PyGObject Tutorial Documentation

Release 1.0

Sebastian Pölsterl

May 08, 2013

CONTENTS

Release 3.4

Date January 14, 2013

This tutorial gives an introduction to writing GTK+ 3 applications in Python.

Prior to working through this tutorial, it is recommended that you have a reasonable grasp of the Python programming language. GUI programming introduces new problems compared to interacting with the standard output (console / terminal). It is necessary for you to know how to create and run Python files, understand basic interpreter errors, and work with strings, integers, floats and Boolean values. For the more advanced widgets in this tutorial, good knowledge of lists and tuples will be needed.

Although this tutorial describes the most important classes and methods within GTK+ 3, it is not supposed to serve as an API reference. Please refer to the GTK+ 3 Reference Manual for a detailed description of the API.

Contents:

CHAPTER

ONE

INSTALLATION

The fist step before we start with actual coding consists of setting up PyGObject and its dependencies. PyGObject is a Python module that enables developers to access GObject-based libraries such as GTK+ within Python. It exclusively supports GTK+ version 3 or later. If you want to use GTK+ 2 in your application, use PyGTK, instead.

1.1 Dependencies

- GTK+3
- Python 2 (2.6 or later) or Python 3 (3.1 or later)
- gobject-introspection

1.2 Prebuilt Packages

Recent versions of PyGObject and its dependencies are packaged by nearly all major Linux distributions. So, if you use Linux, you can probably get started by installing the package from the official repository for your distribution.

1.3 Installing From Source

The easiest way to install PyGObject from source is using JHBuild. It is designed to easily build source packages and discover what dependencies need to be build and in what order. To setup JHBuild, please follow the JHBuild manual.

Once you have installed JHBuild successfully, download the latest configuration from [1]. Copy files with the suffix *.modules* to JHBuild's module directory and the file *sample-tarball.jhbuildrc* to *~/.jhbuildrc*.

If you have not done it before, verify that your build environment is setup correctly by running:

```
$ jhbuild sanitycheck
```

It will print any applications and libraries that are currently missing on your system but required for building. You should install those using your distribution's package repository. A list of package names for different distributions is maintained on the GNOME wiki. Run the command above again to ensure the required tools are present.

Executing the following command will build PyGObject and all its dependencies:

```
$ jhbuild build pygobject
```

[1] http://download.gnome.org/teams/releng/

Finally, you might want to install GTK+ from source as well:

```
$ jhbuild build gtk+
```

To start a shell with the same environment as used by JHBuild, run:

```
$ jhbuild shell
```

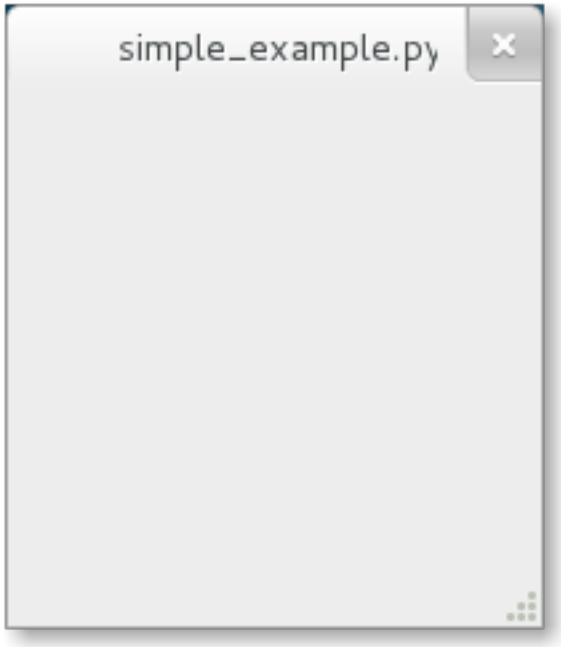
CHAPTER

TWO

GETTING STARTED

2.1 Simple Example

To start with our tutorial we create the simplest program possible. This program will create an empty 200 x 200 pixel window.




```
1 #!/usr/bin/python
2 from gi.repository import Gtk
3
4 win = Gtk.Window()
5 win.connect("delete-event", Gtk.main_quit)
6 win.show_all()
7 Gtk.main()
```

We will now explain each line of the example.

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

The first line of all Python programs should start with `#!` followed by the path to the Python interpreter you want to invoke.

```
from gi.repository import Gtk
```

In order to access GTK+ classes and functions we first must import the Gtk module. The next line creates an empty window.

```
win = Gtk.Window()
```

Followed by connecting to the window's delete event to ensure that the application is terminated if we click on the *x* to close the window.

```
win.connect("delete-event", Gtk.main_quit)
```

In the next step we display the window.

```
win.show_all()
```

Finally, we start the GTK+ processing loop which we quit when the window is closed (see line 5).

```
Gtk.main()
```

To run the program, open a terminal, change to the directory of the file, and enter:

```
python simple_example.py
```

2.2 Extended Example

For something a little more useful, here's the PyGObject version of the classic "Hello World" program.


```
1  #!/usr/bin/python
2  from gi.repository import Gtk
3
4  class MyWindow(Gtk.Window):
5
6      def __init__(self):
7          Gtk.Window.__init__(self, title="Hello World")
8
9          self.button = Gtk.Button(label="Click Here")
10         self.button.connect("clicked", self.on_button_clicked)
11         self.add(self.button)
12
13     def on_button_clicked(self, widget):
14         print "Hello World"
15
16 win = MyWindow()
17 win.connect("delete-event", Gtk.main_quit)
18 win.show_all()
19 Gtk.main()
```

This example differs from the simple example as we sub-class `Gtk.Window` to define our own `MyWindow` class.

```
class MyWindow(Gtk.Window):
```

In the class's constructor we have to call the constructor of the super class. In addition, we tell it to set the value of the property *title* to *Hello World*.

```
        Gtk.Window.__init__(self, title="Hello World")
```

The next three lines are used to create a button widget, connect to its *clicked* signal and add it as child to the top-level window.

```
        self.button = Gtk.Button(label="Click Here")
        self.button.connect("clicked", self.on_button_clicked)
        self.add(self.button)
```

Accordingly, the method `on_button_clicked()` will be called if you click on the button.

```
    def on_button_clicked(self, widget):
        print "Hello World"
```

The last block, outside of the class, is very similar to the simple example above, but instead of creating an instance of the generic `Gtk.Window` class, we create an instance of `MyWindow`.

CHAPTER

THREE

BASICS

This section will introduce some of the most important aspects of GTK+.

3.1 Main loop and Signals

Like most GUI toolkits, GTK+ uses an event-driven programming model. When the user is doing nothing, GTK+ sits in the main loop and waits for input. If the user performs some action - say, a mouse click - then the main loop "wakes up" and delivers an event to GTK+.

When widgets receive an event, they frequently emit one or more signals. Signals notify your program that "something interesting happened" by invoking functions you've connected to the signal. Such functions are commonly known as *callbacks*. When your callbacks are invoked, you would typically take some action - for example, when an Open button is clicked you might display a file chooser dialog. After a callback finishes, GTK+ will return to the main loop and await more user input.

A generic example is:

```
handler_id = widget.connect("event", callback, data)
```

Firstly, *widget* is an instance of a widget we created earlier. Next, the event we are interested in. Each widget has its own particular events which can occur. For instance, if you have a button you usually want to connect to the "clicked" event. This means that when the button is clicked, the signal is issued. Thirdly, the *callback* argument is the name of the callback function. It contains the code which runs when signals of the specified type are issued. Finally, the *data* argument includes any data which should be passed when the signal is issued. However, this argument is completely optional and can be left out if not required.

The function returns a number that identifies this particular signal-callback pair. It is required to disconnect from a signal such that the callback function will not be called during any future or currently ongoing emissions of the signal it has been connected to.

```
widget.disconnect(handler_id)
```

Almost all applications will connect to the "delete-event" signal of the top-level window. It is emitted if a user requests that a toplevel window is closed. The default handler for this signal destroys the window, but does not terminate the application. Connecting the "delete-event" signal to the function `Gtk.main_quit()` will result in the desired behaviour.

```
window.connect("delete-event", Gtk.main_quit)
```

Calling `Gtk.main_quit()` makes the main loop inside of `Gtk.main()` return.

3.2 Properties

Properties describe the configuration and state of widgets. As for signals, each widget has its own particular set of properties. For example, a button has the property "label" which contains the text of the label widget inside the button. You can specify the name and value of any number of properties as keyword arguments when creating an instance of a widget. To create a label aligned to the right with the text "Hello World" and an angle of 25 degrees, use:

```
label = Gtk.Label(label="Hello World", angle=25, halign=Gtk.Align.END)
```

which is equivalent to

```
label = Gtk.Label()
label.set_label("Hello World")
label.set_angle(25)
label.set_halign(Gtk.Align.END)
```

Instead of using getters and setters you can also get and set the properties with `widget.get_property("prop-name")` and `widget.set_property("prop-name", value)`, respectively.

CHAPTER

FOUR

HOW TO DEAL WITH STRINGS

This section explains how strings are represented in Python 2.x, Python 3.x and GTK+ and discusses common errors that arise when working with strings.

4.1 Definitions

Conceptionally, a string is a list of characters such as 'A', 'B', 'C' or 'É'. **Characters** are abstract representations and their meaning depends on the language and context they are used in. The Unicode standard describes how characters are represented by **code points**. For example the characters above are represented with the code points U+0041, U+0042, U+0043, and U+00C9, respectively. Basically, code points are numbers in the range from 0 to 0x10FFFF.

As mentioned earlier, the representation of a string as a list of code points is abstract. In order to convert this abstract representation into a sequence of bytes the Unicode string must be **encoded**. The simplest from of encoding is ASCII and is performed as follows:

1. If the code point is < 128, each byte is the same as the value of the code point.
2. If the code point is 128 or greater, the Unicode string can't be represented in this encoding. (Python raises a `UnicodeEncodeError` exception in this case.)

Although ASCII encoding is simple to apply it can only encode for 128 different characters which is hardly enough. One of the most commonly used encodings that addresses this problem is UTF-8 (it can handle any Unicode code point). UTF stands for "Unicode Transformation Format", and the '8' means that 8-bit numbers are used in the encoding.

4.2 Python 2

4.2.1 Python 2.x's Unicode Support

Python 2 comes with two different kinds of objects that can be used to represent strings, `str` and `unicode`. Instances of the latter are used to express Unicode strings, whereas instances of the `str` type are byte representations (the encoded string). Under the hood, Python represents Unicode strings as either 16- or 32-bit integers, depending on how the Python interpreter was compiled. Unicode strings can be converted to 8-bit strings with `unicode.encode()`:

```
>>> unicode_string = u"Fu\u00dfb\u00e4lle"
>>> print unicode_string
Fußbälle
>>> type(unicode_string)
<type 'unicode'>
```

```
>>> unicode_string.encode("utf-8")
'Fu\xc3\x9fb\xc3\xa4lle'
```

Python's 8-bit strings have a `str.decode()` method that interprets the string using the given encoding:

```
>>> utf8_string = unicode_string.encode("utf-8")
>>> type(utf8_string)
<type 'str'>
>>> u2 = utf8_string.decode("utf-8")
>>> unicode_string == u2
True
```

Unfortunately, Python 2.x allows you to mix `unicode` and `str` if the 8-bit string happened to contain only 7-bit (ASCII) bytes, but would get `UnicodeDecodeError` if it contained non-ASCII values:

```
>>> utf8_string = " sind rund"
>>> unicode_string + utf8_string
u'Fu\xdfb\xe4lle sind rund'
>>> utf8_string = " k\xc3\xb6nnten rund sein"
>>> print utf8_string
 könnten rund sein
>>> unicode_string + utf8_string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 2: ordinal not in range(128)
```

4.2.2 Unicode in GTK+

GTK+ uses UTF-8 encoded strings for all text. This means that if you call a method that returns a string you will always obtain an instance of the `str` type. The same applies to methods that expect one or more strings as parameter, they must be UTF-8 encoded. However, for convenience PyGObject will automatically convert any `unicode` instance to `str` if supplied as argument:

```
>>> from gi.repository import Gtk
>>> label = Gtk.Label()
>>> unicode_string = u"Fu\u00dfb\u00e4lle"
>>> label.set_text(unicode_string)
>>> txt = label.get_text()
>>> type(txt), txt
(<type 'str'>, 'Fu\xc3\x9fb\xc3\xa4lle')
>>> txt == unicode_string
__main__:1: UnicodeWarning: Unicode equal comparison failed to convert both arguments to Unicode - in
False
```

Note the warning at the end. Although we called `Gtk.Label.set_text()` with a `unicode` instance as argument, `Gtk.Label.get_text()` will always return a `str` instance. Accordingly, `txt` and `unicode_string` are *not* equal.

This is especially important if you want to internationalize your program using gettext. You have to make sure that gettext will return UTF-8 encoded 8-bit strings for all languages. In general it is recommended to not use `unicode` objects in GTK+ applications at all and only use UTF-8 encoded `str` objects since GTK+ does not fully integrate with `unicode` objects. Otherwise, you would have to decode the return values to Unicode strings each time you call a GTK+ method:

```
>>> txt = label.get_text().decode("utf-8")
>>> txt == unicode_string
True
```

4.3 Python 3

4.3.1 Python 3.x's Unicode support

Since Python 3.0, all strings are stored as Unicode in an instance of the `str` type. *Encoded* strings on the other hand are represented as binary data in the form of instances of the `bytes` type. Conceptionally, `str` refers to *text*, whereas `bytes` refers to *data*. Use `str.encode()` to go from `str` to `bytes`, and `bytes.decode()` to go from `bytes` to `str`.

In addition, it is no longer possible to mix Unicode strings with encoded strings, because it will result in a `TypeError`:

```
>>> text = "Fu\u00dfb\u00e4lle"
>>> data = b" sind rund"
>>> text + data
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> text + data.decode("utf-8")
'Fußbälle sind rund'
>>> text.encode("utf-8") + data
b'Fu\xc3\x9fb\xc3\xa4lle sind rund'
```

4.3.2 Unicode in GTK+

As a consequence, things are much cleaner and consistent with Python 3.x, because PyGObject will automatically encode/decode to/from UTF-8 if you pass a string to a method or a method returns a string. Strings, or *text*, will always be represented as instances of `str` only:

```
>>> from gi.repository import Gtk
>>> label = Gtk.Label()
>>> text = "Fu\u00dfb\u00e4lle"
>>> label.set_text(text)
>>> txt = label.get_text()
>>> type(txt), txt
(<class 'str'>, 'Fußbälle')
>>> txt == text
True
```

4.4 References

What's new in Python 3.0 describes the new concepts that clearly distinguish between text and data.

The Unicode HOWTO discusses Python 2.x's support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode.

The Unicode HOWTO for Python 3.x discusses Unicode support in Python 3.x.

UTF-8 encoding table and Unicode characters contains a list of Unicode code points and their respective UTF-8 encoding.

CHAPTER

FIVE

LAYOUT CONTAINERS

While many GUI toolkits require you to precisely place widgets in a window, using absolute positioning, GTK+ uses a different approach. Rather than specifying the position and size of each widget in the window, you can arrange your widgets in rows, columns, and/or tables. The size of your window can be determined automatically, based on the sizes of the widgets it contains. And the sizes of the widgets are, in turn, determined by the amount of text they contain, or the minimum and maximum sizes that you specify, and/or how you have requested that the available space should be shared between sets of widgets. You can perfect your layout by specifying padding distance and centering values for each of your widgets. GTK+ then uses all this information to resize and reposition everything sensibly and smoothly when the user manipulates the window.

GTK+ arranges widgets hierarchically, using *containers*. They are invisible to the end user and are inserted into a window, or placed within each other to layout components. There are two flavours of containers: single-child containers, which are all descendants of `Gtk.Bin`, and multiple-child containers, which are descendants of `Gtk.Container`. The most commonly used are vertical or horizontal boxes (`Gtk.Box`), tables (`Gtk.Table`) and grids (`Gtk.Grid`).

5.1 Boxes

Boxes are invisible containers into which we can pack our widgets. When packing widgets into a horizontal box, the objects are inserted horizontally from left to right or right to left depending on whether `Gtk.Box.pack_start()` or `Gtk.Box.pack_end()` is used. In a vertical box, widgets are packed from top to bottom or vice versa. You may use any combination of boxes inside or beside other boxes to create the desired effect.

5.1.1 Box Objects

class `Gtk.`**`Box`**`(`[*homogenous*[, *spacing*]])

The rectangular area of a `Gtk.Box` is organized into either a single row or a single column of child widgets depending upon whether the "orientation" property is set to `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`.

If *homogeneous* is `True`, all widgets in the box will be the same size, of which the size is determined by the largest child widget. If it is omitted, *homogeneous* defaults to `False`.

spacing is the number of pixels to place by default between children. If omitted, no spacing is used, i.e. *spacing* is set to 0.

By default, child widgets are organized into a single row, i.e. the "orientation" property is set to `Gtk.Orientation.HORIZONTAL`.

`Gtk.Box` uses a notion of *packing*. Packing refers to adding widgets with reference to a particular position in a `Gtk.Container`. For a `Gtk.Box`, there are two reference positions: the start and the end of the box. If "orientation" is `Gtk.Orientation.VERTICAL`, the start is defined as the top of the box and the end is

defined as the bottom. If "orientation" is `Gtk.Orientation.HORIZONTAL`, the start is defined as the left side and the end is defined as the right side.

pack_start(*child*, *expand*, *fill*, *padding*)
Adds *child* to box, packed with reference to the start of box. The *child* is packed after any other child packed with reference to the start of box.

child should be a `Gtk.Widget` to be added to this box. The *expand* argument when set to `True` allows the *child* widget to take all available space it can. Alternately, if the value is set to `False`, the box will be shrunken to the same size as the child widget.

