualmente el que partes de la ventana se limpien para luego dibujarse paso a paso. La solución a este problema es el uso de un pixmap oculto. En vez de dibujar directamente en la pantalla se dibuja en la imagen almacenada en la memoria del servidor (que no se muestra), y luego se copian las partes relevantes a la pantalla.

Para crear un pixmap fuera de pantalla, se usa esta función:

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

El parámetro *window* especifica una ventana `gtk.gdk.Window` de la que este *pixmap* tomará algunas propiedades. *width* (ancho) y *height* (alto) especifican el tamaño del *pixmap*. *depth* especifica la profundidad de color, es decir, el número de bits por píxel de la nueva ventana. Si *depth* es -1 o se omite, coincidirá con la profundidad de la ventana.

Creamos el pixmap en nuestro manejador del evento "configure_event". Este evento se genera cada vez que la ventana cambia de tamaño, incluso cuando se crea por primera vez.

```
32   # Creamos un nuevo mapa de píxeles de respaldo con el tamaño adecuado
33   def configure_event(widget, event):
34       global pixmap
35
36       x, y, width, height = widget.get_allocation()
37       pixmap = gtk.gdk.Pixmap(widget.get_window(), width, height)
38       gtk.draw_rectangle(pixmap, widget.get_style().white_gc,
39                          gtk.TRUE, 0, 0, width, height)
40
41       return gtk.TRUE
```

La llamada a draw_rectangle() deja inicialmente el pixmap en blanco. Se comentará esto un poco más detenidamente en breve.

El manejador del evento de exposición simplemente copia las partes correspondientes del pixmap en el área de dibujo usando el método draw_pixmap(). (Se determina el área que se debe redibujar mediante el atributo event.area del evento de exposición):

```
43   # Redibujamos la pantalla a partir del pixmap de respaldo
44   def expose_event(widget, event):
45       x , y, width, height = event.area
46       widget.window.draw_drawable(widget.get_style().fg_gc[gtk.STATE_NORMAL ↩
    ],
47                                   pixmap, x, y, x, y, width, height)
48       return gtk.FALSE
```

Ya se ha visto cómo manetener la pantalla sincronizada con el pixmap de respaldo, pero ¿cómo se pintan cosas interesantes en el pixmap? Existe un gran número de llamadas de PyGTK para dibujar en objetos dibujables. Un objeto dibujable es sencillamente algo en lo que se puede dibujar. Puede ser una ventana, un pixmap, o un bitmap (imagen en blanco y negro). Ya se han visto dos de esas llamadas con anterioridad, draw_rectangle() y draw_pixmap(). La lista completa es:

```
# dibuja punto
drawable.draw_point(gc, x, y)

# dibuja línea
drawable.draw_line(gc, x1, y1, x2, y2)

# dibuja rectángulo
drawable.draw_rectangle(gc, fill, x, y, width, height)

# dibuja arco
drawable.draw_arc(gc, fill, x, y, width, height, angle1, angle2)

# dibuja polígono
drawable.draw_polygon(gc, fill, points)

# dibuja dibujable
drawable.draw_drawable(gc, src, xsrc, ysrc, xdest, ydest, width, height)

# dibuja puntos
drawable.draw_points(gc, points)

# dibuja líneas
drawable.draw_lines(gc, points)

# dibuja segmentos
drawable.draw_segments(gc, segments)

# dibuja imagen rgb
drawable.draw_rgb_image(gc, x, y, width, height, dither, buffer, rowstride)

# dibuja imagen rgb 32 bits
drawable.draw_rgb_32_image(gc, x, y, width, height, dither, buffer, rowstride)

# dibuja imagen escala grises
drawable.draw_gray_image(gc, x, y, width, height, dither, buffer, rowstride)
```

Los métodos del área de dibujo son los mismos que los métodos de dibujo de los objetos dibujables por lo que se puede consultar la sección Métodos de Dibujo para más detalles sobre estas funciones. Todas estas funciones comparten los primeros argumentos. El primer argumento es un contexto gráfico (*gc*).

Un contexto gráfico encapsula información sobre cuestiones como los colores de fondo y frente o el ancho de línea. PyGTK posee un conjunto completo de funciones para crear y modificar contextos gráficos, pero para mantener las cosas fáciles aquí simplemente se usarán contextos gráficos predefinidos. Consúltese la sección Contexto Gráfico del Área de Dibujo para obtener más información sobre los contextos gráficos. Cada control tiene un estilo asociado (que se puede modificar en un fichero `gtkrc`, según se indica en la sección Ficheros rc de GTK+.) Éste, entre otras cosas, almacena diversos contextos gráficos. Algunos ejemplos del acceso a estos contextos gráficos son:

```
widget.get_style().white_gc

widget.get_style().black_gc

widget.get_style().fg_gc[STATE_NORMAL]

widget.get_style().bg_gc[STATE_PRELIGHT]
```

Los campos *fg_gc*, *bg_gc*, *dark_gc*, y *light_gc* se indexan con un parámetro que puede tomar los siguientes valores:

```
STATE_NORMAL,       # El estado durante la operación normal
STATE_ACTIVE,       # El control está activado, como cuando se pulsa un botón
STATE_PRELIGHT,     # El puntero del ratón está sobre el control
STATE_SELECTED,     # El control está seleccionado
```

```
STATE_INSENSITIVE  # El control está desactivado
```

Por ejemplo, para `STATE_SELECTED` el color de frente predeterminado es el blanco y el color de fondo predeterminado es azul oscuro.