If the *fill* argument is set to `True`, the *child* widget takes all available space and is equal to the size of the box. This only has an effect when *expand* is set to `True`. A child is always allocated the full height of a horizontally oriented and the full width of a vertically oriented box. This option affects the other dimension.

padding defines extra space in pixels to put between this child and its neighbours, over and above the global amount specified by "spacing" property. If *child* is a widget at one of the reference ends of box, then padding pixels are also put between *child* and the reference edge of this box.

pack_end(*child*, *expand*, *fill*, *padding*)
Adds *child* to box, packed with reference to the end of box. The *child* is packed after (away from end of) any other child packed with reference to the end of box.

Arguments are the same as for `pack_start()`.

set_homogeneous(*homogeneous*)
If *homogeneous* is `True`, all widgets in the box will be the same size, of which the size is determined by the largest child widget.

5.1.2 Example

Let's take a look at a slightly modified version of the extended example with two buttons.


```
from gi.repository import Gtk

class MyWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Hello World")

        self.box = Gtk.Box(spacing=6)
        self.add(self.box)

        self.button1 = Gtk.Button(label="Hello")
        self.button1.connect("clicked", self.on_button1_clicked)
        self.box.pack_start(self.button1, True, True, 0)

        self.button2 = Gtk.Button(label="Goodbye")
        self.button2.connect("clicked", self.on_button2_clicked)
        self.box.pack_start(self.button2, True, True, 0)
```

```
18
19      def on_button1_clicked(self, widget):
20          print "Hello"
21
22      def on_button2_clicked(self, widget):
23          print "Goodbye"
24
25  win = MyWindow()
26  win.connect("delete-event", Gtk.main_quit)
27  win.show_all()
28  Gtk.main()
```

First, we create a horizontally orientated box container where 6 pixels are placed between children. This box becomes the child of the top-level window.

```
        self.box = Gtk.Box(spacing=6)
        self.add(self.box)
```

Subsequently, we add two different buttons to the box container.

```
        self.button1 = Gtk.Button(label="Hello")
        self.button1.connect("clicked", self.on_button1_clicked)
        self.box.pack_start(self.button1, True, True, 0)

        self.button2 = Gtk.Button(label="Goodbye")
        self.button2.connect("clicked", self.on_button2_clicked)
        self.box.pack_start(self.button2, True, True, 0)
```

While with `Gtk.Box.pack_start()` widgets are positioned from left to right, `Gtk.Box.pack_end()` positions them from right to left.

5.2 Grid

`Gtk.Grid` is a container which arranges its child widgets in rows and columns, but you do not need to specify the dimensions in the constructor. Children are added using `Gtk.Grid.attach()`. They can span multiple rows or columns. It is also possible to add a child next to an existing child, using `Gtk.Grid.attach_next_to()`.

`Gtk.Grid` can be used like a `Gtk.Box` by just using `Gtk.Grid.add()`, which will place children next to each other in the direction determined by the "orientation" property (defaults to `Gtk.Orientation.HORIZONTAL`).

5.2.1 Grid Objects

class `Gtk.`**`Grid`**

Creates a new grid widget.

`attach`(*child*, *left*, *top*, *width*, *height*)

Adds *child* to this grid.

The position of *child* is determined by the index of the cell left to it (*left*) and above of it (*top*). The number of 'cells' that *child* will occupy is determined by *width* and *height*.

`attach_next_to`(*child*, *sibling*, *side*, *width*, *height*)

Adds *child* to this grid, next to *sibling*. *side* is the side of *sibling* that *child* is positioned next to. It can be one of

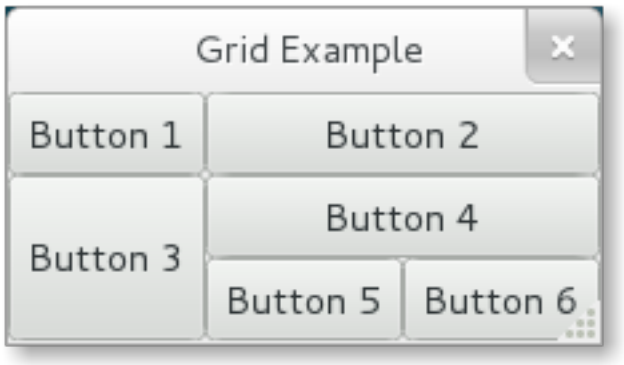
- `Gtk.PositionType.LEFT`

- `Gtk.PositionType.RIGHT`
- `Gtk.PositionType.TOP`
- `Gtk.PositionType.BOTTOM`

width and *height* determine the number of 'cells' that *child* will occupy.

add(*widget*)
Adds *widget* to this grid in the direction determined by the "orientation" property.

5.2.2 Example




```
from gi.repository import Gtk

class GridWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Grid Example")

        grid = Gtk.Grid()
        self.add(grid)

        button1 = Gtk.Button(label="Button 1")
        button2 = Gtk.Button(label="Button 2")
        button3 = Gtk.Button(label="Button 3")
        button4 = Gtk.Button(label="Button 4")
        button5 = Gtk.Button(label="Button 5")
        button6 = Gtk.Button(label="Button 6")

        grid.add(button1)
        grid.attach(button2, 1, 0, 2, 1)
        grid.attach_next_to(button3, button1, Gtk.PositionType.BOTTOM, 1, 2)
        grid.attach_next_to(button4, button3, Gtk.PositionType.RIGHT, 2, 1)
        grid.attach(button5, 1, 2, 1, 1)
        grid.attach_next_to(button6, button5, Gtk.PositionType.RIGHT, 1, 1)

win = GridWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

5.3 Table

Tables allows us to place widgets in a grid similar to `Gtk.Grid`.

The grid's dimensions need to be specified in the `Gtk.Table` constructor. To place a widget into a box, use `Gtk.Table.attach()`.

`Gtk.Table.set_row_spacing()` and `Gtk.Table.set_col_spacing()` set the spacing between the rows at the specified row or column. Note that for columns, the space goes to the right of the column, and for rows, the space goes below the row.

You can also set a consistent spacing for all rows and/or columns with `Gtk.Table.set_row_spacings()` and `Gtk.Table.set_col_spacings()`. Note that with these calls, the last row and last column do not get any spacing.

5.3.1 Table Objects

Deprecated since version 3.4: Use `Gtk.Grid` instead.

class `Gtk.`**`Table`**(*rows*, *columns*, *homogeneous*)

The first argument is the number of rows to make in the table, while the second, obviously, is the number of columns. If *homogeneous* is `True`, the table cells will all be the same size (the size of the largest widget in the table).

`attach`(*child*, *left_attach*, *right_attach*, *top_attach*, *bottom_attach*[, *xoptions*[, *yoptions*[, *xpadding*[, *ypadding*]]]])

Adds a widget to a table.

child is the widget that should be added to the table. The number of 'cells' that a widget will occupy is specified by *left_attach*, *right_attach*, *top_attach* and *bottom_attach*. These each represent the leftmost, rightmost, uppermost and lowest column and row numbers of the table. (Columns and rows are indexed from zero).

For example, if you want a button in the lower-right cell of a 2 x 2 table, and want it to occupy that cell only, then the code looks like the following.

```
button = Gtk.Button()
table = Gtk.Table(2, 2, True)
table.attach(button, 1, 2, 1, 2)
```

If, on the other hand, you wanted a widget to take up the entire top row of our 2 x 2 table, you'd use

```
table.attach(button, 0, 2, 0, 1)
```

xoptions and *yoptions* are used to specify packing options and may be bitwise ORed together to allow multiple options. These options are:

- `Gtk.AttachOptions.EXPAND`: The widget should expand to take up any extra space in its container that has been allocated.
- `Gtk.AttachOptions.FILL`: The widget will expand to use all the room available.
- `Gtk.AttachOptions.SHRINK`: Reduce size allocated to the widget to prevent it from moving off screen.

If omitted, *xoptions* and *yoptions* defaults to `Gtk.AttachOptions.EXPAND | Gtk.AttachOptions.FILL`.

Finally, the padding arguments work just as they do for `Gtk.Box.pack_start()`. If omitted, *xpadding* and *ypadding* defaults to `0`.

set_row_spacing(*row*, *spacing*)
Changes the space between a given table row and the subsequent row.

set_col_spacing(*col*, *spacing*)
Alters the amount of space between a given table column and the following column.

set_row_spacings(*spacing*)
Sets the space between every row in this table equal to *spacing*.

set_col_spacings(*spacing*)
Sets the space between every column in this table equal to *spacing*.

5.3.2 Example


```
from gi.repository import Gtk

class TableWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Table Example")

        table = Gtk.Table(3, 3, True)
        self.add(table)

        button1 = Gtk.Button(label="Button 1")
        button2 = Gtk.Button(label="Button 2")
        button3 = Gtk.Button(label="Button 3")
        button4 = Gtk.Button(label="Button 4")
        button5 = Gtk.Button(label="Button 5")
        button6 = Gtk.Button(label="Button 6")

        table.attach(button1, 0, 1, 0, 1)
        table.attach(button2, 1, 3, 0, 1)
        table.attach(button3, 0, 1, 1, 3)
        table.attach(button4, 1, 3, 1, 2)
        table.attach(button5, 1, 2, 2, 3)
        table.attach(button6, 2, 3, 2, 3)

win = TableWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

CHAPTER

SIX

LABEL

Labels are the main method of placing non-editable text in windows, for instance to place a title next to a `Gtk.Entry` widget. You can specify the text in the constructor, or later with the `Gtk.Label.set_text()` or `Gtk.Label.set_markup()` methods.

The width of the label will be adjusted automatically. You can produce multi-line labels by putting line breaks ("\n") in the label string.

Labels can be made selectable with `Gtk.Label.set_selectable()`. Selectable labels allow the user to copy the label contents to the clipboard. Only labels that contain useful-to-copy information — such as error messages — should be made selectable.

The label text can be justified using the `Gtk.Label.set_justify()` method. The widget is also capable of word-wrapping, which can be activated with `Gtk.Label.set_line_wrap()`.

`Gtk.Label` support some simple formatting, for instance allowing you to make some text bold, colored, or larger. You can do this by providing a string to `Gtk.Label.set_markup()`, using the Pango Markup syntax [1]. For instance, `<b>bold text</b> and <s>strikethrough text</s>`. In addition, `Gtk.Label` supports clickable hyperlinks. The markup for links is borrowed from HTML, using the a with href and title attributes. GTK+ renders links similar to the way they appear in web browsers, with colored, underlined text. The title attribute is displayed as a tooltip on the link.

```
label.set_markup("Go to <a href=\"http://www.gtk.org\" title=\"Our website\">GTK+ website</a> for mo
```

Labels may contain *mnemonics*. Mnemonics are underlined characters in the label, used for keyboard navigation. Mnemonics are created by providing a string with an underscore before the mnemonic character, such as "_File", to the functions `Gtk.Label.new_with_mnemonic()` or `Gtk.Label.set_text_with_mnemonic()`. Mnemonics automatically activate any activatable widget the label is inside, such as a `Gtk.Button`; if the label is not inside the mnemonic's target widget, you have to tell the label about the target using `Gtk.Label.set_mnemonic_widget()`.

6.1 Label Objects

class `Gtk.`**`Label`**(*[text]*)

Creates a new label with the given *text* inside it. If *text* is omitted, an empty label is created.

static `new_with_mnemonic`(*text*)

Creates a new label with *text* inside it.

If characters in *text* are preceded by an underscore, they are underlined. If you need a literal underscore character in a label, use '__' (two underscores). The first underlined character represents a keyboard accelerator called a

[1] Pango Markup Syntax, http://developer.gnome.org/pango/stable/PangoMarkupFormat.html

mnemonic. The mnemonic key can be used to activate another widget, chosen automatically, or explicitly using `Gtk.Label.set_mnemonic_widget()`.

If `Gtk.Label.set_mnemonic_widget()` is not called, then the first activatable ancestor of the `Gtk.Label` will be chosen as the mnemonic widget. For instance, if the label is inside a button or menu item, the button or menu item will automatically become the mnemonic widget and be activated by the mnemonic.

set_justify(*justification*)

Sets the alignment of the lines in the text of the label relative to each other. *justification* can be one of `Gtk.Justification.LEFT`, `Gtk.Justification.RIGHT`, `Gtk.Justification.CENTER`, `Gtk.Justification.FILL`. This method has no effect on labels containing only a single line.

set_line_wrap(*wrap*)

If *wrap* is `True`, lines will be broken if text exceeds the widget's size. If *wrap* is `False`, text will be cut off by the edge of the widget if it exceeds the widget size.

set_markup(*markup*)

Parses *markup* which is marked up with the Pango text markup language [1], setting the label's text accordingly. The markup passed must be valid; for example literal <, >, & characters must be escaped as < > and &amp.

set_mnemonic_widget(*widget*)

If the label has been set so that it has an mnemonic key, the label can be associated with a widget that is the target of the mnemonic.

set_selectable(*selectable*)

Selectable labels allow the user to select text from the label, for copy-and-paste.

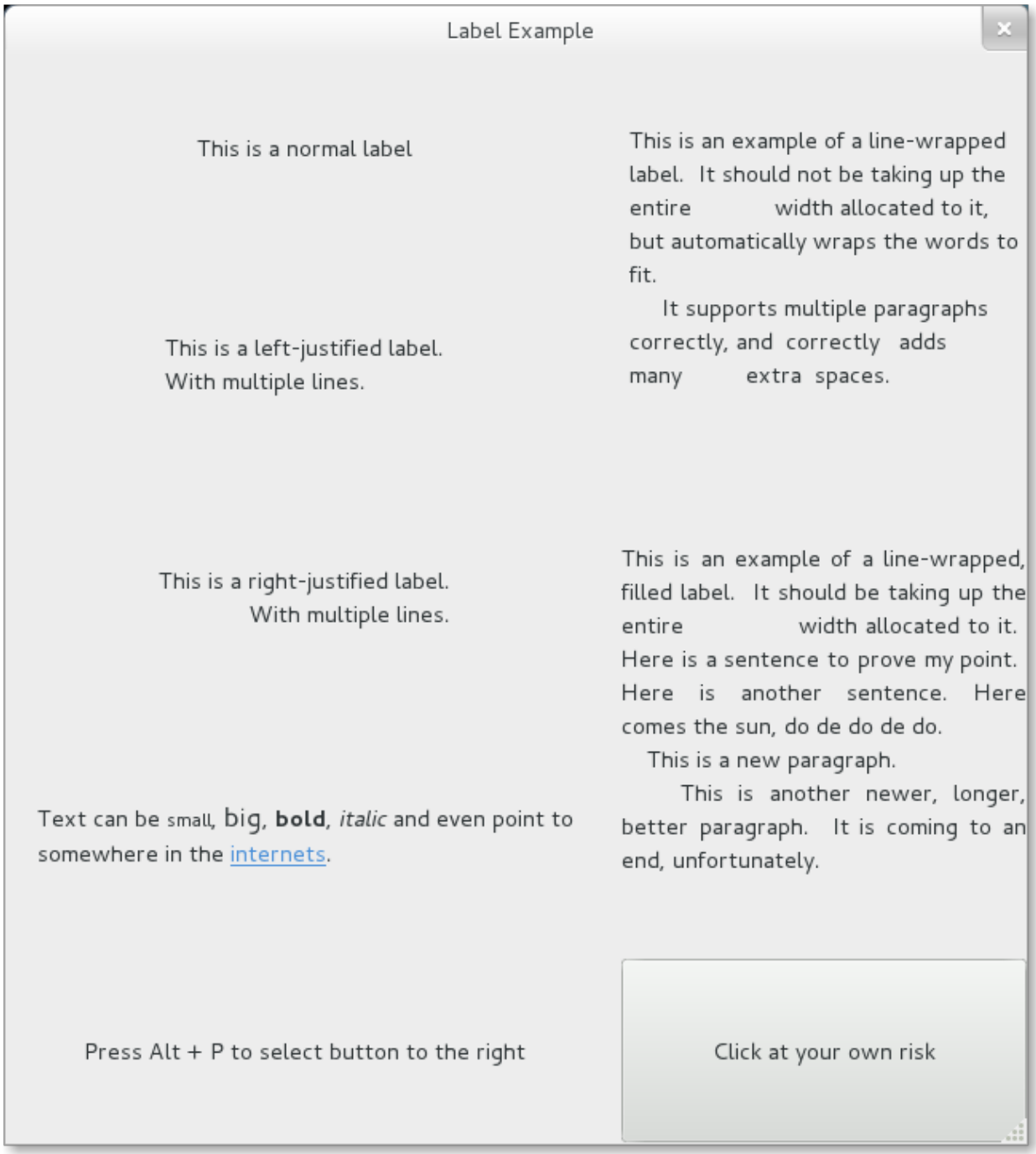
set_text(*text*)

Sets the text within this widget. It overwrites any text that was there before.

set_text_with_mnemonic(*text*)

See `new_with_mnemonic()`.