La función draw_brush() (pinta trazo) del ejemplo, que es la que realmente dibuja en el pixmap, es la siguiente:

```
50   # Dibujamos un rectángulo en la pantalla
51   def draw_brush(widget, x, y):
52       rect = (int(x - 5), int(y - 5), 10, 10)
53       pixmap.draw_rectangle(widget.get_style().black_gc, gtk.TRUE,
54                             rect[0], rect[1], rect[2], rect[3])
55       apply(widget.queue_draw_area, rect)
```

Tras dibujar el rectángulo que representa el trazo en el pixmap llamamos a la función:

```
widget.queue_draw_area(x, y, width, height)
```

que notifica a X que ese área ha de ser actualizada. En algún momento X generará un evento de exposición (posiblemente combinando las áreas que se le pasan en varias llamadas a draw()) que hará que se llame al manejador del evento de exposición que se creó anteriormente que copiará las partes relevantes en la pantalla.

Y... ya se ha tratado todo el programa de dibujo, exceptuando algunos detalles mundanos como la creación de la ventana principal.

Capítulo 25

Trucos para Escribir Aplicaciones PyGTK

Esta sección es simplemente una recolección de conocimientos, guías de estilo generales y trucos para crear buenas aplicaciones PyGTK. Actualmente esta sección es muy corta, pero espero que se vaya haciendo más larga en futuras ediciones de este tutorial.

25.1. El usario debería manejar la interfaz, no al contrario

PyGTK, como otros conjuntos de herramientas, te proporciona maneras de invocar widgets, tales como la bandera `DIALOG_MODAL` que se pasa a los dialogos, que requiere una respuesta desde el usuario antes de que el resto de la aplicación continúe. En Python, como en otros lenguajes, es una buena práctica evitar el uso de elementos de interfaz modales.

En cada interacción modal la aplicación fuerza un flujo particular en el usuario y, si bien esto es en algún momento inevitable, como regla general debe evitarse, ya que la aplicación debe tratar de adaptarse al flujo de trabajo preferido por el usuario.

Un caso particularmente frecuente de esto es la solicitud de confirmación. Cada una de ellas debería correponder a un punto en el que se debería porporcionar la posibilidad de deshacer los cambios. GIMP, la aplicación para la que se desarrolló inicialmente GTk+, evita muchas operaciones que necesitarían detenerse y solicitar confirmación del usuario mediante una instrucción de deshacer que permite deshacer cualquier operación que ejecuta.

25.2. Separa el modelo de datos de la interfaz

El sistema de objetos flexible e implícito de Python reduce el coste de la toma de decisiones sobre la arquitectura de la aplicación en comparación con otros lenguajes más rígidos (sí, *estamos* pensando en C++). Una de ellas es la cuidadosa separación del modelo de datos (las clases y estructuras de datos que representan el estado para los que la aplicación está diseñada para manejar) del controlador (las clases que implementan la interfaz de usuario).

En Python, un patrón de uso habitual es el de tener una clase principal editora/controladora que encapsula la interfaz de usuario (con, probablemente, pequeñas clases para controles que mantengan su estado) y una clase de modelo principal que encapsula el estado de la aplicación (probablemente con algunos miembros que son en sí mismos instancias de pequeñas clases para representación de datos). La clase controladora (controlador) llama a métodos de la clase modelo (el modelo) para realizar su manipulación de datos. A su vez, el modelo delega la representación en pantalla y el procesado de entrada de datos al controlador.

Reducir la interfaz entre el modelo y el controlador facilita evitar quedar atrapado en decisiones iniciales sobre la relación entre cualquiera de ambas partes. También hace más sencillo el mantenimiento de la aplicación y el diagnóstico de fallos.

25.3. Cómo separar los Métodos de Retrollamada de los Manejadores de Señal

25.3.1. Introducción

No es necesario almacenar todas las retrollamadas en un único archivo de programa. Se pueden separar en clases independientes, en archivos separados. De esta forma la aplicación principal puede derivar sus métodos de esas clases mediante herencia. Al final se acaba con toda la funcionalidad inicial con el beneficio de un mantenimiento más sencillo, reusabilidad del código y menores tamaños de archivo, lo que significa menor carga para los editores de texto.

25.3.2. Herencia

La herencia es una forma de reutilizar el código. Una clase puede heredar su funcionalidad de otras clases, y un aspecto estupendo de la herencia es que podemos usarla para dividir un programa enorme en grupos lógicos de partes menores, más fáciles de mantener.

Ahora, dediquemos un momento a la terminología. Una clase derivada, también denominada subclase o clase hija, es una clase que deriva parte de su funcionalidad de otras clases. Una clase base, también llamada superclase o clase madre, es de dónde hereda una clase derivada.

Abajo se muestra un pequeño ejemplo para familiarizarse con la herencia. Es una buena idea probar esto en el intérprete de Python para ganar experiencia de primera mano.

Creamos dos clases base:

```
class base1:
   base1_attribute = 1
   def base1_method(self):
     return "hola desde la clase base 1"

class base2:
   base2_attribute = 2
   def base2_method(self):
     return "hola desde la clase base 2"
```

Luego creamos una clase derivada que hereda de esas dos clases base:

```
class derived(base1, base2):  # clase derivada de dos clases base
   var3 = 3
```

Ahora la clase derivada tiene toda la funcionalidad de las clases base.

```
x = derived()         # crea una instancia de la clase derivada
x.base1_attribute     # 1
x.base2_attribute     # 2
x.var3                # 3
x.base1_method()      # hola desde la clase base 1
x.base2_method()      # hola desde la clase base 2
```

El objeto llamado x tiene la habilidad de acceder a las variables y métodos de las clases base porque ha heredado su funcionalidad. Ahora apliquemos este concepto a una aplicación de PyGTK.

25.3.3. Herencia aplicada a PyGTK

Crea un archivo llamado gui.py, y luego copia este código en él:

```
#Un archivo llamado: gui.py

import pygtk
import gtk

# Crea una definición de clase llamada gui
class gui:
  #
  #          MÉTODOS DE RETROLLAMADA
```

```
  #-----------------------------------
  def open(self, widget):
    print "abre cosas"
  def save(self, widget):
    print "guarda cosas"
  def undo(self, widget):
    print "deshace cosas"
  def destroy(self, widget):
    gtk.main_quit()


  def __init__(self):
    #
    #        CÓDIGO DE CONSTRUCCIÓN DE GUI
    #----------------------------------------------
    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="Abre cosas")
    open_button.set_use_stock(True)
    self.vbox1.pack_start(open_button, False, False, 0)
    open_button.show()
    save_button = gtk.Button(label="Guarda Cosas")
    self.vbox1.pack_start(save_button, False, False, 0)
    save_button.show()
    undo_button = gtk.Button(label="Deshacer")
    self.vbox1.pack_start(undo_button, False, False, 0)
    undo_button.show()
    self.vbox1.show()
    #
    #        MANEJADORES DE SEÑAL
    #----------------------------------------------
    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()       # crea un objeto gui
 gui_instance.main()        # llama al método principal
```

Si se ejecuta el programa se comprueba que se trata de una única ventana con algunos botones. Tal como está organizado ahora el programa, todo el código está en un único archivo. Pero en un momento se verá cómo dividir el programa en múltiples archivos. La idea es retirar esos cuatro métodos de retrollamada de la clase gui y ponerlos en sus propias clases, en archivos independientes. En el caso de que se tuviesen cientos de métodos de retrollamada se procuraría agruparlos de alguna forma lógica como, por ejemplo, situando todos los métodos que se dedican a entrada/salida en la misma clase, y de esa manera se harían otras clases para grupos de métodos.

Lo primero que tenemos que hacer es construir algunas clases para los métodos del archivo gui.py. Creamos tres nuevos archivos de texto, llamados io.py, undo.py y destroy.py, y sitúa esos archivos en el mismo subdirectorio que el archivo gui.py. Copia el código siguiente en el archivo io.py:

```
class io:
  def open(self, widget):
    print "abre cosas"

  def save(self, widget):
    print "guarda cosas"
```

Estos son los dos métodos de retrollamada, open y save, del programa gui.py. Copia el siguiente bloque de código en el archivo undo.py:

```
class undo:
  def undo(self, widget):
    print "deshace cosas"
```

Este es el método undo de gui.py. Y, finalmente, copia el siguiente código en destroy.py:

```
import gtk

class destroy:
  def destroy(self, widget):
    gtk.main_quit()
```

Ahora todos los métodos están separados en clases independientes.

IMPORTANTE

En tus futuras aplicaciones querrás importar módulos como gtk, pango, os ect... en tu clase derivada (la que contiene todo el código de inicialización de GUI), pero recuerda que necesitarás importar también algunos de esos módulos en las clases base. Podrías tener que crear una instancia de un control de gtk en el método de una clase base, y en ese caso necesitarías importar gtk.

Este es solamente un ejemplo de una clase base en la que sería necesario importar gtk.

```
import gtk

class Font_io
  def Font_Chooser(self,widget):
    self.fontchooser = gtk.FontSelectionDialog("Elige Fuente ↩
        ")
    self.fontchooser.show()
```

Hay que observar que define un control gtk: un diálogo de selección de fuente. Normalmente se importaría gtk en la clase principal (la clase derivada) y todo funcionaría correctamente. Pero desde el momento que se extrae este Fon_Chooser de la clase principal y se pone en su propia clase y luego se intenta heredar de ella, nos encontraríamos con un error. En este aso, ni siquiera se detectaría el error hasta el momento de ejecutar la aplicación. Pero cuando se intente usar el Font_Chooser se encontraría que gtk no está definido, aunque se haya importado en la clase derivada. Así que se ha de recordar que, cuando se crean clases base, es necesario añadir sus propios import.

Con esas tres clases en sus tres archivos py, es necesario cambiar el código en gui.py de tres modos:

1. Importar las clases que se han creado.
2. Modificar la definición de la clase.
3. Eliminar los métodos de retrollamada.

El siguiente código actualizado muestra cómo hacerlo:

```
#Un archivo llamado:  gui.py
#(versión actualizada)
#(con herencia múltiple)

import pygtk
import gtk
```

```
from io import file_io                          #
from undo import undo                           # 1. Importa tus clases
from destroy import destroy                     #

# Crea una definición de clase llamada gui
class gui(io, undo, destroy):                   # 2. Definición de clase modificada
                                                # 3. Retrollamadas eliminadas
  def __init__(self):
    #
    #        CÓDIGO DE CONSTRUCCIÓN DE GUI
    #----------------------------------------------
    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()
    #
    #        MANEJADORES DE SEÑAL
    #----------------------------------------------
    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()       # crea un objeto gui
 gui_instance.main()        # llamada al método principal
```

Estas tres líneas son nuevas:

```
from io import io
from undo import undo
from destroy import destroy
```

Las instrucciones import tienen la forma:

```
from [nombre de archivo de la clase] import [nombre de la clase]
```

Este es el cambio de la definición de la clase:

```
class gui(io, undo, destroy):
```

Los nombres de las clases base van entre paréntesis en la definición de la clase. Ahora la clase gui es una clase derivada y es capaz de usar todos los atributos y métodos definidos en sus clases base. También, la case gui hereda de múltiples clases (dos o más); es lo que se conoce como heréncia múltiple.