6.2 Example




```
from gi.repository import Gtk

class LabelWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Label Example")

        hbox = Gtk.Box(spacing=10)
```

```
        hbox.set_homogeneous(False)
        vbox_left = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=10)
        vbox_left.set_homogeneous(False)
        vbox_right = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=10)
        vbox_right.set_homogeneous(False)

        hbox.pack_start(vbox_left, True, True, 0)
        hbox.pack_start(vbox_right, True, True, 0)

        label = Gtk.Label("This is a normal label")
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label()
        label.set_text("This is a left-justified label.\nWith multiple lines.")
        label.set_justify(Gtk.Justification.LEFT)
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label("This is a right-justified label.\nWith multiple lines.")
        label.set_justify(Gtk.Justification.RIGHT)
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label("This is an example of a line-wrapped label.  It "
                          "should not be taking up the entire              "
                          "width allocated to it, but automatically "
                          "wraps the words to fit.\n"
                          "     It supports multiple paragraphs correctly, "
                          "and  correctly   adds "
                          "many          extra  spaces. ")
        label.set_line_wrap(True)
        vbox_right.pack_start(label, True, True, 0)

        label = Gtk.Label("This is an example of a line-wrapped, filled label.  "
                          "It should be taking "
                          "up the entire              width allocated to it.  "
                          "Here is a sentence to prove "
                          "my point.  Here is another sentence. "
                          "Here comes the sun, do de do de do.\n"
                          "    This is a new paragraph.\n"
                          "    This is another newer, longer, better "
                          "paragraph.  It is coming to an end, "
                          "unfortunately.")
        label.set_line_wrap(True)
        label.set_justify(Gtk.Justification.FILL)
        vbox_right.pack_start(label, True, True, 0)

        label = Gtk.Label()
        label.set_markup("Text can be <small>small</small>, <big>big</big>, "
                         "<b>bold</b>, <i>italic</i> and even point to somewhere "
                         "in the <a href=\"http://www.gtk.org\" "
                         "title=\"Click to find out more\">internets</a>.")
        label.set_line_wrap(True)
        vbox_left.pack_start(label, True, True, 0)

        label = Gtk.Label.new_with_mnemonic("_Press Alt + P to select button to the right")
        vbox_left.pack_start(label, True, True, 0)
        label.set_selectable(True)

        button = Gtk.Button(label="Click at your own risk")
```

```
67          label.set_mnemonic_widget(button)
68          vbox_right.pack_start(button, True, True, 0)
69
70          self.add(hbox)
71
72  window = LabelWindow()
73  window.connect("delete-event", Gtk.main_quit)
74  window.show_all()
75  Gtk.main()
```

CHAPTER

SEVEN

ENTRY

Entry widgets allow the user to enter text. You can change the contents with the `Gtk.Entry.set_text()` method, and read the current contents with the `Gtk.Entry.get_text()` method. You can also limit the number of characters the Entry can take by calling `Gtk.Entry.set_max_length()`.

Occasionally you might want to make an Entry widget read-only. This can be done by passing `False` to the `Gtk.Entry.set_editable()` method.

Entry widgets can also be used to retrieve passwords from the user. It is common practice to hide the characters typed into the entry to prevent revealing the password to a third party. Calling `Gtk.Entry.set_visibility()` with `False` will cause the text to be hidden.

`Gtk.Entry` has the ability to display progress or activity information behind the text. This is similar to `Gtk.ProgressBar` widget and is commonly found in web browsers to indicate how much of a page download has been completed. To make an entry display such information, use `Gtk.Entry.set_progress_fraction()`, `Gtk.Entry.set_progress_pulse_step()`, or `Gtk.Entry.progress_pulse()`.

Additionally, an Entry can show icons at either side of the entry. These icons can be activatable by clicking, can be set up as drag source and can have tooltips. To add an icon, use `Gtk.Entry.set_icon_from_stock()` or one of the various other functions that set an icon from an icon name, a pixbuf, or icon theme. To set a tooltip on an icon, use `Gtk.Entry.set_icon_tooltip_text()` or the corresponding function for markup.

7.1 Entry Objects

class `Gtk.`**`Entry`**

`get_text``()`
Retrieves the contents of the entry widget.

`set_text``(`*text*`)`
Sets the text in the widget to the given value, replacing the current contents.

`set_visibility``(`*visible*`)`
Sets whether the contents of the entry are visible or not. When *visible* is set to `False`, characters are displayed as the invisible char, and will also appear that way when the text in the entry widget is copied elsewhere.

`set_max_length``(`*max*`)`
Sets the maximum allowed length of the contents of the widget. If the current contents are longer than the given length, then they will be truncated to fit.

set_editable(*is_editable*)
Determines if the user can edit the text in the editable widget or not. If *is_editable* is `True`, the user is allowed to edit the text in the widget.

set_progress_fraction(*fraction*)
Causes the entry's progress indicator to "fill in" the given fraction of the bar. The fraction should be between 0.0 and 1.0, inclusive.

set_progress_pulse_step(*fraction*)
Sets the fraction of total entry width to move the progress bouncing block for each call to `progress_pulse()`.

progress_pulse()
Indicates that some progress is made, but you don't know how much. Causes the entry's progress indicator to enter "activity mode," where a block bounces back and forth. Each call to `progress_pulse()` causes the block to move by a little bit (the amount of movement per pulse is determined by `set_progress_pulse_step()`).

set_icon_from_stock(*icon_pos*, *stock_id*)
Sets the icon shown in the entry at the specified position from a *stock item*. If *stock_id* is `None`, no icon will be shown in the specified position.

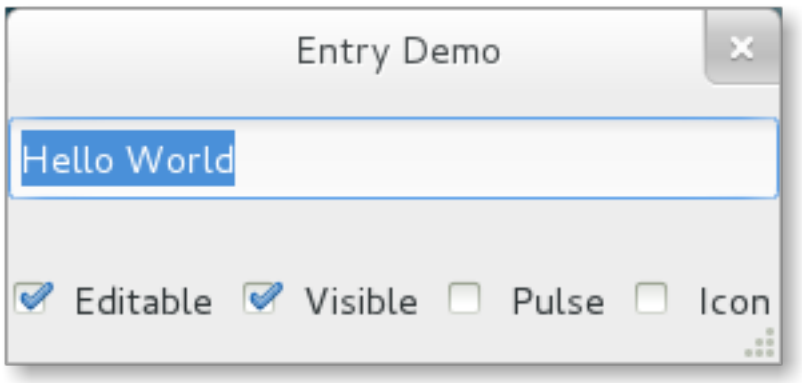
icon_pos specifies the side of the entry at which an icon is placed and can be one of

- `Gtk.EntryIconPosition.PRIMARY`: At the beginning of the entry (depending on the text direction).
- `Gtk.EntryIconPosition.SECONDARY`: At the end of the entry (depending on the text direction).

set_icon_tooltip_text(*icon_pos*, *tooltip*)
Sets *tooltip* as the contents of the tooltip for the icon at the specified position. If *tooltip* is `None`, an existing tooltip is removed.

For allowed values for *icon_pos* see `set_icon_from_stock()`.

7.2 Example




```
1 from gi.repository import Gtk, GObject
2
3 class EntryWindow(Gtk.Window):
4
5     def __init__(self):
6         Gtk.Window.__init__(self, title="Entry Demo")
```

```
        self.set_size_request(200, 100)

        self.timeout_id = None

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
        self.add(vbox)

        self.entry = Gtk.Entry()
        self.entry.set_text("Hello World")
        vbox.pack_start(self.entry, True, True, 0)

        hbox = Gtk.Box(spacing=6)
        vbox.pack_start(hbox, True, True, 0)

        self.check_editable = Gtk.CheckButton("Editable")
        self.check_editable.connect("toggled", self.on_editable_toggled)
        self.check_editable.set_active(True)
        hbox.pack_start(self.check_editable, True, True, 0)

        self.check_visible = Gtk.CheckButton("Visible")
        self.check_visible.connect("toggled", self.on_visible_toggled)
        self.check_visible.set_active(True)
        hbox.pack_start(self.check_visible, True, True, 0)

        self.pulse = Gtk.CheckButton("Pulse")
        self.pulse.connect("toggled", self.on_pulse_toggled)
        self.pulse.set_active(False)
        hbox.pack_start(self.pulse, True, True, 0)

        self.icon = Gtk.CheckButton("Icon")
        self.icon.connect("toggled", self.on_icon_toggled)
        self.icon.set_active(False)
        hbox.pack_start(self.icon, True, True, 0)

    def on_editable_toggled(self, button):
        value = button.get_active()
        self.entry.set_editable(value)

    def on_visible_toggled(self, button):
        value = button.get_active()
        self.entry.set_visibility(value)

    def on_pulse_toggled(self, button):
        if button.get_active():
            self.entry.set_progress_pulse_step(0.2)
            # Call self.do_pulse every 100 ms
            self.timeout_id = GObject.timeout_add(100, self.do_pulse, None)
        else:
            # Don't call self.do_pulse anymore
            GObject.source_remove(self.timeout_id)
            self.timeout_id = None
            self.entry.set_progress_pulse_step(0)

    def do_pulse(self, user_data):
        self.entry.progress_pulse()
        return True

    def on_icon_toggled(self, button):
```

```
65          if button.get_active():
66              stock_id = Gtk.STOCK_FIND
67          else:
68              stock_id = None
69          self.entry.set_icon_from_stock(Gtk.EntryIconPosition.PRIMARY,
70              stock_id)
71
72  win = EntryWindow()
73  win.connect("delete-event", Gtk.main_quit)
74  win.show_all()
75  Gtk.main()
```

CHAPTER

EIGHT

BUTTON WIDGETS

8.1 Button

The Button widget is another commonly used widget. It is generally used to attach a function that is called when the button is pressed.

The `Gtk.Button` widget can hold any valid child widget. That is it can hold most any other standard `Gtk.Widget`. The most commonly used child is the `Gtk.Label`.

Usually, you want to connect to the button's "clicked" signal which is emitted when the button has been pressed and released.

8.1.1 Button Objects

class `Gtk.`**`Button`**(\[*label*\[, *stock*\[, *use_underline*\]\]\])

If label is not `None`, creates a new `Gtk.Button` with a `Gtk.Label` child containing the given text.

If *stock* is not `None`, creates a new button containing the image and text from a *stock item*.

If *use_underline* is set to `True`, an underline in the text indicates the next character should be used for the mnemonic accelerator key.

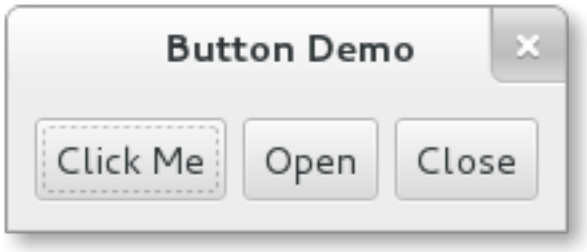
`set_label`(*label*)

Sets the text of the label of the button to *label*.

`set_use_underline`(*use_underline*)

If *True*, an underline in the text of the button label indicates the next character should be used for the mnemonic accelerator key.

8.1.2 Example



```
from gi.repository import Gtk

class ButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Button Demo")
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)

        button = Gtk.Button("Click Me")
        button.connect("clicked", self.on_click_me_clicked)
        hbox.pack_start(button, True, True, 0)

        button = Gtk.Button(stock=Gtk.STOCK_OPEN)
        button.connect("clicked", self.on_open_clicked)
        hbox.pack_start(button, True, True, 0)

        button = Gtk.Button("_Close", use_underline=True)
        button.connect("clicked", self.on_close_clicked)
        hbox.pack_start(button, True, True, 0)

    def on_click_me_clicked(self, button):
        print "\"Click me\" button was clicked"

    def on_open_clicked(self, button):
        print "\"Open\" button was clicked"

    def on_close_clicked(self, button):
        print "Closing application"
        Gtk.main_quit()

win = ButtonWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

8.2 ToggleButton

A `Gtk.ToggleButton` is very similar to a normal `Gtk.Button`, but when clicked they remain activated, or pressed, until clicked again. When the state of the button is changed, the "toggled" signal is emitted.

To retrieve the state of the `Gtk.ToggleButton`, you can use the `Gtk.ToggleButton.get_active()` method. This returns `True` if the button is "down". You can also set the toggle button's state, with `Gtk.ToggleButton.set_active()`. Note that, if you do this, and the state actually changes, it causes the "toggled" signal to be emitted.

8.2.1 ToggleButton Objects

class `Gtk.`**`ToggleButton`**`(`[*label*[, *stock*[, *use_underline*]]]`)`

The arguments are the same as for the `Gtk.Button` constructor.

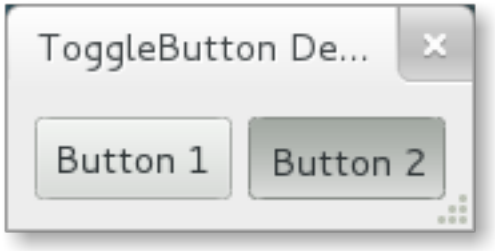
`get_active``()`

Returns the buttons current state. Returns `True` if the toggle button is pressed in and `False` if it is raised.

set_active(*is_active*)

Sets the status of the toggle button. Set to `True` if you want the button to be pressed in, and `False` to raise it. This action causes the "toggled" signal to be emitted.

8.2.2 Example




```
from gi.repository import Gtk

class ToggleButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="ToggleButton Demo")
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)

        button = Gtk.ToggleButton("Button 1")
        button.connect("toggled", self.on_button_toggled, "1")
        hbox.pack_start(button, True, True, 0)

        button = Gtk.ToggleButton("B_utton 2", use_underline=True)
        button.set_active(True)
        button.connect("toggled", self.on_button_toggled, "2")
        hbox.pack_start(button, True, True, 0)

    def on_button_toggled(self, button, name):
        if button.get_active():
            state = "on"
        else:
            state = "off"
        print "Button", name, "was turned", state

win = ToggleButtonWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

8.3 CheckButton

`Gtk.CheckButton` inherits from `Gtk.ToggleButton`. The only real difference between the two is `Gtk.CheckButton`'s appearance. A `Gtk.CheckButton` places a discrete `Gtk.ToggleButton` next

to a widget, (usually a Gtk.Label). The "toggled" signal, Gtk.ToggleButton.set_active() and Gtk.ToggleButton.get_active() are inherited.

8.3.1 CheckButton Objects

class Gtk.**CheckButton**([*label*[, *stock*[, *use_underline*]]])
Arguments are the same as for Gtk.Button.

8.4 RadioButton

Like checkboxes, radio buttons also inherit from Gtk.ToggleButton, but these work in groups, and only one Gtk.RadioButton in a group can be selected at any one time. Therefore, a Gtk.RadioButton is one way of giving the user a choice from many options.

Radio buttons can be created with one of the static methods Gtk.RadioButton.new_from_widget(), Gtk.RadioButton.new_with_label_from_widget() or Gtk.RadioButton.new_with_mnemonic_from_widget
The first radio button in a group will be created passing None as the *group* argument. In subsequent calls, the group you wish to add this button to should be passed as an argument.

When first run, the first radio button in the group will be active. This can be changed by calling Gtk.ToggleButton.set_active() with True as first argument.

Changing a Gtk.RadioButton's widget group after its creation can be achieved by calling Gtk.RadioButton.join_group().

8.4.1 RadioButton Objects

class Gtk.**RadioButton**

static new_from_widget(*group*)
Creates a new Gtk.RadioButton, adding it to the same group as the *group* widget. If *group* is None, a new group is created.

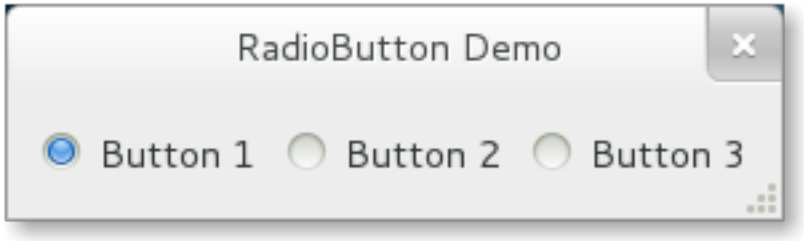
static new_with_label_from_widget(*group*, *label*)
Creates a new Gtk.RadioButton. The text of the label widget inside the button will be set to *label*. *group* is the same as for new_from_widget().

static new_with_mnemonic_from_widget(*group*, *label*)
Same as new_with_label_from_widget(), but underscores in *label* indicate the mnemonic for the button.

join_group(*group*)

Joins this radio button to the group of another Gtk.RadioButton object.

8.4.2 Example




```
from gi.repository import Gtk

class RadioButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="RadioButton Demo")
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)

        button1 = Gtk.RadioButton.new_with_label_from_widget(None, "Button 1")
        button1.connect("toggled", self.on_button_toggled, "1")
        hbox.pack_start(button1, False, False, 0)

        button2 = Gtk.RadioButton.new_from_widget(button1)
        button2.set_label("Button 2")
        button2.connect("toggled", self.on_button_toggled, "2")
        hbox.pack_start(button2, False, False, 0)

        button3 = Gtk.RadioButton.new_with_mnemonic_from_widget(button1, "B_utton 3")
        button3.connect("toggled", self.on_button_toggled, "3")
        hbox.pack_start(button3, False, False, 0)

    def on_button_toggled(self, button, name):
        if button.get_active():
            state = "on"
        else:
            state = "off"
        print "Button", name, "was turned", state

win = RadioButtonWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

8.5 LinkButton

A `Gtk.LinkButton` is a `Gtk.Button` with a hyperlink, similar to the one used by web browsers, which triggers an action when clicked. It is useful to show quick links to resources.

The URI bound to a `Gtk.LinkButton` can be set specifically using `Gtk.LinkButton.set_uri()`, and retrieved using `Gtk.LinkButton.get_uri()`.

8.5.1 LinkButton Objects

class Gtk.**LinkButton**(*uri*[, *label*])

uri is the address of the website which should be loaded. The label is set as the text which should be displayed. If it is set to None or omitted, the web address will be displayed instead.

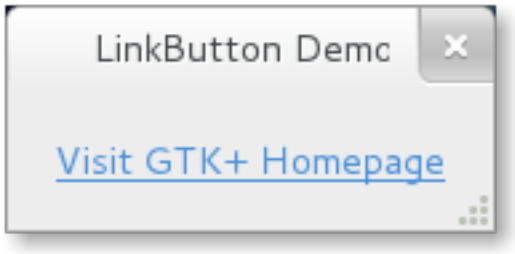
get_uri()

Retrieves the URI set using set_uri().

set_uri(*uri*)

Sets *uri* as the URI where this button points to. As a side-effect this unsets the 'visited' state of the button.

8.5.2 Example




```
from gi.repository import Gtk

class LinkButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="LinkButton Demo")
        self.set_border_width(10)

        button = Gtk.LinkButton("http://www.gtk.org", "Visit GTK+ Homepage")
        self.add(button)

win = LinkButtonWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

8.6 SpinButton

A Gtk.SpinButton is an ideal way to allow the user to set the value of some attribute. Rather than having to directly type a number into a Gtk.Entry, Gtk.SpinButton allows the user to click on one of two arrows to increment or decrement the displayed value. A value can still be typed in, with the bonus that it can be checked to ensure it is in a given range. The main properties of a Gtk.SpinButton are set through Gtk.Adjustment.