Ahora cambia el archivo gui.py a la versión actualizada y ejecuta la aplicación de nuevo. Verás que funciona exactamente del mismo modo, salvo que ahora todos los métodos de retrollamada están en clases independients, y son heredadas por la clase gui.

Hay otro tema de interés que comentar. Mientras que tu aplicación gui.py y tus archivos de clases base estén en el mismo directorio todo funcionará correctamente. Pero si quieres crear otro directorio en el que situar los archivos con las clases base, entonces es necesario añadir dos líneas más de código junto al resto de intrucciones import en gui.py. Así:

```
import sys
sys.path.append("classes")
```

en donde "classes" es el nombre del directiro en el que se guardan las clases base. Esto permite que Python sepa dónde localizarlas. Pruébalo. Simplemente crea un directorio llamado classes en el directorio en el que está el programa gui.py. Luego añade los archivos de las tres clases base al directorio classes. Añade las dos líneas de código anteriores a la parte superior del archivo gui.py. Y, ¡listo!

Una nota final para quienes usan py2exe para compilar aplicaciones Python. Si se ponen las cases base en el directorio de Python, entonces py2exe los incluirá en la versión compilada de la aplicación como con cualquier otro módulo Python.

Capítulo 26

Contribuir

Este documento, como muchos programas que se encuentran por ahí, fue escrito gratuitamente por voluntarias. Si encuentra algún aspecto de PyGTK que no haya sido recogido en esta documentación, por favor considere la contribución al mismo.

Si se decide a contribuir, envíe por favor, mediante correo electrónico, su texto en inglés a John Finlay (finlay@moeraki.com) o, si es en español, a Lorenzo Gil Sánchez (lgs@cvs.gnome.org). Asimismo, tenga en cuenta que la totalidad de este documento es libre y gratuita, y cualquier contribución que se aporte debe ser en esas mismas condiciones de libertad y gratuidad. De ese modo cualquiera podrá tanto utilizar libremente fragmentos del tutorial en sus programas como distribuir libremente copias de este documento...

Muchas gracias.

Capítulo 27

Créditos

27.1. Créditos Original de GTK+

Los siguientes son los créditos originales de los Tutoriales GTK+ 1.2 y GTK+ 2.0 (de los que este tutorial es prácticamente copia literal):

- Bawer Dagdeviren, chamele0n@geocities.com por el tutorial de menús.
- Raph Levien, raph@acm.org por el "hello world" a GTK, empaquetamiento de controles, y diversa sabiduría general. Él es también un hogar para este tutorial.
- Peter Mattis, petm@xcf.berkeley.edu por el programa GTK más simple... y la habilidad para hacerlo :)
- Werner Koch werner.koch@guug.de por convertir el texto plano original a SGML, y la jerarquía de clases de controles.
- Mark Crichton crichton@expert.cc.purdue.edu por el código de factoría de menús, y el tutorial de empaquetamiento en tablas.
- Owen Taylor owt1@cornell.edu por la sección del control EventBox (y el parche para la distribución). Él también es el responsable del tutorial y código de selecciones, así como de las secciones escribir tu propio control GTK, y la aplicación de ejemplo. ¡Un montón de gracias por toda tu ayuda Owen!
- Mark VanderBoom mvboom42@calvin.edu por su maravilloso trabajo en los controles Notebook, Progress Bar, Dialogs y Selección de Archivos. ¡Muchas gracias Mark! Has sido de gran ayuda.
- Tim Janik timj@gtk.org por su gran trabajo en el control Lists. Su excelente trabajo en la extracción automática de información de las señales y del control Tree de GTK. Gracias Tim :)
- Rajat Datta rajat@ix.netcom.com por su excelente trabajo en el tutorial de Pixmap.
- Michael K. Johnson johnsonm@redhat.com por información y código de los menús popup.
- David Huggins-Daines bn711@freenet.carleton.ca por las secciones de los controles Range y Tree.
- Stefan Mars mars@lysator.liu.se por la sección de CList.
- David A. Wheeler dwheeler@ida.org por parte del texto GLib y varias correcciones y mejoras del tutorial. El texto GLib está a su vez basado en material desarrollado por Damon Chaplin DAChaplin@msn.com
- David King por la comprobación de estilo del documento completo.

Y a todos vosotros que habéis comentado y ayudado a refinar este documento.
Gracias.

Capítulo 28

Copyright del Tutorial y Nota de Permisos

Apéndice A

Señales de GTK

Como PyGTk es un conjunto de controles orientado a objetos, tiene una jerarquía de herencia. Este mecanismo de herencia se aplica a las señales. Por tanto, se debe utilizar la jerarquía de controles cuando se usen las señales listadas en esta sección.

A.1. gtk.Object

```
  destroy(object, data)
```

A.2. gtk.Widget

```
show(GtkWidget, data)

hide(widget, data)

map(widget, data)

unmap(widget, data)

realize(widget, data)

unrealize(widget, data)

draw(widget, area, data)

draw-focus(widget, data)

draw-default(widget, data)

size-request(widget, requisition, data)

size-allocate(widget, allocation, data)

state-changed(widget, state, data)

parent-set(widget, object, data)

style-set(widget, style, data)

add-accelerator(widget, accel_signal_id, accel_group, accel_key, accel_mods,
                accel_flags, data)

remove-accelerator(widget, accel_group, accel_key, accel_mods, data)

bool = event(widget, event, data)
```

```
bool = button-press-event(widget, event, data)

bool = button-release-event(widget, event, data)

bool = motion-notify-event(widget, event, data)

bool = delete-event(widget, event, data)

bool = destroy-event(widget, event, data)

bool = expose-event(widget, event, data)

bool = key-press-event(widget, event, data)

bool = key-release-event(widget, event, data)

bool = enter-notify-event(widget, event, data)

bool = leave-notify-event(widget, event, data)

bool = configure-event(widget, event, data)

bool = focus-in-event(widget, event, data)

bool = focus-out-event(widget, event, data)

bool = map-event(widget, event, data)

bool = unmap-event(widget, event, data)

bool = property-notify-event(widget, event, data)

bool = selection-clear-event(widget, event, data)

bool = selection-request-event(widget, event, data)

bool = selection-notify-event(widget, event, data)

selection-get(widget, selection_data, info, time, data)

selection-received(widget, selection_data, time, data)

bool = proximity-in-event(widget, event, data)

bool = proximity-out-event(widget, event, data)

drag-begin(widget, context, data)

drag-end(widget, context, data)

drag-data-delete(widget, context, data)

drag-leave(widget, context, time, data)

bool = drag-motion(widget, context, x, y, time, data)

bool = drag-drop(widget, context, x, y, time, data)

drag-data-get(widget, context, selection_data, info, time, data)

drag-data-received(widget, context, info, time, selection_data,
                   info, time, data)
```

```
bool = client-event(widget, event, data)

bool = no-expose-event(widget, event, data)

bool = visibility-notify-event(widget, event, data)

debug-msg(widget, string, data)
```

A.3. GtkData

```
  disconnect(data_obj, data)
```

A.4. gtk.Container

```
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. gtk.Calendar

```
month-changed(calendar, data)

day-selected(calendar, data)

day-selected-double-click(calendar, data)

prev-month(calendar, data)

next-month(calendar, data)

prev-year(calendar, data)

next-year(calendar, data)
```

A.6. gtk.Editable

```
changed(editable, data)

insert-text(editable, new_text, text_length, position, data)

delete-text(editable, start_pos, end_pos, data)

activate(editable, data)

set-editable(editable, is_editable, data)

move-cursor(editable, x, y, data)
```

```
move-word(editable, num_words, data)

move-page(editable, x, y, data)

move-to-row(editable, row, data)

move-to-column(editable, column, data)

kill-char(editable, direction, data)

kill-word(editable, drirection, data)

kill-line(editable, direction, data)

cut-clipboard(editable, data)

copy-clipboard(editable, data)

paste-clipboard(editable, data)
```

A.7. gtk.Notebook

```
  switch-page(noteboook, page, page_num, data)
```

A.8. gtk.List

```
selection-changed(list, data)

select-child(list, widget, data)

unselect-child(list, widget, data)
```

A.9. gtk.MenuShell

```
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. gtk.Toolbar

```
orientation-changed(toolbar, orientation, data)

style-changed(toolbar, toolbar_style, data)
```

A.11. gtk.Button

```
pressed(button, data)

released(button, data)

clicked(button, data)

enter(button, data)

leave(button, data)
```

A.12. gtk.Item

```
select(item, data)

deselect(item, data)

toggle(item, data)
```

A.13. gtk.Window

```
set-focus(window, widget, data)
```

A.14. gtk.HandleBox

```
child-attached(handle_box, widget, data)

child-detached(handle_box, widget, data)
```

A.15. gtk.ToggleButton

```
toggled(toggle_button, data)
```

A.16. gtk.MenuItem

```
activate(menu_item, data)

activate-item(menu_item, data)
```

A.17. gtk.CheckMenuItem

```
toggled(check_menu_item, data)
```

A.18. gtk.InputDialog

```
enable-device(input_dialog, deviceid, data)

disable-device(input_dialog, deviceid, data)
```

A.19. gtk.ColorSelection

```
color-changed(color_selection, data)
```

A.20. gtk.StatusBar

```
text-pushed(statusbar, context_id, text, data)

text-popped(statusbar, context_id, text, data)
```

A.21. gtk.Curve

```
curve-type-changed(curve, data)
```

A.22. gtk.Adjustment

```
changed(adjustment, data)

value-changed(adjustment, data)
```