To change the value that Gtk.SpinButton is showing, use Gtk.SpinButton.set_value(). The value entered can either be an integer or float, depending on your requirements, use Gtk.SpinButton.get_value() or Gtk.SpinButton.get_value_as_int(), respectively.

When you allow the displaying of float values in the spin button, you may wish to adjust the number of decimal spaces displayed by calling Gtk.SpinButton.set_digits().

By default, Gtk.SpinButton accepts textual data. If you wish to limit this to numerical values only, call Gtk.SpinButton.set_numeric() with True as argument.

We can also adjust the update policy of Gtk.SpinButton. There are two options here; by default the spin button updates the value even if the data entered is invalid. Alternatively, we can set the policy to only update when the value entered is valid by calling Gtk.SpinButton.set_update_policy().

8.6.1 SpinButton Objects

class Gtk.**SpinButton**

set_adjustment(*adjustment*)
Replaces the Gtk.Adjustment associated with this spin button.

set_digits(*digits*)
Set the precision to be displayed by this spin button. Up to 20 digit precision is allowed.

set_increments(*step*, *page*)
Sets the step and page increments for this spin button. This affects how quickly the value changes when the spin button's arrows are activated.

set_value(*value*)
Sets the value of this spin button.

get_value()
Get the value of this spin button represented as an float.

get_value_as_int()
Get the value of this spin button represented as an integer.

set_numeric(*numeric*)
If *numeric* is False, non-numeric text can be typed into the spin button, else only numbers can be typed.

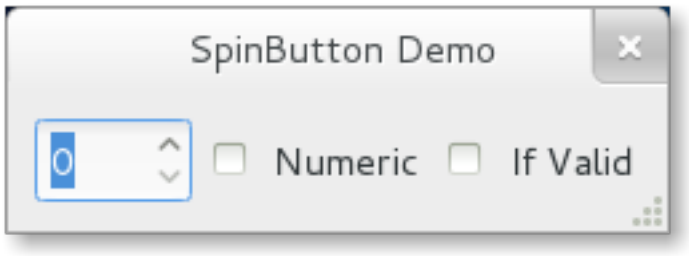
set_update_policy(*policy*)
Sets the update behavior of a spin button. This determines whether the spin button is always updated or only when a valid value is set. The *policy* argument can either be Gtk.SpinButtonUpdatePolicy.ALWAYS or Gtk.SpinButtonUpdatePolicy.IF_VALID.

8.6.2 Adjustment Objects

class Gtk.**Adjustment**(*value*, *lower*, *upper*, *step_increment*, *page_increment*, *page_size*)
The Gtk.Adjustment object represents a value which has an associated lower and upper bound, together with step and page increments, and a page size. It is used within several GTK+ widgets, including Gtk.SpinButton, Gtk.Viewport, and Gtk.Range.

value is the initial value, *lower* the minimum value, *upper* the maximum value, *step_increment* the step increment, *page_increment* the page increment, and *page_size* the page size.

8.6.3 Example




```
from gi.repository import Gtk

class SpinButtonWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="SpinButton Demo")
        self.set_border_width(10)

        hbox = Gtk.Box(spacing=6)
        self.add(hbox)

        adjustment = Gtk.Adjustment(0, 0, 100, 1, 10, 0)
        self.spinbutton = Gtk.SpinButton()
        self.spinbutton.set_adjustment(adjustment)
        hbox.pack_start(self.spinbutton, False, False, 0)

        check_numeric = Gtk.CheckButton("Numeric")
        check_numeric.connect("toggled", self.on_numeric_toggled)
        hbox.pack_start(check_numeric, False, False, 0)

        check_ifvalid = Gtk.CheckButton("If Valid")
        check_ifvalid.connect("toggled", self.on_ifvalid_toggled)
        hbox.pack_start(check_ifvalid, False, False, 0)

    def on_numeric_toggled(self, button):
        self.spinbutton.set_numeric(button.get_active())

    def on_ifvalid_toggled(self, button):
        if button.get_active():
            policy = Gtk.SpinButtonUpdatePolicy.IF_VALID
        else:
            policy = Gtk.SpinButtonUpdatePolicy.ALWAYS
        self.spinbutton.set_update_policy(policy)

win = SpinButtonWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

CHAPTER

NINE

PROGRESSBAR

The Gtk.ProgressBar is typically used to display the progress of a long running operation. It provides a visual clue that processing is underway. The Gtk.ProgressBar can be used in two different modes: *percentage mode* and *activity mode*.

When an application can determine how much work needs to take place (e.g. read a fixed number of bytes from a file) and can monitor its progress, it can use the Gtk.ProgressBar in *percentage mode* and the user sees a growing bar indicating the percentage of the work that has been completed. In this mode, the application is required to call Gtk.ProgressBar.set_fraction() periodically to update the progress bar, passing a float between 0 and 1 to provide the new percentage value.

When an application has no accurate way of knowing the amount of work to do, it can use *activity mode*, which shows activity by a block moving back and forth within the progress area. In this mode, the application is required to call Gtk.ProgressBar.pulse() periodically to update the progress bar. You can also choose the step size, with the Gtk.ProgressBar.set_pulse_step() method.

By default, Gtk.ProgressBar is horizontal and left-to-right, but you can change it to a vertical progress bar by using the Gtk.ProgressBar.set_orientation() method. Changing the direction the progress bar grows can be done using Gtk.ProgressBar.set_inverted(). Gtk.ProgressBar can also contain text which can be set by calling Gtk.ProgressBar.set_text() and Gtk.ProgressBar.set_show_text().

9.1 ProgressBar Objects

class Gtk.**ProgressBar**

set_fraction(*fraction*)
Causes the progress bar to "fill in" the given fraction of the bar. *fraction* should be between 0.0 and 1.0, inclusive.

set_pulse_step(*fraction*)
Sets the fraction of total progress bar length to move the bouncing block for each call to pulse().

pulse()
Indicates that some progress is made, but you don't know how much. Causes the progress bar to enter *activity mode*, where a block bounces back and forth. Each call to pulse() causes the block to move by a little bit (the amount of movement per pulse is determined by set_pulse_step()).

set_orientation(*orientation*)
Sets the orientation. *orientation* can be one of

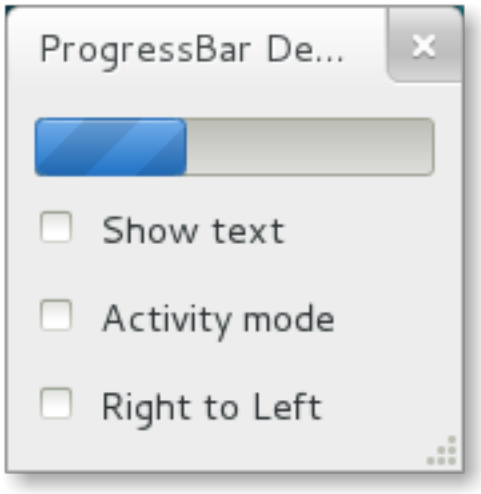
- Gtk.Orientation.HORIZONTAL
- Gtk.Orientation.VERTICAL

set_show_text(*show_text*)
Sets whether the progressbar will show text superimposed over the bar. The shown text is either the value of the "text" property or, if that is `None`, the "fraction" value, as a percentage.

set_text(*text*)
Causes the given *text* to appear superimposed on the progress bar.

set_inverted(*inverted*)
Progress bars normally grow from top to bottom or left to right. Inverted progress bars grow in the opposite direction.

9.2 Example




```
from gi.repository import Gtk, GObject

class ProgressBarWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="ProgressBar Demo")
        self.set_border_width(10)

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
        self.add(vbox)

        self.progressbar = Gtk.ProgressBar()
        vbox.pack_start(self.progressbar, True, True, 0)

        button = Gtk.CheckButton("Show text")
        button.connect("toggled", self.on_show_text_toggled)
        vbox.pack_start(button, True, True, 0)

        button = Gtk.CheckButton("Activity mode")
        button.connect("toggled", self.on_activity_mode_toggled)
        vbox.pack_start(button, True, True, 0)

        button = Gtk.CheckButton("Right to Left")
        button.connect("toggled", self.on_right_to_left_toggled)
        vbox.pack_start(button, True, True, 0)
```

```
26
27          self.timeout_id = GObject.timeout_add(50, self.on_timeout, None)
28          self.activity_mode = False
29
30      def on_show_text_toggled(self, button):
31          show_text = button.get_active()
32          if show_text:
33              text = "some text"
34          else:
35              text = None
36          self.progressbar.set_text(text)
37          self.progressbar.set_show_text(show_text)
38
39      def on_activity_mode_toggled(self, button):
40          self.activity_mode = button.get_active()
41          if self.activity_mode:
42              self.progressbar.pulse()
43          else:
44              self.progressbar.set_fraction(0.0)
45
46      def on_right_to_left_toggled(self, button):
47          value = button.get_active()
48          self.progressbar.set_inverted(value)
49
50      def on_timeout(self, user_data):
51          """
52          Update value on the progress bar
53          """
54          if self.activity_mode:
55              self.progressbar.pulse()
56          else:
57              new_value = self.progressbar.get_fraction() + 0.01
58
59              if new_value > 1:
60                  new_value = 0
61
62              self.progressbar.set_fraction(new_value)
63
64          # As this is a timeout function, return True so that it
65          # continues to get called
66          return True
67
68  win = ProgressBarWindow()
69  win.connect("delete-event", Gtk.main_quit)
70  win.show_all()
71  Gtk.main()
```

CHAPTER

TEN

SPINNER

The `Gtk.Spinner` displays an icon-size spinning animation. It is often used as an alternative to a `GtkProgressBar` for displaying indefinite activity, instead of actual progress.

To start the animation, use `Gtk.Spinner.start()`, to stop it use `Gtk.Spinner.stop()`.

10.1 Spinner Objects

class `Gtk.`**`Spinner`**

`start``()`
Starts the animation of the spinner.

`stop``()`
Stops the animation of the spinner.

10.2 Example


```
from gi.repository import Gtk

class SpinnerAnimation(Gtk.Window):

    def __init__(self):

        Gtk.Window.__init__(self, title="Spinner")
```

```
8           self.set_border_width(3)
9           self.connect("delete-event", Gtk.main_quit)
10
11          self.button = Gtk.ToggleButton("Start Spinning")
12          self.button.connect("toggled", self.on_button_toggled)
13          self.button.set_active(False)
14
15          self.spinner = Gtk.Spinner()
16
17          self.table = Gtk.Table(3, 2, True)
18          self.table.attach(self.button, 0, 2, 0, 1)
19          self.table.attach(self.spinner, 0, 2, 2, 3)
20
21          self.add(self.table)
22          self.show_all()
23
24      def on_button_toggled(self, button):
25
26          if button.get_active():
27              self.spinner.start()
28              self.button.set_label("Stop Spinning")
29
30          else:
31              self.spinner.stop()
32              self.button.set_label("Start Spinning")
33
34
35  myspinner = SpinnerAnimation()
36
37  Gtk.main()
```

CHAPTER

ELEVEN

TREE AND LIST WIDGETS

A Gtk.TreeView and its associated widgets are an extremely powerful way of displaying data. They are used in conjunction with a Gtk.ListStore or Gtk.TreeStore and provide a way of displaying and manipulating data in many ways, including:

- Automatically updates when data added, removed or edited
- Drag and drop support
- Sorting of data
- Support embedding widgets such as check boxes, progress bars, etc.
- Reorderable and resizable columns
- Filtering of data

With the power and flexibility of a Gtk.TreeView comes complexity. It is often difficult for beginner developers to be able to utilize correctly due to the number of methods which are required.

11.1 The Model

Each Gtk.TreeView has an associated Gtk.TreeModel, which contains the data displayed by the TreeView. Each Gtk.TreeModel can be used by more than one Gtk.TreeView. For instance, this allows the same underlying data to be displayed and edited in 2 different ways at the same time. Or the 2 Views might display different columns from the same Model data, in the same way that 2 SQL queries (or "views") might show different fields from the same database table.

Although you can theoretically implement your own Model, you will normally use either the Gtk.ListStore or Gtk.TreeStore model classes. Gtk.ListStore contains simple rows of data, and each row has no children, whereas Gtk.TreeStore contains rows of data, and each row may have child rows.

When constructing a model you have to specify the data types for each column the model holds.

```
store = Gtk.ListStore(str, str, float)
```

This creates a list store with three columns, two string columns, and a float column.

Adding data to the model is done using Gtk.ListStore.append() or Gtk.TreeStore.append(), depending upon which sort of model was created.

```
treeiter = store.append(["The Art of Computer Programming", "Donald E. Knuth", 25.46])
```

Both methods return a Gtk.TreeIter instance, which points to the location of the newly inserted row. You can retrieve a Gtk.TreeIter by calling Gtk.TreeModel.get_iter().

Once, data has been inserted you can retrieve or modify data using the tree iter and column index.

```
print store[treeiter][2] # Prints value of third column
store[treeiter][2] = 42.15
```

As with Python's built-in `list` object you can use `len()` to get the number of rows and use slices to retrieve or set values.

```
# Print number of rows
print len(store)
# Print all but first column
print store[treeiter][1:]
# Print last column
print store[treeiter][-1]
# Set first two columns
store[treeiter][:2] = ["Donald Ervin Knuth", 41.99]
```

Iterating over all rows of a tree model is very simple as well.

```
for row in store:
    # Print values of all columns
    print row[:]
```

Keep in mind, that if you use `Gtk.TreeStore`, the above code will only iterate over the rows of the top level, but not the children of the nodes. To iterate over all rows and its children, use the `print_tree_store` function.

```
def print_tree_store(store):
    rootiter = store.get_iter_first()
    print_rows(store, rootiter, "")

def print_rows(store, treeiter, indent):
    while treeiter != None:
        print indent + str(store[treeiter][:])
        if store.iter_has_child(treeiter):
            childiter = store.iter_children(treeiter)
            print_rows(store, childiter, indent + "\t")
        treeiter = store.iter_next(treeiter)
```

Apart from accessing values stored in a `Gtk.TreeModel` with the list-like method mentioned above, it is also possible to either use `Gtk.TreeIter` or `Gtk.TreePath` instances. Both reference a particular row in a tree model. One can convert a path to an iterator by calling `Gtk.TreeModel.get_iter()`. As `Gtk.ListStore` contains only one level, i.e. nodes do not have any child nodes, a path is essentially the index of the row you want to access.

```
# Get path pointing to 6th row in list store
path = Gtk.TreePath(5)
treeiter = liststore.get_iter(path)
# Get value at 2nd column
value = liststore.get_value(treeiter, 1)
```

In the case of `Gtk.TreeStore`, a path is a list of indexes or a string. The string form is a list of numbers separated by a colon. Each number refers to the offset at that level. Thus, the path "0" refers to the root node and the path "2:4" refers to the fifth child of the third node.

```
# Get path pointing to 5th child of 3rd row in tree store
path = Gtk.TreePath([2, 4])
treeiter = treestore.get_iter(path)
# Get value at 2nd column
value = treestore.get_value(treeiter, 1)
```

Instances of `Gtk.TreePath` can be accessed like lists, i.e. `len(treepath)` returns the depth of the item `treepath` is pointing to, and `treepath[i]` returns the child's index on the *i*-th level.

11.1.1 TreeModel Objects

class `Gtk.`**`TreeModel`**

`get_iter`(*path*)
Returns a `Gtk.TreeIter` instance pointing to *path*.

path is expected to be a colon separated list of numbers or a tuple. For example, the string "10:4:0" or tuple (10, 4, 0) would create a path of depth 3 pointing to the 11th child of the root node, the 5th child of that 11th child, and the 1st child of that 5th child.

`iter_next`(*treeiter*)
Returns a `Gtk.TreeIter` instance pointing the node following *treeiter* at the current level or `None` if there is no next iter.

`iter_previous`(*treeiter*)
Returns a `Gtk.TreeIter` instance pointing the node previous to *treeiter* at the current level or `None` if there is no previous iter.

`iter_has_child`(*treeiter*)
Returns `True` if *treeiter* has children, `False` otherwise.

`iter_children`(*treeiter*)
Returns a `Gtk.TreeIter` instance pointing to the first child of *treeiter* or `None` if *treeiter* has no children.

`get_iter_first`()
Returns a `Gtk.TreeIter` instance pointing to the first iterator in the tree (the one at the path "0") or `None` if the tree is empty.

11.1.2 ListStore Objects

class `Gtk.`**`ListStore`**(*data_type*[, ...])
Creates a new `Gtk.ListStore` with the specified column data types. Each row added to the list store will have an item in each column.

Supported data types are the standard Python ones and GTK+ types:

- str, int, float, long, bool, object
- GObject.GObject

`append`([*row*])

Appends a new row to this list store.

row is a list of values for each column, i.e. `len(store) == len(row)`. If *row* is omitted or `None`, an empty row will be appended.

Returns a `Gtk.TreeIter` pointing to the appended row.

11.1.3 TreeStore Objects

class `Gtk.`**`TreeStore`**(*data_type*[, ...])
Arguments are the same as for the `Gtk.ListStore` constructor.

`append`(*parent*[, *row*])

Appends a new row to this tree store. *parent* must be a valid `Gtk.TreeIter`. If *parent* is not `None`, then it will append the new row after the last child of *parent*, otherwise it will append a row to the top level.

row is a list of values for each column, i.e. `len(store) == len(row)`. If *row* is omitted or `None`, an empty row will be appended.

Returns a `Gtk.TreeIter` pointing to the appended row.

11.1.4 TreePath Objects

class `Gtk.`**`TreePath`**(*path*)
Construct a `Gtk.TreePath` pointing to the node specified by *path*.

If *path* is a string it is expected to be a colon separated list of numbers. For example, the string "10:4:0" would create a path of depth 3 pointing to the 11th child of the root node, the 5th child of that 11th child, and the 1st child of that 5th child.

If *path* is a list or a tuple it is expected to contain the indexes of the nodes. Referring to the above mentioned example, the expression `Gtk.TreePath("10:4:0")` is equivalent to `Gtk.TreePath([10, 4, 3])`.

11.2 The View

While there are several different models to choose from, there is only one view widget to deal with. It works with either the list or the tree store. Setting up a `Gtk.TreeView` is not a difficult matter. It needs a `Gtk.TreeModel` to know where to retrieve its data from, either by passing it to the `Gtk.TreeView` constructor, or by calling `Gtk.TreeView.set_model()`.