Apéndice B

Ejemplos de Código

B.1. scribblesimple.py

```
 1  #!/usr/bin/env python
 2
 3  # example scribblesimple.py
 4
 5 # GTK - The GIMP Toolkit
 6 # Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh MacDonald
 7 # Copyright (C) 2001-2002 John Finlay
 8 #
 9 # This library is free software; you can redistribute it and/or
10 # modify it under the terms of the GNU Library General Public
11 # License as published by the Free Software Foundation; either
12 # version 2 of the License, or (at your option) any later version.
13 #
14 # This library is distributed in the hope that it will be useful,
15 # but WITHOUT ANY WARRANTY; without even the implied warranty of
16 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
17 # Library General Public License for more details.
18 #
19 # You should have received a copy of the GNU Library General Public
20 # License along with this library; if not, write to the
21 # Free Software Foundation, Inc., 59 Temple Place - Suite 330,
22 # Boston, MA 02111-1307, USA.
23
24
25 import gtk
26
27 # Backing pixmap for drawing area
28 pixmap = None
29
30 # Create a new backing pixmap of the appropriate size
31 def configure_event(widget, event):
32     global pixmap
33
34     x, y, width, height = widget.get_allocation()
35     pixmap = gtk.gdk.Pixmap(widget.window, width, height)
36     pixmap.draw_rectangle(widget.get_style().white_gc,
37                           gtk.TRUE, 0, 0, width, height)
38
39     return gtk.TRUE
40
41 # Redraw the screen from the backing pixmap
42 def expose_event(widget, event):
43     x , y, width, height = event.area
44     widget.window.draw_drawable(widget.get_style().fg_gc[gtk.STATE_NORMAL],
45                                 pixmap, x, y, x, y, width, height)
```

```
 46     return gtk.FALSE
 47
 48 # Draw a rectangle on the screen
 49 def draw_brush(widget, x, y):
 50     rect = (x - 5, y - 5, 10, 10)
 51     pixmap.draw_rectangle(widget.get_style().black_gc, gtk.TRUE,
 52                           rect[0], rect[1], rect[2], rect[3])
 53     widget.queue_draw_area(rect[0], rect[1], rect[2], rect[3])
 54
 55 def button_press_event(widget, event):
 56     if event.button == 1 and pixmap != None:
 57         draw_brush(widget, event.x, event.y)
 58     return gtk.TRUE
 59
 60 def motion_notify_event(widget, event):
 61     if event.is_hint:
 62         x, y, state = event.window.get_pointer()
 63     else:
 64         x = event.x
 65         y = event.y
 66         state = event.state
 67
 68     if state & gtk.gdk.BUTTON1_MASK and pixmap != None:
 69         draw_brush(widget, x, y)
 70
 71     return gtk.TRUE
 72
 73 def main():
 74     window = gtk.Window(gtk.WINDOW_TOPLEVEL)
 75     window.set_name ("Test Input")
 76
 77     vbox = gtk.VBox(gtk.FALSE, 0)
 78     window.add(vbox)
 79     vbox.show()
 80
 81     window.connect("destroy", gtk.mainquit)
 82
 83     # Create the drawing area
 84     drawing_area = gtk.DrawingArea()
 85     drawing_area.set_size_request(200, 200)
 86     vbox.pack_start(drawing_area, gtk.TRUE, gtk.TRUE, 0)
 87
 88     drawing_area.show()
 89
 90     # Signals used to handle backing pixmap
 91     drawing_area.connect("expose_event", expose_event)
 92     drawing_area.connect("configure_event", configure_event)
 93
 94     # Event signals
 95     drawing_area.connect("motion_notify_event", motion_notify_event)
 96     drawing_area.connect("button_press_event", button_press_event)
 97
 98     drawing_area.set_events(gtk.gdk.EXPOSURE_MASK
 99                             | gtk.gdk.LEAVE_NOTIFY_MASK
100                             | gtk.gdk.BUTTON_PRESS_MASK
101                             | gtk.gdk.POINTER_MOTION_MASK
102                             | gtk.gdk.POINTER_MOTION_HINT_MASK)
103
104     # .. And a quit button
105     button = gtk.Button("Quit")
106     vbox.pack_start(button, gtk.FALSE, gtk.FALSE, 0)
107
108     button.connect_object("clicked", lambda w: w.destroy(), window)
109     button.show()
```

```
110
111     window.show()
112
113     gtk.main()
114
115     return 0
116
117 if __name__ == "__main__":
118     main()
```

Apéndice C

ChangeLog

```
2004-12-22    <lgs@sicem.biz>

        * pygtk2-tut.xml: cambiada la version a 2.3pre1

2004-12-21    <lgs@sicem.biz>

        * Version 2.2

2004-12-17    <lgs@sicem.biz>

        * pygtk2-tut.xml: MovingOn.xml, PackingWidgets.xml
        revisados por pachi

2004-12-09    <lgs@sicem.biz>

        * pygtk2-tut.xml: DragAndDrop.xml, ManagingSelections.xml y
        TimeoutsIOAndIdleFunctions.xml revisados por pachi

2004-12-03    <lgs@sicem.biz>

        * AdvancedEventAndSignalHandling.xml: revisado por pachi

2004-12-02    <lgs@sicem.biz>

        * pygtk2-tut.xml: revisado ExpanderWidget.xml, Scribble.xml,
        GtkRcFiles.xml y Contributing.xml por pachi

2004-11-23    <lgs@sicem.biz>

        * ColorButtonAndFontButton.xml: cambiado el encoding en el prologo
        xml

        * NewInPyGTK24.xml (linkend): enlazado ColorButtonAndFontButton.xml

        * pygtk2-tut.xml: revisado SettingWidgetAttributes.xml,
        TimeoutsIOAndIdleFuncionts.xml, UndocumentedWidgets.xml,
        AdvancedEventAndSignalHandling.xml, GtkSignals.xml

2004-11-20    <lgs@sicem.biz>

        * pygtk2-tut.xml: revisado Introduction.xml, MovingOn.xml,
        PackingWidgets.xml

2004-11-04    <lgs@sicem.biz>
```

```
        * pygtk2-tut.xml: GettingStarted.xml revisado por pachi

2004-11-03    <lgs@sicem.biz>

        * pygtk2-tut.xml: NewInPyGTK24.xml, ComboBoxAndComboBoxEntry.xml,
        EntryCompletion.xml, ExpanderWidget.xml, FileChooser.xml,
        GenericCellRenderer.xml y GenericTreeModel.xml por pachi

2004-11-02    <lgs@sicem.biz>

        * pygtk2-tut.xml: NewWidgetsAndObjects.xml, CellRenderers.xml,
        TreeModel.xml por pachi

2004-10-25    <lgs@sicem.biz>

        * pygtk2-tut.xml: revisado ContainerWidgets.xml, CodeExamples.xml,
        ButtonWidget.xml, RangeWidgets.xml, MiscellaneousWidgets.xml y
        NewInPyGTK24.xml por pachi

2004-10-21    <lgs@sicem.biz>

        * TextViewWidget.xml (linkend): pequenio fallo al no cerrar la marca
        <literal> en la linea 1501

        * pygtk2-tut.xml: WidgetOverview.xml y Adjustments.xml revisados por
        pachi.

        * ActionsAndActionGroups.xml:
        * CellRendereres.xml:
        * ComboBoxAndComboBoxEntry.xml:
        * ExpanderWidget.xml:
        * FileChooser.xml:
        * GenericCellRenderer.xml:
        * GenericTreeModel.xml:
        * NewInPyGTK24.xml:
        * NewWidgetsAndObjects.xml:
        * TreeModel.xml: set the encoding to iso-8859-1

2004-10-19    <lgs@sicem.biz>

        * pygtk2-tut.xml: le asigno a pachi NewWidgetsAndObjects.xml

2004-10-18    <lgs@sicem.biz>

        * pygtk2-tut.xml: pachi ha revisado TextViewWidget.xml y MenuWidget.xml

2004-10-08  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: pachi ha revisado GtkSignals.xml y parece que esta
        bien

2004-10-04  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * ActionsAndActionGroups.xml: added (submited by pachi)

2004-09-21  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: quito la marca de revision
```

```
        * TextViewWidget.xml: revisado por pachi

2004-09-20  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: pachi esta revisando TextViewWidget.xml

2004-09-14  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * TreeViewWidget.xml: incluyo el TreeModel.xml desde el
        TreeViewWidget.xml

2004-09-01  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: meto a pachi en los creditos

        * DrawingArea.xml: revisado por pachi

2004-08-26  Lorenzo Gil Sanchez  <lgs@sexmachine.homelinux.net>

        * pygtk2-tut.xml: marco PackingWidgets como revisado

        * PackingWidgets.xml: revisado

2004-08-24  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: le asigno DrawingArea a pachi

2004-08-20  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: meto a Fernando en los créditos

        * EntryCompletion.xml : traducido por Fernando

2004-08-19  Lorenzo Gil Sanchez  <lgs@sexmachine.homelinux.net>

        * pygtk2-tut.xml: marco MovingOn como revisado

        * MovingOn.xml: capitulo revisado

2004-08-19  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: Inigo sigue mandando parches:
        AdvancedEventAndSignalHandling, DragAndDrop, GtkRcFiles,
        SettingWidgetAttributes, TimeoutsIOAndIdleFunctions,
        UndocumentedWidgets
        * UndocumentedWidgets.xml: Revisado por Inigo
        * TimeoutsIOAndIdleFunctions.xml:
        * SettingWidgetAttributes.xml:
        * GtkRcFiles.xml:
        * DragAndDrop.xml:
        * AdvancedEventAndSignalHandling.xml:


        * Contributing.xml: he puesto el encoding iso-8859-1 en el inicio del
        prologo xml
        * Credits.xml:
        * Copyright.xml:
```

```
        * pygtk2-tut.xml: el parche de Inigo de DragAndDrop, GtkRcFiles,
        TipsForWritingPygtkApplications, Contributing, Credits y Copyright
        se acepta

2004-08-18  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: meto dos nuevos xml (NewWidgetsAndObjects y
        NewInPyGTK24) y le asigno el segundo a fsmw

2004-08-13  Lorenzo Gil Sanchez  <lgs@sicem.biz>

        * pygtk2-tut.xml: me apropio del capitulo MovingOn.xml

2004-08-11  Lorenzo Gil Sanchez  <lgs@sexmachine.homelinux.net>

        * pygtk2-tut.xml: quito la marca para indicar que ya se ha revisado

        * GettingStarted.xml: capitulo revisado

        * pygtk2-tut.xml: me apropio del capitulo GettingStarted.xml

2004-07-28  John Finlay  <finlay@moeraki.com>

        * NewWidgetsAndObject.xml Create.

        * pygtk2-tut.xml Add NewWidgetsAndObject.xml file. Bump version
        number and set date.

2004-07-20  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (sec-ManagingCellRenderers) Fix title.
        More detail on set_sort_column_id().

2004-07-12  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (sec-CreatingTreeView) Fix faulty capitalization.
        (thanks to Doug Quale)

2004-07-08  John Finlay  <finlay@moeraki.com>

        * Adjustments.xml AdvancedEventAndSignalHandling.xml