```
tree = Gtk.TreeView(store)
```

Once the `Gtk.TreeView` widget has a model, it will need to know how to display the model. It does this with columns and cell renderers.

Cell renderers are used to draw the data in the tree model in a way. There are a number of cell renderers that come with GTK+, for instance `Gtk.CellRendererText`, `Gtk.CellRendererPixbuf` and `Gtk.CellRendererToggle`. In addition, it is relatively easy to write a custom renderer yourself.

A `Gtk.TreeViewColumn` is the object that `Gtk.TreeView` uses to organize the vertical columns in the tree view. It needs to know the name of the column to label for the user, what type of cell renderer to use, and which piece of data to retrieve from the model for a given row.

```
renderer = Gtk.CellRendererText()
column = Gtk.TreeViewColumn("Title", renderer, text=0)
tree.append_column(column)
```

To render more than one model column in a view column, you need to create a `Gtk.TreeViewColumn` instance and use `Gtk.TreeViewColumn.pack_start()` to add the model columns to it.

```
column = Gtk.TreeViewColumn("Title and Author")

title = Gtk.CellRendererText()
```

```
author = Gtk.CellRendererText()

column.pack_start(title, True)
column.pack_start(author, True)

column.add_attribute(title, "text", 0)
column.add_attribute(author, "text", 1)

tree.append_column(column)
```

11.2.1 TreeView Objects

class `Gtk.`**`TreeView`**(*[treemodel]*)

Creates a new `Gtk.TreeView` widget with the model initialized to *treemodel*. *treemodel* must be a class implementing `Gtk.TreeModel`, such as `Gtk.ListStore` or `Gtk.TreeStore`. If *treemodel* is omitted or `None`, the model remains unset and you have to call `set_model()` later.

`set_model`(*model*)

Sets the model for this tree view. If this tree view already has a model set, it will remove it before setting the new model. If *model* is `None`, then it will unset the old model.

`get_model`()

Returns the model this tree view is based on. Returns `None` if the model is unset.

`append_column`(*column*)

Appends *column* to the list of columns.

`get_selection`()

Gets the `Gtk.TreeSelection` associated with this tree view.

`enable_model_drag_source`(*start_button_mask*, *targets*, *actions*)

Arguments are the same as `Gtk.Widget.drag_source_set()`

`enable_model_dest_source`(*targets*, *actions*)

Arguments are the same as `Gtk.Widget.drag_dest_set()`

11.2.2 TreeViewColumn Objects

class `Gtk.`**`TreeViewColumn`**(*label*[, *renderer*[, ***kwargs*]])

Creates a new `Gtk.TreeViewColumn`.

renderer is expected to be a `Gtk.CellRenderer` instance, and *kwargs* key-value pairs specifying the default values of *renderer*'s properties. This is equivalent to calling `pack_start()` and `add_attribute()` for each key-value pair in *kwargs*.

If *renderer* is omitted, you have to call `pack_start()` or `pack_end()` yourself.

`add_attribute`(*renderer*, *attribute*, *value*)

Adds an attribute mapping to this column.

attribute is the parameter on *renderer* to be set from the value. So for example if column 2 of the model contains strings, you could have the "text" attribute of a `Gtk.CellRendererText` get its values from column 2.

`pack_start`(*renderer*, *expand*)

Packs the *renderer* into the beginning of this column. If expand is `False`, then the *renderer* is allocated no more space than it needs. Any unused space is divided evenly between cells for which expand is `True`.

pack_end(*renderer*, *expand*)
Adds the *renderer* to end of this column. If expand is `False`, then the *renderer* is allocated no more space than it needs. Any unused space is divided evenly between cells for which expand is `True`.

set_sort_column_id(*sort_column_id*)
Sets the column of the model by which this column (of the view) should be sorted. This also makes the columnd header clickable.

get_sort_column_id()
Gets the column id set by `Gtk.TreeViewColumn.set_sort_column_id()`

set_sort_indicator(*setting*)
Sets whether a little arrow is displayed in the column header to in

setting can either be `True` (indicator is shown) or `False`

get_sort_indicator()
Gets the value set by `Gtk.TreeViewColumn.set_sort_indicator()`

set_sort_order(*order*)
Changes the order by which the column is sorted.

order can either be `Gtk.SortType.ASCENDING` or `Gtk.SortType.DESCENDING`.

get_sort_order()
Gets the sort order set by `Gtk.TreeViewColumn.set_sort_order()`

11.3 The Selection

Most applications will need to not only deal with displaying data, but also receiving input events from users. To do this, simply get a reference to a selection object and connect to the "changed" signal.

```
select = tree.get_selection()
select.connect("changed", on_tree_selection_changed)
```

Then to retrieve data for the row selected:

```
def on_tree_selection_changed(selection):
    model, treeiter = selection.get_selected()
    if treeiter != None:
        print "You selected", model[treeiter][0]
```

You can control what selections are allowed by calling `Gtk.TreeSelection.set_mode()`. `Gtk.TreeSelection.get_selected()` does not work if the selection mode is set to `Gtk.SelectionMode.MULTIPLE`, use `Gtk.TreeSelection.get_selected_rows()` instead.

11.3.1 TreeSelection Objects

class `Gtk.`**TreeSelection**

set_mode(*type*)
Where type is one of

- `Gtk.SelectionMode.NONE`: No selection is possible
- `Gtk.SelectionMode.SINGLE`: Zero or one element may be selected

- `Gtk.SelectionMode.BROWSE`: Exactly one element is selected. In some circumstances, such as initially or during a search operation, it's possible for no element to be selected. What is really enforced is that the user can't deselect a currently selected element except by selecting another element.
- `Gtk.SelectionMode.MULTIPLE`: Any number of elements may be selected. Clicks toggle the state of an item. Any number of elements may be selected. The Ctrl key may be used to enlarge the selection, and Shift key to select between the focus and the child pointed to. Some widgets may also allow Click-drag to select a range of elements.

get_selected()
Returns a tuple `(model, treeiter)`, where *model* is the current model and *treeiter* a `Gtk.TreeIter` pointing to the currently selected row. *treeiter* is `None` if no rows are selected.

This function will not work if the mode of this selection is `Gtk.SelectionMode.MULTIPLE`.

get_selected_rows()
Returns a list of `Gtk.TreePath` instances of all selected rows.

11.4 Sorting

Sorting is an important feature for tree views and is supported by the standard tree models (`Gtk.TreeStore` and `Gtk.ListStore`), which implement the `Gtk.TreeSortable` interface.

11.4.1 Sorting by clicking on columns

A column of a `Gtk.TreeView` can easily made sortable with a call to `Gtk.TreeViewColumn.set_sort_column_id()`. Afterwards the column can be sorted by clicking on its header.

First we need a simple `Gtk.TreeView` and a `Gtk.ListStore` as a model.

```
model = Gtk.ListStore(str)
model.append(["Benjamin"])
model.append(["Charles"])
model.append(["alfred"])
model.append(["Alfred"])
model.append(["David"])
model.append(["charles"])
model.append(["david"])
model.append(["benjamin"])

treeView = Gtk.TreeView(model)

cellRenderer = Gtk.CellRendererText()
column = Gtk.TreeViewColumn("Title", renderer, text=0)
```

The next step is to enable sorting. Note that the *column_id* (`0` in the example) refers to the column of the model and **not** to the TreeView's column.

```
column.set_sort_column_id(0)
```

11.4.2 Setting a custom sort function

It is also possible to set a custom comparison function in order to change the sorting behaviour. As an example we will create a comparison function that sorts case-sensitive. In the example above the sorted list looked like:

```
alfred
Alfred
benjamin
Benjamin
charles
Charles
david
David
```

The case-sensitive sorted list will look like:

```
Alfred
Benjamin
Charles
David
alfred
benjamin
charles
david
```

First of all a comparison function is needed. This function gets two rows and has to return a negative integer if the first one should come before the second one, zero if they are equal and a positive integer if the second one should come before the second one.

```
def compare(model, row1, row2, user_data):
    sort_column, _ = model.get_sort_column_id()
    value1 = model.get_value(row1, sort_column)
    value2 = model.get_value(row2, sort_column)
    if value1 < value2:
        return -1
    elif value1 == value2:
        return 0
    else:
        return 1
```

Then the sort function has to be set by `Gtk.TreeSortable.set_sort_func()`.

```
model.set_sort_func(0, compare, None)
```

11.4.3 TreeSortable objects

class `Gtk.`**`TreeSortable`**

`set_sort_column_id`(*sort_column_id*, *order*)
Sets the current sort column to *sort_column_id*.

order can either be `Gtk.SortType.ASCENDING` or `Gtk.SortType.DESCENDING`.

`get_sort_column_id`()
Returns a tuple consisting of the current sort column and order.

`set_sort_func`(*sort_column_id*, *sort_func*, *user_data*)
Sets the comparison function used when sorting by the colum *sort_column_id*.

user_data gets passed to *sort_func*.

sort_func is a function with the signature `sort_func(model, iter1, iter2, user_data)` and should return a negative integer if *iter1* sorts before *iter2*, zero if they are equal and a positive integer if *iter2* sorts before *iter1*.

set_default_sort_func(*sort_func*, *user_data*)
See Gtk.TreeSortable.set_sort_func(). This sets the comparison function that is used when sorting by the default sort column

CHAPTER

TWELVE

CELLRENDERERS

`Gtk.CellRenderer` widgets are used to display information within widgets such as the `Gtk.TreeView` or `Gtk.ComboBox`. They work closely with the associated widgets and are very powerful, with lots of configuration options for displaying a large amount of data in different ways. There are seven `Gtk.CellRenderer` widgets which can be used for different purposes:

- `Gtk.CellRendererText`
- `Gtk.CellRendererToggle`
- `Gtk.CellRendererPixbuf`
- `Gtk.CellRendererCombo`
- `Gtk.CellRendererProgress`
- `Gtk.CellRendererSpinner`
- `Gtk.CellRendererSpin`
- `Gtk.CellRendererAccel`

12.1 CellRendererText

A `Gtk.CellRendererText` renders a given text in its cell, using the font, color and style information provided by its properties. The text will be ellipsized if it is too long and the "ellipsize" property allows it.

By default, text in `Gtk.CellRendererText` widgets is not editable. This can be changed by setting the value of the "editable" property to `True`:

```
cell.set_property("editable", True)
```

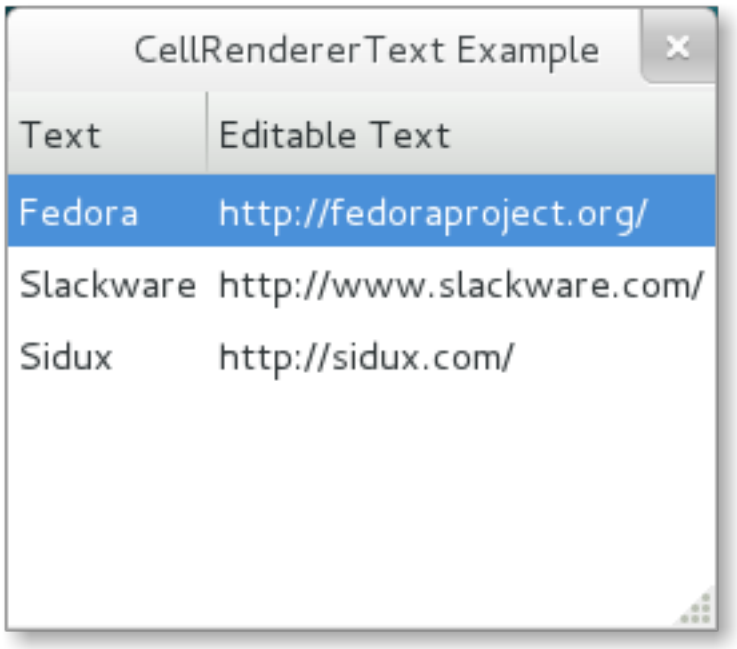
You can then connect to the "edited" signal and update your `Gtk.TreeModel` accordingly.

12.1.1 CellRendererText Objects

class `Gtk.`**`CellRendererText`**

Creates a new `Gtk.CellRendererText` instance. Adjust how text is drawn using object properties. Also, with `Gtk.TreeViewColumn`, you can bind a property to a value in a `GtkTreeModel`. For example, you can bind the "text" property on the cell renderer to a string value in the model, thus rendering a different string in each row of the `Gtk.TreeView`.

12.1.2 Example




```
from gi.repository import Gtk

class CellRendererTextWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererText Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, str)
        self.liststore.append(["Fedora", "http://fedoraproject.org/"])
        self.liststore.append(["Slackware", "http://www.slackware.com/"])
        self.liststore.append(["Sidux", "http://sidux.com/"])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_editabletext = Gtk.CellRendererText()
        renderer_editabletext.set_property("editable", True)

        column_editabletext = Gtk.TreeViewColumn("Editable Text",
            renderer_editabletext, text=1)
        treeview.append_column(column_editabletext)

        renderer_editabletext.connect("edited", self.text_edited)

        self.add(treeview)

    def text_edited(self, widget, path, text):
        self.liststore[path][1] = text
```

```
34
35 win = CellRendererTextWindow()
36 win.connect("delete-event", Gtk.main_quit)
37 win.show_all()
38 Gtk.main()
```

12.2 CellRendererToggle

`Gtk.CellRendererToggle` renders a toggle button in a cell. The button is drawn as a radio- or checkbutton, depending on the "radio" property. When activated, it emits the "toggled" signal.

As a `Gtk.CellRendererToggle` can have two states, active and not active, you most likely want to bind the "active" property on the cell renderer to a boolean value in the model, thus causing the check button to reflect the state of the model.

12.2.1 CellRendererToggle Objects

class `Gtk.`**`CellRendererToggle`**

Creates a new `Gtk.CellRendererToggle` instance.

`set_active`(*setting*)

Activates or deactivates a cell renderer.

`get_active`()

Returns whether the cell renderer is active.

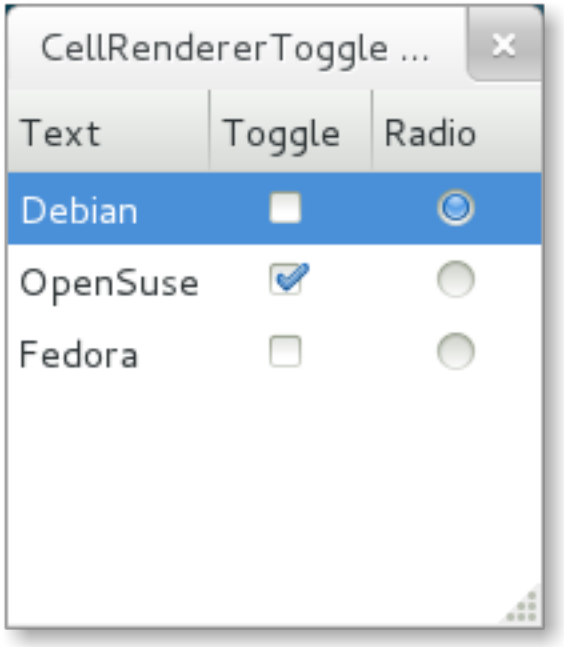
`set_radio`(*radio*)

If *radio* is `True`, the cell renderer renders a radio toggle (i.e. a toggle in a group of mutually-exclusive toggles). If `False`, it renders a check toggle (a standalone boolean option).

`get_radio`()

Returns whether we're rendering radio toggles rather than checkboxes.

12.2.2 Example




```
from gi.repository import Gtk

class CellRendererToggleWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererToggle Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, bool, bool)
        self.liststore.append(["Debian", False, True])
        self.liststore.append(["OpenSuse", True, False])
        self.liststore.append(["Fedora", False, False])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_toggle = Gtk.CellRendererToggle()
        renderer_toggle.connect("toggled", self.on_cell_toggled)

        column_toggle = Gtk.TreeViewColumn("Toggle", renderer_toggle, active=1)
        treeview.append_column(column_toggle)

        renderer_radio = Gtk.CellRendererToggle()
        renderer_radio.set_radio(True)
        renderer_radio.connect("toggled", self.on_cell_radio_toggled)

        column_radio = Gtk.TreeViewColumn("Radio", renderer_radio, active=2)
        treeview.append_column(column_radio)

```

```
34          self.add(treeview)
35
36      def on_cell_toggled(self, widget, path):
37          self.liststore[path][1] = not self.liststore[path][1]
38
39      def on_cell_radio_toggled(self, widget, path):
40          selected_path = Gtk.TreePath(path)
41          for row in self.liststore:
42              row[2] = (row.path == selected_path)
43
44  win = CellRendererToggleWindow()
45  win.connect("delete-event", Gtk.main_quit)
46  win.show_all()
47  Gtk.main()
```

12.3 CellRendererPixbuf

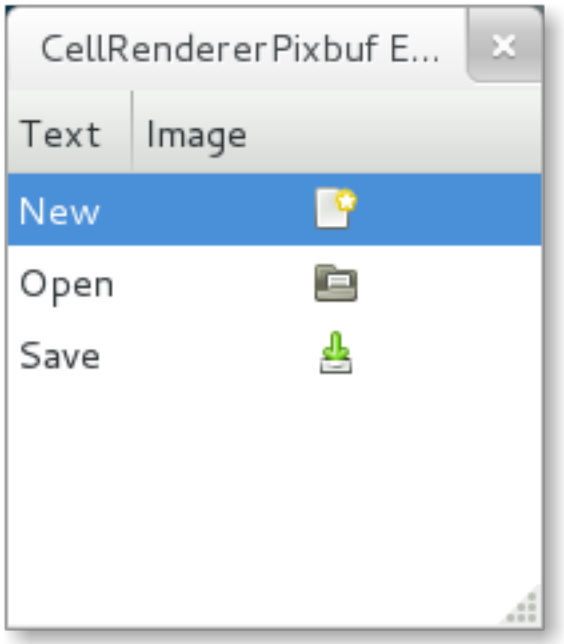
A `Gtk.CellRendererPixbuf` can be used to render an image in a cell. It allows to render either a given `Gdk.Pixbuf` (set via the "pixbuf" property) or a *stock item* (set via the "stock-id" property).

12.3.1 CellRendererPixbuf Objects

class `Gtk.`**`CellRendererPixbuf`**

Creates a new `Gtk.CellRendererPixbuf`. Adjust rendering parameters using object properties. For example, you can bind the "pixbuf" or "stock-id" property on the cell renderer to a pixbuf value in the model, thus rendering a different image in each row of the `Gtk.TreeView`.

12.3.2 Example



```
from gi.repository import Gtk

class CellRendererPixbufWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererPixbuf Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, str)
        self.liststore.append(["New", Gtk.STOCK_NEW])
        self.liststore.append(["Open", Gtk.STOCK_OPEN])
        self.liststore.append(["Save", Gtk.STOCK_SAVE])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_pixbuf = Gtk.CellRendererPixbuf()

        column_pixbuf = Gtk.TreeViewColumn("Image", renderer_pixbuf, stock_id=1)
        treeview.append_column(column_pixbuf)

        self.add(treeview)

win = CellRendererPixbufWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

12.4 CellRendererCombo

`Gtk.CellRendererCombo` renders text in a cell like `Gtk.CellRendererText` from which it is derived. But while the latter offers a simple entry to edit the text, `Gtk.CellRendererCombo` offers a `Gtk.ComboBox` widget to edit the text. The values to display in the combo box are taken from the `Gtk.TreeModel` specified in the "model" property.

The combo cell renderer takes care of adding a text cell renderer to the combo box and sets it to display the column specified by its "text-column" property.

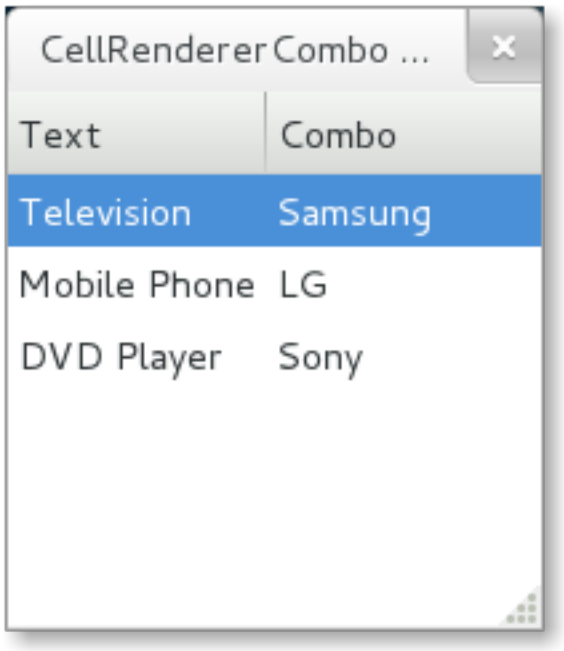
A `Gtk.CellRendererCombo` can operate in two modes. It can be used with and without an associated `Gtk.Entry` widget, depending on the value of the "has-entry" property.

12.4.1 CellRendererCombo Objects

class `Gtk.`**`CellRendererCombo`**

Creates a new `Gtk.CellRendererCombo`. Adjust how text is drawn using object properties. For example, you can bind the "text" property on the cell renderer to a string value in the model, thus rendering a different string in each row of the `Gtk.TreeView`.

12.4.2 Example




```
from gi.repository import Gtk

class CellRendererComboWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererCombo Example")

        self.set_default_size(200, 200)

        liststore_manufacturers = Gtk.ListStore(str)
        manufacturers = ["Sony", "LG", "Panasonic", "Toshiba", "Nokia", "Samsung"]
        for item in manufacturers:
            liststore_manufacturers.append([item])

        self.liststore_hardware = Gtk.ListStore(str, str)
        self.liststore_hardware.append(["Television", "Samsung"])
        self.liststore_hardware.append(["Mobile Phone", "LG"])
        self.liststore_hardware.append(["DVD Player", "Sony"])

        treeview = Gtk.TreeView(model=self.liststore_hardware)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_combo = Gtk.CellRendererCombo()
        renderer_combo.set_property("editable", True)
        renderer_combo.set_property("model", liststore_manufacturers)
        renderer_combo.set_property("text-column", 0)
        renderer_combo.set_property("has-entry", False)
        renderer_combo.connect("edited", self.on_combo_changed)

        column_combo = Gtk.TreeViewColumn("Combo", renderer_combo, text=1)
```

```
34          treeview.append_column(column_combo)
35
36          self.add(treeview)
37
38      def on_combo_changed(self, widget, path, text):
39          self.liststore_hardware[path][1] = text
40
41  win = CellRendererComboWindow()
42  win.connect("delete-event", Gtk.main_quit)
43  win.show_all()
44  Gtk.main()
```

12.5 CellRendererProgress

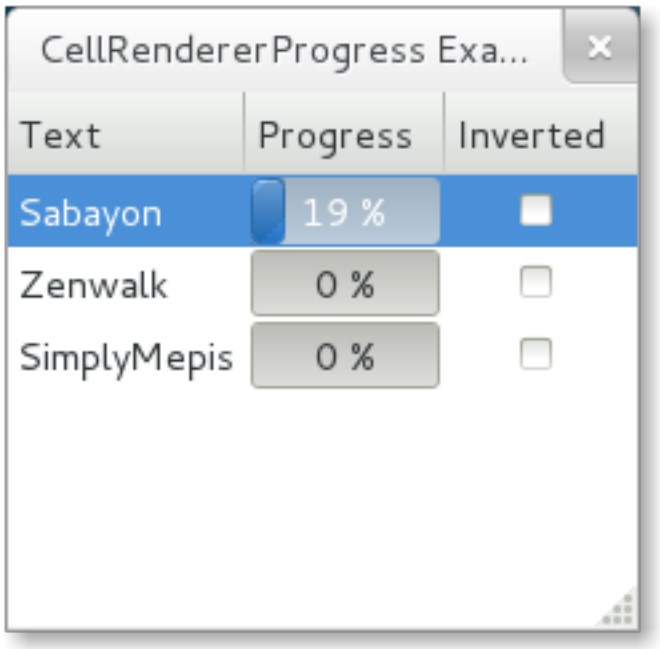
`Gtk.CellRendererProgress` renders a numeric value as a progress bar in a cell. Additionally, it can display a text on top of the progress bar.

The percentage value of the progress bar can be modified by changing the "value" property. Similar to `Gtk.ProgressBar`, you can enable the *activity mode* by incrementing the "pulse" property instead of the "value" property.

12.5.1 CellRendererProgress Objects

class `Gtk.`**`CellRendererProgress`**
Creates a new `Gtk.CellRendererProgress`.

12.5.2 Example



```
from gi.repository import Gtk, GObject

class CellRendererProgressWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererProgress Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, int, bool)
        self.current_iter = self.liststore.append(["Sabayon", 0, False])
        self.liststore.append(["Zenwalk", 0, False])
        self.liststore.append(["SimplyMepis", 0, False])

        treeview = Gtk.TreeView(model=self.liststore)

        renderer_text = Gtk.CellRendererText()
        column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
        treeview.append_column(column_text)

        renderer_progress = Gtk.CellRendererProgress()
        column_progress = Gtk.TreeViewColumn("Progress", renderer_progress,
            value=1, inverted=2)
        treeview.append_column(column_progress)

        renderer_toggle = Gtk.CellRendererToggle()
        renderer_toggle.connect("toggled", self.on_inverted_toggled)
        column_toggle = Gtk.TreeViewColumn("Inverted", renderer_toggle,
            active=2)
        treeview.append_column(column_toggle)

        self.add(treeview)

        self.timeout_id = GObject.timeout_add(100, self.on_timeout, None)

    def on_inverted_toggled(self, widget, path):
        self.liststore[path][2] = not self.liststore[path][2]

    def on_timeout(self, user_data):
        new_value = self.liststore[self.current_iter][1] + 1
        if new_value > 100:
            self.current_iter = self.liststore.iter_next(self.current_iter)
            if self.current_iter == None:
                self.reset_model()
            new_value = self.liststore[self.current_iter][1] + 1

        self.liststore[self.current_iter][1] = new_value
        return True

    def reset_model(self):
        for row in self.liststore:
            row[1] = 0
        self.current_iter = self.liststore.get_iter_first()

win = CellRendererProgressWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

12.6 CellRendererSpin

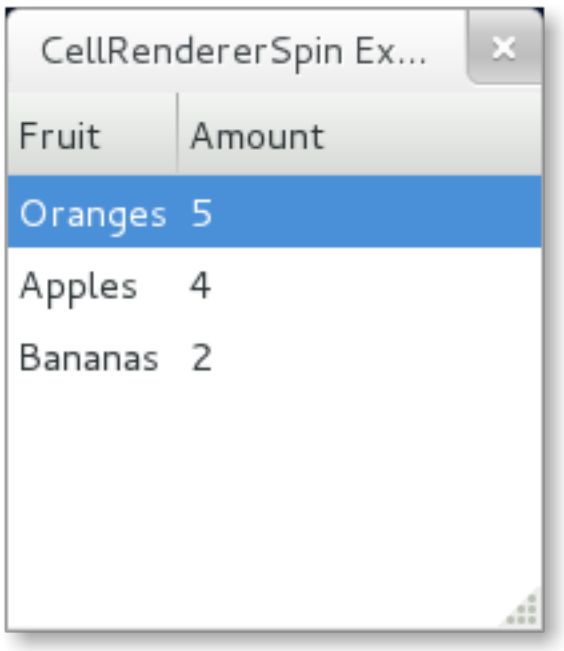
`Gtk.CellRendererSpin` renders text in a cell like `Gtk.CellRendererText` from which it is derived. But while the latter offers a simple entry to edit the text, `Gtk.CellRendererSpin` offers a `Gtk.SpinButton` widget. Of course, that means that the text has to be parseable as a floating point number.

The range of the spinbutton is taken from the adjustment property of the cell renderer, which can be set explicitly or mapped to a column in the tree model, like all properties of cell renders. `Gtk.CellRendererSpin` also has properties for the climb rate and the number of digits to display.

12.6.1 CellRendererSpin Objects

class `Gtk.`**`CellRendererSpin`**
Creates a new `Gtk.CellRendererSpin`.

12.6.2 Example




```
from gi.repository import Gtk

class CellRendererSpinWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="CellRendererSpin Example")

        self.set_default_size(200, 200)

        self.liststore = Gtk.ListStore(str, int)
        self.liststore.append(["Oranges", 5])
        self.liststore.append(["Apples", 4])
        self.liststore.append(["Bananas", 2])

```

```
15          treeview = Gtk.TreeView(model=self.liststore)
16
17          renderer_text = Gtk.CellRendererText()
18          column_text = Gtk.TreeViewColumn("Fruit", renderer_text, text=0)
19          treeview.append_column(column_text)
20
21          renderer_spin = Gtk.CellRendererSpin()
22          renderer_spin.connect("edited", self.on_amount_edited)
23          renderer_spin.set_property("editable", True)
24
25          adjustment = Gtk.Adjustment(0, 0, 100, 1, 10, 0)
26          renderer_spin.set_property("adjustment", adjustment)
27
28          column_spin = Gtk.TreeViewColumn("Amount", renderer_spin, text=1)
29          treeview.append_column(column_spin)
30
31          self.add(treeview)
32
33      def on_amount_edited(self, widget, path, value):
34          self.liststore[path][1] = int(value)
35
36  win = CellRendererSpinWindow()
37  win.connect("delete-event", Gtk.main_quit)
38  win.show_all()
39  Gtk.main()
```

CHAPTER

THIRTEEN

COMBOBOX

A `Gtk.ComboBox` allows for the selection of an item from a dropdown menu. They are preferable to having many radio buttons on screen as they take up less room. If appropriate, it can show extra information about each item, such as text, a picture, a checkbox, or a progress bar.

`Gtk.ComboBox` is very similar to `Gtk.TreeView`, as both use the model-view pattern; the list of valid choices is specified in the form of a tree model, and the display of the choices can be adapted to the data in the model by using *cell renderers*. If the combo box contains a large number of items, it may be better to display them in a grid rather than a list. This can be done by calling `Gtk.ComboBox.set_wrap_width()`.

The `Gtk.ComboBox` widget usually restricts the user to the available choices, but it can optionally have an `Gtk.Entry`, allowing the user to enter arbitrary text if none of the available choices are suitable. To do this, use one of the static methods `Gtk.ComboBox.new_with_entry()` or `Gtk.ComboBox.new_with_model_and_entry()` to create an `Gtk.ComboBox` instance.

For a simple list of textual choices, the model-view API of `Gtk.ComboBox` can be a bit overwhelming. In this case, `Gtk.ComboBoxText` offers a simple alternative. Both `Gtk.ComboBox` and `Gtk.ComboBoxText` can contain an entry.

13.1 ComboBox objects

class `Gtk.`**ComboBox**

static **new_with_entry**()
Creates a new empty `Gtk.ComboBox` with an entry.

static **new_with_model**(*model*)
Creates a new `Gtk.ComboBox` with the model initialized to *model*.

static **new_with_model_and_entry**(*model*)
Creates a new `Gtk.ComboBox` with an entry and the model initialized to *model*.

get_active_iter()
Returns a `Gtk.TreeIter` pointing to the current active item. If no active item exists, `None` is returned.

set_model(*model*)
Sets the model used by this combo box to be *model*. Will unset a previously set model (if applicable). If model is `None`, then it will unset the model. Note that this function does not clear the cell renderers.

get_model()
Returns the `Gtk.TreeModel` which is acting as data source for this combo box.

set_entry_text_column (*text_column*)
Sets the model column which this combo box should use to get strings from to be *text_column*. The column *text_column* in the model of this combo box must be of type `str`.

This is only relevant if this combo box has been created with the "has-entry" property set to `True`.

set_wrap_width (*width*)
Sets the wrap width of this combo box to be *width*. The wrap width is basically the preferred number of columns when you want the popup to be layed out in a grid.

13.2 ComboBoxText objects

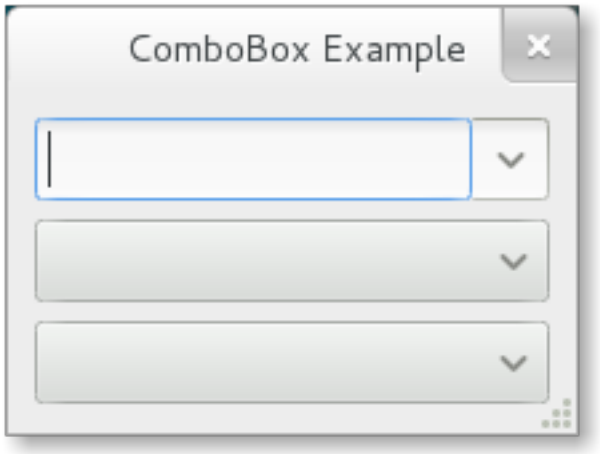
class `Gtk.`**ComboBoxText**

static new_with_entry ()
Creates a new empty `Gtk.ComboBoxText` with an entry.

append_text (*text*)
Appends *text* to the list of strings stored in this combo box.

get_active_text ()
Returns the currently active string in this combo box, or `None` if none is selected. If this combo box contains an entry, this function will return its contents (which will not necessarily be an item from the list).

13.3 Example




```
from gi.repository import Gtk

class ComboBoxWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="ComboBox Example")

        self.set_border_width(10)

        name_store = Gtk.ListStore(int, str)
        name_store.append([1, "Billy Bob"])
```