        ButtonWidget.xml ChangeLog ContainerWidgets.xml
        DragAndDrop.xml DrawingArea.xml GettingStarted.xml
        ManagingSelections.xml MenuWidget.xml
        MiscellaneousWidgets.xml MovingOn.xml
        PackingWidgets.xml RangeWidgets.xml Scribble.xml
        SettingWidgetAttributes.xml TextViewWidget.xml
        TimeoutsIOAndIdleFunctions.xml WidgetOverview.xml
        Update files with example programs.

2004-07-06  John Finlay  <finlay@moeraki.com>

        * examples/*.py Update examples to eliminate deprecated methods
        and use import pygtk.

        ============== 2.1 ==================
2004-07-06  John Finlay  <finlay@moeraki.com>

        * pygtk2-tut.xml Bump version number to 2.1 and set
```

```
        pubdate.

        * TreeViewWidgets.xml Revise the treeviewdnd.py example to
        illustrate row reordering with external drag and drop and
        add explanation.

2004-07-03  John Finlay  <finlay@moeraki.com>

        * TimeoutsIOAndIdleFunctions.xml Update descriptions to use
        the gobject functions.

2004-06-30  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml Extract the CellRenderers section into
        CellRenderers.xml.

        * CellRenderers.xml Create and add section on editable
        CellRendererText.

        * TreeViewWidget.xml Extract the TreeModel section and put into
        new file TreeModel.xml. Add detail to the TreeViewColumn use of
        its sort column ID.

        * TreeModel.xml Create and add section on sorting TreeModel rows
        using the TreeSortable interface.

2004-06-27  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (Cell Data Function) Add filelisting example
        using cell data functions. Add XInclude header to include generic
        tree model and cell renderer subsections.
        Fix typos and errors in links. Fix bugs in example listings.
        Add section on TreeModel signals.

2004-06-22  John Finlay  <finlay@moeraki.com>

        * Introduction.xml Add note about pygtkconsole.py and gpython.py
        programs do not work on Windows. Thanks to vector180.

2004-06-14  John Finlay  <finlay@moeraki.com>

        * DragAndDrop.xml Fix signal lists for drag source and dest.
        Add detail to the overview drag cycle. Add detail about signal
        handler operation.

        * DragAndDrop.xml Add small example program dragtargets.py to
        print out drag targets.

2004-05-31  John Finlay  <finlay@moeraki.com>

        * GettingStarted.xml Change wording in helloworld.py example
        program - delete_event() comments confusing. Thanks to Ming Hua.

2004-05-28  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml (TreeModelFilter) Replace 'file' with 'filter'.
        Thanks to Guilherme Salgado.

2004-05-27  John Finlay  <finlay@moeraki.com>
```

```
        * TreeViewWidget.xml (AccessingDataValues) Fix store.set example
        column number wrong. Thanks to Rafael Villar Burke and Guilherme
        Salgado.
        (CellRendererAttributes) Fix error. Thanks to Doug Quale.
        (TreeModelIntroduction)
        (PythonProtocolSupport) Fix grammatical and spelling errors.
        Thanks to Thomas Mills Hinkle.

2004-05-25  John Finlay  <finlay@moeraki.com>

        * Introduction.xml Add reference links to www.pygtk.org website
        and describe some of its resources.

        ============ 2.0 ==============
2004-05-24  John Finlay  <finlay@moeraki.com>

        * TreeViewWidget.xml Add beginning of tutorial chapter.

        * Introduction.xml Add reference to gpython.py program.

        * pygtk2-tut.xml Bump release number to 2.0.

2004-03-31  John Finlay  <finlay@moeraki.com>

        * MiscellaneousWidgets.xml Fix bug in calendar.py example causing
        date string to be off by one day in some time zones. Fixes #138487.
        (thanks to Eduard Luhtonen)

2004-01-28  John Finlay  <finlay@moeraki.com>

        * DrawingArea.xml Modify description of DrawingArea to clarify that
        drawing is done on the wrapped gtk.gdk.Window. Modify GC description
        to clarify that new GCs created from drawables. (thanks to Antoon
        Pardon)

        * UndocumentedWidgets.xml Remove the section on Plugs and Sockets -
        now in ContainerWidgets.xml.

        * ContainerWidgets.xml Add section on Plugs and Sockets written
        by Nathan Hurst.

        * pygtk2-tut.xml Change date and version number.

2003-11-05  John Finlay  <finlay@moeraki.com>

        * Introduction.xml Add reference to the PyGTK 2.0 Reference Manual.

2003-11-04  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml
        * RangeWidgets.xml
        * WidgetOverview.xml Remove reference to testgtk.py since it doesn't
        exist in PyGTK 2.0 (thanks to Steve Chaplin)

2003-10-07  John Finlay  <finlay@moeraki.com>

        * TextViewWidget.xml Change PANGO_ to pango. (thanks to Stephane Klein)
        * pygtk2-tut.xml Change date and version number.
```

```
2003-10-06  John Finlay  <finlay@moeraki.com>

        * GettingStarted.xml Change third to second in description of signal
        handler arguments. (thanks to Kyle Smith)

2003-09-26  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml Fix text layout error in frame shadow
        description (thanks to Steve Chaplin)

2003-09-19  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml
        * layout.py Use random module instead of whrandom in
        layout.py example program (thanks to Steve Chaplin)

        * PackingWidgets.xml
        * packbox.py Use set_size_request() instead of set_usize() in
        packbox.py example (thanks to Steve Chaplin)

2003-07-11  John Finlay  <finlay@moeraki.com>

        * ContainerWidgets.xml Fix link references to class-gtkalignment to
        use a ulink instead of a link.

        * ChangeLog Add this change log file

        * pygtk2-tut.xml Change date and add a version number. Add ChangeLog
        as an appendix.
```

Índice alfabético

s
shadow_type, 60