```
12          name_store.append([11, "Billy Bob Junior"])
13          name_store.append([12, "Sue Bob"])
14          name_store.append([2, "Joey Jojo"])
15          name_store.append([3, "Rob McRoberts"])
16          name_store.append([31, "Xavier McRoberts"])
17
18          vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
19
20          name_combo = Gtk.ComboBox.new_with_model_and_entry(name_store)
21          name_combo.connect("changed", self.on_name_combo_changed)
22          name_combo.set_entry_text_column(1)
23          vbox.pack_start(name_combo, False, False, 0)
24
25          country_store = Gtk.ListStore(str)
26          countries = ["Austria", "Brazil", "Belgium", "France", "Germany",
27              "Switzerland", "United Kingdom", "United States of America", "Uruguay"]
28          for country in countries:
29              country_store.append([country])
30
31          country_combo = Gtk.ComboBox.new_with_model(country_store)
32          country_combo.connect("changed", self.on_country_combo_changed)
33          renderer_text = Gtk.CellRendererText()
34          country_combo.pack_start(renderer_text, True)
35          country_combo.add_attribute(renderer_text, "text", 0)
36          vbox.pack_start(country_combo, False, False, True)
37
38          currencies = ["Euro", "US Dollars", "British Pound", "Japanese Yen",
39              "Russian Ruble", "Mexican peso", "Swiss franc"]
40          currency_combo = Gtk.ComboBoxText()
41          currency_combo.set_entry_text_column(0)
42          currency_combo.connect("changed", self.on_currency_combo_changed)
43          for currency in currencies:
44              currency_combo.append_text(currency)
45
46          vbox.pack_start(currency_combo, False, False, 0)
47
48          self.add(vbox)
49
50      def on_name_combo_changed(self, combo):
51          tree_iter = combo.get_active_iter()
52          if tree_iter != None:
53              model = combo.get_model()
54              row_id, name = model[tree_iter][:2]
55              print "Selected: ID=%d, name=%s" % (row_id, name)
56          else:
57              entry = combo.get_child()
58              print "Entered: %s" % entry.get_text()
59
60      def on_country_combo_changed(self, combo):
61          tree_iter = combo.get_active_iter()
62          if tree_iter != None:
63              model = combo.get_model()
64              country = model[tree_iter][0]
65              print "Selected: country=%s" % country
66
67      def on_currency_combo_changed(self, combo):
68          text = combo.get_active_text()
69          if text != None:
```

```
70              print "Selected: currency=%s" % text
71
72  win = ComboBoxWindow()
73  win.connect("delete-event", Gtk.main_quit)
74  win.show_all()
75  Gtk.main()
```

CHAPTER

FOURTEEN

ICONVIEW

A Gtk.IconView is a widget that displays a collection of icons in a grid view. It supports features such as drag and drop, multiple selections and item reordering.

Similarly to Gtk.TreeView, Gtk.IconView uses a Gtk.ListStore for its model. Instead of using *cell renderers*, Gtk.IconView requires that one of the columns in its Gtk.ListStore contains GdkPixbuf.Pixbuf objects.

Gtk.IconView supports numerous selection modes to allow for either selecting multiple icons at a time, restricting selections to just one item or disallowing selecting items completely. To specify a selection mode, the Gtk.IconView.set_selection_mode() method is used with one of the Gtk.SelectionMode selection modes.

14.1 IconView objects

class Gtk.**IconView**

static new_with_area(*area*)
Creates a new Gtk.IconView widget using the specified *area* to layout cells inside the icons.

static new_with_model(*model*)
Creates a new Gtk.IconView widget with the model *model*.

set_model(*model*)
Sets the model for a Gtk.IconView. If the Gtk.IconView already has a model set, it will remove it before setting the new model. If *model* is None, then it will unset the old model.

get_model()
Returns the model the Gtk.IconView is based on. Returns None if the model is unset.

set_text_column(*column*)
Sets the column with text to be *column*. The text column must be of type str.

get_text_column()
Return the column with text, or -1 if it's unset.

set_markup_column(*column*)
Sets the column with markup information for the Gtk.IconView to be *column*. The markup column must be of type str. If the markup column is set to something, it overrides the text column set by set_text_column().

get_markup_column()
Returns the column with markup text, or -1 if it's unset.

set_pixbuf_column(*column*)
Sets the column with pixbufs to be *column*. The pixbuf column must be of type `GdkPixbuf.Pixbuf`

get_pixbuf_column()
Returns the column with pixbufs, or -1 if it's unset.

get_item_at_pos(*x*, *y*)
Finds the path at the point(*x*, *y*), relative to bin_window coordinates. In contrast to `get_path_at_pos()`, this method also obtains the cell at the specified position. See `convert_widget_to_bin_window_coords()` for converting widget coordinates to bin_window coordinates.

convert_widget_to_bin_coords(*x*, *y*)
Converts widget coordinates to coordinates for the bin_window, as expected by e.g. `get_path_at_pos()`

set_cursor(*path*, *cell*, *start_editing*)
Sets the current keyboard focus to be at *path*, and selects it. This is useful when you want to focus the user's attention on a particular item. If *cell* is not `None`, then focus is given to the cell specified by it. Additionally, if *start_editing* is True, then editing should be started in the specified cell.

This function is often followed by `grab_focus()` in order to give keyboard focus to the widget. Please note that editing can only happen when the widget is realized.

get_cursor()
Returns the current cursor path and cell. If the cursor isn't currently set, then path will be `None`. If no cell currently has focus, then cell will be `None`.

selected_foreach(*func*, *data*)
Calls a function for each selected icon. Note that the model or selection cannot be modified from within this method.

set_selection_mode(*mode*)
Sets the `Gtk.SelectionMode` of the `Gtk.IconView`.

get_selection_mode()
Gets the `Gtk.SelectionMode` of the `Gtk.IconView`.

set_item_orientation(*orientation*)
Sets the "item-orientation" property which determines whether the labels are drawn beside the icons instead of below.

get_item_orientation()
Returns the `Gtk.Orientation` of the "item-orientation" property which determines whether the labels are drawn beside the icons instead of below.

set_columns(*columns*)
Sets the "columns" property which determines in how many columns the icons are arranged. If *columns* is -1, the number of columns will be chosen automatically to fill the available area.

get_columns()
Returns the value of the "columns" property.

set_item_width(*item_width*)
Sets the "item-width" property which specifies the width to use for each item. If it is set to -1, the icon view will automatically determine a suitable item size.

get_item_width()
Returns the value of the "item-width" property.

set_spacing(*spacing*)
Sets the "spacing" property which specifies the space which is inserted between the cells (i.e. the icon and the text) of an item.

set_row_spacing(*row_spacing*)
Sets the "row-spacing" property which specifies the space which is inserted between the rows of the icon view.

get_row_spacing()
Returns the value of the "row-spacing" property.

set_column_spacing(*column_spacing*)
Sets the "column-spacing" property which specifies the space which is inserted between the columns of the icon view.

get_column_spacing()
Returns the value of the "column-spacing" property.

set_margin(*margin*)
Sets the "margin" property which specifies the space which is inserted at the top, bottom, left and right of the icon view.

get_margin()
Returns the value of the "margin" property.

set_item_padding(*item_padding*)
Sets the "item-padding" property which specifies the padding around each of the icon view's items.

get_item_padding()
Returns the value of the "item-padding property.

select_path(*path*)
Selects the row at *path*.

unselect_path(*path*)
Unselects the row at *path*.

path_is_selected(*path*)
Returns `True` if the icon pointed to by *path* is currently selected. If *path* does not point to a valid location, `False` is returned.

get_selected_items()
Creates a list of paths of all selected items. Additionally, if you are planning on modifying the model after calling this function, you may want to convert the returned list into a list of `Gtk.TreeRowReference`.

select_all()
Selects all the icons. The `Gtk.IconView` must has its selection mode set to `Gtk.SelectionMode.MULTIPLE`

unselect_all()
Unselects all the icons.

scroll_to_path(*path*, *use_align*, *row_align*, *col_align*)
Moves the alignments of `Gtk.IconView` to the position specified by *path*. *row_align* determines where the row is placed, the *col_align* determines where *column* is placed. Both are expected to be between 0.0 and 1.0. 0.0 means left/top alignment, 1.0 means right/bottom alignment, 0.5 means center.

If *use_align* is `False`, the the alignment arguments are ignored, and the tree does the minimum amount of work to scroll the item onto the screen. This means that the item will be scrolled to the edge closest to its current position. If the item is currently visible on the screen, nothing is done.

This function only works if the model is set, and *path* is a valid row on the model. If the model changes before the `Gtk.IconView` is realized, the centered path will be modified to reflect this change.

get_visible_range()
Returns the first and last visible `Gtk.TreePath`. Note that there may be invisible paths in between.

set_tooltip_item(*tooltip*, *path*)
Sets the tip area of *tooltip* to be the area covered by the item at *path*. See also `set_tooltip_column()` for a simpler alternative. See also `Gtk.Tooltip.set_tip_area()`.

set_tooltip_cell(*tooltip*, *path*, *cell*)
Sets the tip area of *tooltip* to the area which *cell* occupies in the item pointed to by *path*. See also `Gtk.Tooltip.set_tip_area()`

See also `set_tooltip_column()` for a simpler alternative.

get_tooltip_context(*x*, *y*, *keyboard_tip*)
This function is supposed to be used in a "query-tooltip" signal handler for `Gtk.IconView`. The *x*, *y* and *keyboard_tip* values which are received in the signal handler, should be passed to this method without modification.

The return value indicates whether there is an icon view item at the given coordinates (`True`) or not (`False`) for mouse tooltips. For keyboard tooltips the item returned will be the cursor item. When `True`, then all of the items which have been returned will be set to point to that row and corresponding model. *x* and *y* will always be converted to be relative to the `Gtk.IconView`'s bin_window if *keyboard_tooltip* is `False`.

set_tooltip_column(*column*)
If you only plan to have simple (text-only) tooltips on full items, you can use this function to have `Gtk.IconView` handle these automatically for you. *column* should be set to the column in the `Gtk.IconView`'s model containing the tooltip texts, or -1 to disable this feature.

When enabled, "has-tooltip" will be set to `True` and `Gtk.IconView` will connect a "query-tooltip" signal handler.

Note that the signal handler sets the text with `Gtk.Tooltip.set_markup()`, so &, <, etc have to be escaped in the text.

get_tooltip_column()
Returns the column of `Gtk.IconView`'s model which is being used for displaying tooltips on `Gtk.IconView`'s rows, or -1 if this is disabled.

get_item_row(*path*)
Gets the row in which the item *path* is currently displayed. Row numbers start at 0.

get_item_column(*path*)
Gets the column in which the item *path* is currently displayed. Column numbers start at 0.

enable_model_drag_source(*start_button_mask*, *targets*, *n_targets*, *actions*)
Turns `Gtk.IconView` into a drag source for automatic DND. Calling this method sets "reorderable" to `False`.

enable_model_drag_dest(*targets*, *n_targets*, *actions*)
Turns `Gtk.IconView` into a drop destination for automatic DND. Calling this method sets "reorderable" to `False`.

unset_model_drag_source()
Undoes the effect of `enable_model_drag_source()`. Calling this method sets "reorderable" to `False`.

unset_model_drag_dest()
Undoes the effect of enable_model_drag_dest(). Calling this method sets "reorderable" to False.

set_reorderable(*reorderable*)
This method is a convenience method to allow you to reorder models that support the Gtk.TreeDragSource and the Gtk.TreeDragDest interfaces. Both Gtk.TreeStore and Gtk.ListStore support these. If *reorderable* is True, then the user can reorder the model by dragging and dropping rows. The developer can listen to these changes by connecting the model's "row_inserted" and "row_deleted" signals. The reordering is implemented by setting up the icon view as drag source and destination. Therefore, drag and drop can not be used in a reorderable view for any other purpose.

This function does not give you any degree of control over the order – any reordering is allowed. If more control is needed, you should probably handle drag and drop manually.

get_reorderable()
Retrieves whether the user can reorder the list via drag-and-drop. See set_reorderable().

set_drag_dest_item(*path*, *pos*)
Sets the item that is highlighted for feedback.

get_drag_dest_item()
Gets information about the item that is highlighted for feedback.

get_dest_item_at_pos(*drag_x*, *drag_y*)
Determines the destination item for a given position.

create_drag_icon(*path*)
Creates a Cairo.Surface representation of the item at *path*. This image is used for a drag icon.

14.2 Example

```
from gi.repository import Gtk
from gi.repository.GdkPixbuf import Pixbuf

icons = ["gtk-cut", "gtk-paste", "gtk-copy"]

class IconViewWindow(Gtk.Window):

  def __init__(self):
    Gtk.Window.__init__(self)
    self.set_default_size(200, 200)

    liststore = Gtk.ListStore(Pixbuf, str)
    iconview = Gtk.IconView.new()
    iconview.set_model(liststore)
    iconview.set_pixbuf_column(0)
    iconview.set_text_column(1)

    for icon in icons:
        pixbuf = Gtk.IconTheme.get_default().load_icon(icon, 64, 0)
        liststore.append([pixbuf, "Label"])

    self.add(iconview)

win = IconViewWindow()
win.connect("delete-event", Gtk.main_quit)
```

```
26 win.show_all()
27 Gtk.main()
```

CHAPTER

FIFTEEN

MULTILINE TEXT EDITOR

The Gtk.TextView widget can be used to display and edit large amounts of formatted text. Like the Gtk.TreeView, it has a model/view design. In this case the Gtk.TextBuffer is the model which represents the text being edited. This allows two or more Gtk.TextView widgets to share the same Gtk.TextBuffer, and allows those text buffers to be displayed slightly differently. Or you could maintain several text buffers and choose to display each one at different times in the same Gtk.TextView widget.

15.1 The View

The Gtk.TextView is the frontend with which the user can add, edit and delete textual data. They are commonly used to edit multiple lines of text. When creating a Gtk.TextView it contains its own default Gtk.TextBuffer, which you can access via the Gtk.TextView.get_buffer() method.

By default, text can be added, edited and removed from the Gtk.TextView. You can disable this by calling Gtk.TextView.set_editable(). If the text is not editable, you usually want to hide the text cursor with Gtk.TextView.set_cursor_visible() as well. In some cases it may be useful to set the justification of the text with Gtk.TextView.set_justification(). The text can be displayed at the left edge, (Gtk.Justification.LEFT), at the right edge (Gtk.Justification.RIGHT), centered (Gtk.Justification.CENTER), or distributed across the complete width (Gtk.Justification.FILL).

Another default setting of the Gtk.TextView widget is long lines of text will continue horizontally until a break is entered. To wrap the text and prevent it going off the edges of the screen call Gtk.TextView.set_wrap_mode().

15.1.1 TextView Objects

class Gtk.**TextView**

Creates a new Gtk.TextView associated with an empty default Gtk.TextBuffer.

get_buffer()

Returns the Gtk.TextBuffer being displayed by this text view.

set_editable(*editable*)

Sets the default editability of this Gtk.TextView.

set_cursor_visible(*visible*)

Toggles whether the insertion point is displayed. A buffer with no editable text probably shouldn't have a visible cursor, so you may want to turn the cursor off.

set_justification(*justification*)

Sets the default justification of text.

justification can be one of the following values:

- `Gtk.Justification.LEFT`: Text is placed at the left edge.
- `Gtk.Justification.RIGHT`: Text is placed at the right edge.
- `Gtk.Justification.CENTER`: Text is placed in the center.
- `Gtk.Justification.FILL`: Text is distributed across the complete width.

set_wrap_mode(*wrap_mode*)
Sets the line wrapping for the view.

wrap_mode can be one of the following values:

- `Gtk.WrapMode.NONE`: Do not wrap lines; just make the text area wider.
- `Gtk.WrapMode.CHAR`: Wrap text, breaking lines anywhere the cursor can appear (between characters, usually).
- `Gtk.WrapMode.WORD`: Wrap text, breaking lines in between words.
- `Gtk.WrapMode.WORD_CHAR`: Wrap text, breaking lines in between words, or if that is not enough, also between graphemes.

15.2 The Model

The `Gtk.TextBuffer` is the core of the `Gtk.TextView` widget, and is used to hold whatever text is being displayed in the `Gtk.TextView`. Setting and retrieving the contents is possible with `Gtk.TextBuffer.set_text()` and `Gtk.TextBuffer.get_text()`. However, most text manipulation is accomplished with *iterators*, represented by a `Gtk.TextIter`. An iterator represents a position between two characters in the text buffer. Iterators are not valid indefinitely; whenever the buffer is modified in a way that affects the contents of the buffer, all outstanding iterators become invalid.

Because of this, iterators can't be used to preserve positions across buffer modifications. To preserve a position, use `Gtk.TextMark`. A text buffer contains two built-in marks; an "insert" mark (which is the position of the cursor) and the "selection_bound" mark. Both of them can be retrieved using `Gtk.TextBuffer.get_insert()` and `Gtk.TextBuffer.get_selection_bound()`, respectively. By default, the location of a `Gtk.TextMark` is not shown. This can be changed by calling `Gtk.TextMark.set_visible()`.

Many methods exist to retrieve a `Gtk.TextIter`. For instance, `Gtk.TextBuffer.get_start_iter()` returns an iterator pointing to the first position in the text buffer, whereas `Gtk.TextBuffer.get_end_iter()` returns an iterator pointing past the last valid character. Retrieving the bounds of the selected text can be achieved by calling `Gtk.TextBuffer.get_selection_bounds()`.

To insert text at a specific position use `Gtk.TextBuffer.insert()`. Another useful method is `Gtk.TextBuffer.insert_at_cursor()` which inserts text wherever the cursor may be currently positioned. To remove portions of the text buffer use `Gtk.TextBuffer.delete()`.

In addition, `Gtk.TextIter` can be used to locate textual matches in the buffer using `Gtk.TextIter.forward_search()` and `Gtk.TextIter.backward_search()`. The start and end iters are used as the starting point of the search and move forwards/backwards depending on requirements.

15.2.1 TextBuffer Objects

class `Gtk.`**TextBuffer**

set_text(*text*[, *length*])
Deletes current contents of this buffer, and inserts *length* characters of *text* instead. If *length* is -1 or omitted, *text* is inserted completely.

get_text(*start_iter*, *end_iter*, *include_hidden_chars*)
Returns the text in the range *start_iter* (included) and *end_iter* (excluded). Excludes undisplayed text if *include_hidden_chars* is `False`.

get_insert()
Returns the `Gtk.TextMark` that represents the cursor (insertion point).

get_selection_bound()
Returns the `Gtk.TextMark` that represents the selection bound.

create_mark(*mark_name*, *where*[, *left_gravity*])
Creates a `Gtk.TextMark` at the position of the `Gtk.TextIter` *where*. If *mark_name* is `None`, the mark is anonymous; otherwise, the mark can be retrieved by name using `get_mark()`. If a mark has left gravity, and text is inserted at the mark's current location, the mark will be moved to the left of the newly-inserted text. If the mark has right gravity (*left_gravity* is `False`), the mark will end up on the right of newly-inserted text. The standard left-to-right cursor is a mark with right gravity (when you type, the cursor stays on the right side of the text you're typing).

If *left_gravity* is omitted, it defaults to `False`.

get_mark(*mark_name*)
Returns the `Gtk.TextMark` named name in this buffer, or `None` if no such mark exists in the buffer.

get_start_iter()
Returns a `Gtk.TextIter` pointing to first position in this buffer.

get_end_iter()
Returns a `Gtk.TextIter` pointing past the last valid character in this buffer.

get_selection_bounds()
Returns a tuple of two `Gtk.TextIter` objects pointing to the first character of the selection and to the first character after the selection, respectively. If no text is selected an empty tuple is returned.

insert(*text_iter*, *text*[, *length*])
Inserts *length* characters of *text* at position *text_iter*. If *length* is -1 or omitted, *text* is inserted completely.

insert_at_cursor(*text*[, *length*])
Simply calls `insert()`, using the current cursor position as the insertion point.

delete(*start_iter*, *end_iter*)
Deletes text between *start_iter* and *end_iter*.

create_tag(*tag_name*, ***kwargs*)
Creates a tag and adds it to the tag table of this buffer.

If *tag_name* is `None`, the tag is anonymous, otherwise a tag with the same name must not already exist in the tag table of the buffer.

kwargs is an arbitrary number of key-value pairs that represent a list properties to set on the tag, as with `tag.set_property(prop_name, value)`.

apply_tag(*tag*, *start_iter*, *end_iter*)
Applies *tag* to the given range.

remove_tag(*tag*, *start_iter*, *end_iter*)
Removes all occurrences of *tag* from the given range.

remove_all_tags(*start_iter*, *end_iter*)
Removes all tags in the given range.

class Gtk.**TextIter**

forward_search(*needle*, *flags*, *limit*)
Searches forward for *needle*. The search will not continue past the Gtk.TextIter *limit*.

flags can be set to one of the following, or any combination of it by concatenating them with the bitwise-OR operator |.

- 0: The match must be exact.
- Gtk.TextSearchFlags.VISIBLE_ONLY: The match may have invisible text interspersed in *needle*. i.e. *needle* will be a possibly-noncontiguous subsequence of the matched range.
- Gtk.TextSearchFlags.TEXT_ONLY: The match may have pixbufs or child widgets mixed inside the matched range.
- Gtk.TextSearchFlags.CASE_INSENSITIVE: The text will be matched regardless of what case it is in.

Returns a tupel containing a Gtk.TextIter pointing to the start and to the first character after the match. If no match was found, None is returned.

backward_search(*needle*, *flags*, *limit*)
Same as forward_search(), but moves backward.

class Gtk.**TextMark**

set_visible(*visible*)
Sets the visibility of this mark; the insertion point is normally visible, i.e. you can see it as a vertical bar. Also, the text widget uses a visible mark to indicate where a drop will occur when dragging-and-dropping text. Most other marks are not visible. Marks are not visible by default.

15.3 Tags

Text in a buffer can be marked with tags. A tag is an attribute that can be applied to some range of text. For example, a tag might be called "bold" and make the text inside the tag bold. However, the tag concept is more general than that; tags don't have to affect appearance. They can instead affect the behaviour of mouse and key presses, "lock" a range of text so the user can't edit it, or countless other things. A tag is represented by a Gtk.TextTag object. One Gtk.TextTag can be applied to any number of text ranges in any number of buffers.

Each tag is stored in a Gtk.TextTagTable. A tag table defines a set of tags that can be used together. Each buffer has one tag table associated with it; only tags from that tag table can be used with the buffer. A single tag table can be shared between multiple buffers, however.

To specify that some text in the buffer should have specific formatting, you must define a tag to hold that formatting information, and then apply that tag to the region of text using Gtk.TextBuffer.create_tag() and Gtk.TextBuffer.apply_tag():

```
tag = textbuffer.create_tag("orange_bg", background="orange")
textbuffer.apply_tag(tag, start_iter, end_iter)
```

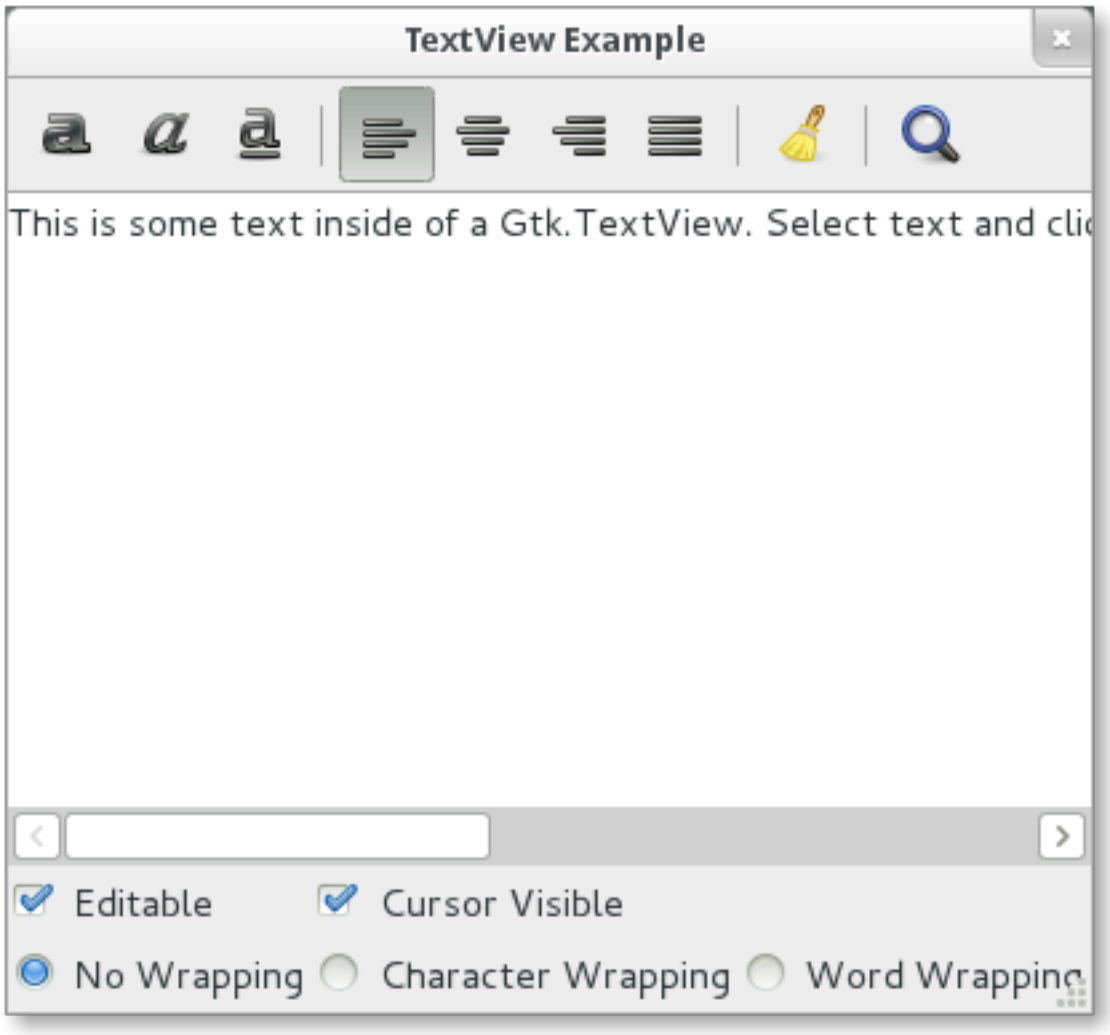
The following are some of the common styles applied to text:

- Background colour ("foreground" property)
- Foreground colour ("background" property)
- Underline ("underline" property)

- Bold (“weight” property)
- Italics (“style” property)
- Strikethrough (“strikethrough” property)
- Justification (“justification” property)
- Size (“size” and “size-points” properties)
- Text wrapping (“wrap-mode” property)

You can also delete particular tags later using `Gtk.TextBuffer.remove_tag()` or delete all tags in a given region by calling `Gtk.TextBuffer.remove_all_tags()`.

15.4 Example




```
from gi.repository import Gtk, Pango

class SearchDialog(Gtk.Dialog):

    def __init__(self, parent):
```

```
        Gtk.Dialog.__init__(self, "Search", parent,
            Gtk.DialogFlags.MODAL, buttons=(
            Gtk.STOCK_FIND, Gtk.ResponseType.OK,
            Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL))

        box = self.get_content_area()

        label = Gtk.Label("Insert text you want to search for:")
        box.add(label)

        self.entry = Gtk.Entry()
        box.add(self.entry)

        self.show_all()

class TextViewWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="TextView Example")

        self.set_default_size(-1, 350)

        self.grid = Gtk.Grid()
        self.add(self.grid)

        self.create_textview()
        self.create_toolbar()
        self.create_buttons()

    def create_toolbar(self):
        toolbar = Gtk.Toolbar()
        self.grid.attach(toolbar, 0, 0, 3, 1)

        button_bold = Gtk.ToolButton.new_from_stock(Gtk.STOCK_BOLD)
        toolbar.insert(button_bold, 0)

        button_italic = Gtk.ToolButton.new_from_stock(Gtk.STOCK_ITALIC)
        toolbar.insert(button_italic, 1)

        button_underline = Gtk.ToolButton.new_from_stock(Gtk.STOCK_UNDERLINE)
        toolbar.insert(button_underline, 2)

        button_bold.connect("clicked", self.on_button_clicked, self.tag_bold)
        button_italic.connect("clicked", self.on_button_clicked, self.tag_italic)
        button_underline.connect("clicked", self.on_button_clicked, self.tag_underline)

        toolbar.insert(Gtk.SeparatorToolItem(), 3)

        radio_justifyleft = Gtk.RadioToolButton()
        radio_justifyleft.set_stock_id(Gtk.STOCK_JUSTIFY_LEFT)
        toolbar.insert(radio_justifyleft, 4)

        radio_justifycenter = Gtk.RadioToolButton.new_with_stock_from_widget(
            radio_justifyleft, Gtk.STOCK_JUSTIFY_CENTER)
        toolbar.insert(radio_justifycenter, 5)

        radio_justifyright = Gtk.RadioToolButton.new_with_stock_from_widget(
            radio_justifyleft, Gtk.STOCK_JUSTIFY_RIGHT)
```

```
64          toolbar.insert(radio_justifyright, 6)
65
66          radio_justifyfill = Gtk.RadioToolButton.new_with_stock_from_widget(
67              radio_justifyleft, Gtk.STOCK_JUSTIFY_FILL)
68          toolbar.insert(radio_justifyfill, 7)
69
70          radio_justifyleft.connect("toggled", self.on_justify_toggled,
71              Gtk.Justification.LEFT)
72          radio_justifycenter.connect("toggled", self.on_justify_toggled,
73              Gtk.Justification.CENTER)
74          radio_justifyright.connect("toggled", self.on_justify_toggled,
75              Gtk.Justification.RIGHT)
76          radio_justifyfill.connect("toggled", self.on_justify_toggled,
77              Gtk.Justification.FILL)
78
79          toolbar.insert(Gtk.SeparatorToolItem(), 8)
80
81          button_clear = Gtk.ToolButton.new_from_stock(Gtk.STOCK_CLEAR)
82          button_clear.connect("clicked", self.on_clear_clicked)
83          toolbar.insert(button_clear, 9)
84
85          toolbar.insert(Gtk.SeparatorToolItem(), 10)
86
87          button_search = Gtk.ToolButton.new_from_stock(Gtk.STOCK_FIND)
88          button_search.connect("clicked", self.on_search_clicked)
89          toolbar.insert(button_search, 11)
90
91      def create_textview(self):
92          scrolledwindow = Gtk.ScrolledWindow()
93          scrolledwindow.set_hexpand(True)
94          scrolledwindow.set_vexpand(True)
95          self.grid.attach(scrolledwindow, 0, 1, 3, 1)
96
97          self.textview = Gtk.TextView()
98          self.textbuffer = self.textview.get_buffer()
99          self.textbuffer.set_text("This is some text inside of a Gtk.TextView. "
100             + "Select text and click one of the buttons 'bold', 'italic', "
101             + "or 'underline' to modify the text accordingly.")
102         scrolledwindow.add(self.textview)
103
104         self.tag_bold = self.textbuffer.create_tag("bold",
105             weight=Pango.Weight.BOLD)
106         self.tag_italic = self.textbuffer.create_tag("italic",
107             style=Pango.Style.ITALIC)
108         self.tag_underline = self.textbuffer.create_tag("underline",
109             underline=Pango.Underline.SINGLE)
110         self.tag_found = self.textbuffer.create_tag("found",
111             background="yellow")
112
113     def create_buttons(self):
114         check_editable = Gtk.CheckButton("Editable")
115         check_editable.set_active(True)
116         check_editable.connect("toggled", self.on_editable_toggled)
117         self.grid.attach(check_editable, 0, 2, 1, 1)
118
119         check_cursor = Gtk.CheckButton("Cursor Visible")
120         check_cursor.set_active(True)
121         check_editable.connect("toggled", self.on_cursor_toggled)
```

```
122         self.grid.attach_next_to(check_cursor, check_editable,
123             Gtk.PositionType.RIGHT, 1, 1)
124
125         radio_wrapnone = Gtk.RadioButton.new_with_label_from_widget(None,
126             "No Wrapping")
127         self.grid.attach(radio_wrapnone, 0, 3, 1, 1)
128
129         radio_wrapchar = Gtk.RadioButton.new_with_label_from_widget(
130             radio_wrapnone, "Character Wrapping")
131         self.grid.attach_next_to(radio_wrapchar, radio_wrapnone,
132             Gtk.PositionType.RIGHT, 1, 1)
133
134         radio_wrapword = Gtk.RadioButton.new_with_label_from_widget(
135             radio_wrapnone, "Word Wrapping")
136         self.grid.attach_next_to(radio_wrapword, radio_wrapchar,
137             Gtk.PositionType.RIGHT, 1, 1)
138
139         radio_wrapnone.connect("toggled", self.on_wrap_toggled, Gtk.WrapMode.NONE)
140         radio_wrapchar.connect("toggled", self.on_wrap_toggled, Gtk.WrapMode.CHAR)
141         radio_wrapword.connect("toggled", self.on_wrap_toggled, Gtk.WrapMode.WORD)
142
143     def on_button_clicked(self, widget, tag):
144         bounds = self.textbuffer.get_selection_bounds()
145         if len(bounds) != 0:
146             start, end = bounds
147             self.textbuffer.apply_tag(tag, start, end)
148
149     def on_clear_clicked(self, widget):
150         start = self.textbuffer.get_start_iter()
151         end = self.textbuffer.get_end_iter()
152         self.textbuffer.remove_all_tags(start, end)
153
154     def on_editable_toggled(self, widget):
155         self.textview.set_editable(widget.get_active())
156
157     def on_cursor_toggled(self, widget):
158         self.textview.set_cursor_visible(widget.get_active())
159
160     def on_wrap_toggled(self, widget, mode):
161         self.textview.set_wrap_mode(mode)
162
163     def on_justify_toggled(self, widget, justification):
164         self.textview.set_justification(justification)
165
166     def on_search_clicked(self, widget):
167         dialog = SearchDialog(self)
168         response = dialog.run()
169         if response == Gtk.ResponseType.OK:
170             cursor_mark = self.textbuffer.get_insert()
171             start = self.textbuffer.get_iter_at_mark(cursor_mark)
172             if start.get_offset() == self.textbuffer.get_char_count():
173                 start = self.textbuffer.get_start_iter()
174
175             self.search_and_mark(dialog.entry.get_text(), start)
176
177         dialog.destroy()
178
179     def search_and_mark(self, text, start):
```

```
180         end = self.textbuffer.get_end_iter()
181         match = start.forward_search(text, 0, end)
182 
183         if match != None:
184             match_start, match_end = match
185             self.textbuffer.apply_tag(self.tag_found, match_start, match_end)
186             self.search_and_mark(text, match_end)
187 
188 win = TextViewWindow()
189 win.connect("delete-event", Gtk.main_quit)
190 win.show_all()
191 Gtk.main()
```

CHAPTER

SIXTEEN

MENUS

GTK+ comes with two different types of menus, `Gtk.MenuBar` and `Gtk.Toolbar`. `Gtk.MenuBar` is a standard menu bar which contains one or more `Gtk.MenuItem` instances or one of its subclasses. `Gtk.Toolbar` widgets are used for quick accessibility to commonly used functions of an application. Examples include creating a new document, printing a page or undoing an operation. It contains one or more instances of `Gtk.ToolItem` or one of its subclasses.

16.1 Actions

Although, there are specific APIs to create menus and toolbars, you should use `Gtk.UIManager` and create `Gtk.Action` instances. Actions are organised into groups. A `Gtk.ActionGroup` is essentially a map from names to `Gtk.Action` objects. All actions that would make sense to use in a particular context should be in a single group. Multiple action groups may be used for a particular user interface. In fact, it is expected that most non-trivial applications will make use of multiple groups. For example, in an application that can edit multiple documents, one group holding global actions (e.g. quit, about, new), and one group per document holding actions that act on that document (eg. save, cut/copy/paste, etc). Each window's menus would be constructed from a combination of two action groups.

Different classes representing different types of actions exist:

- `Gtk.Action`: An action which can be triggered by a menu or toolbar item
- `Gtk.ToggleAction`: An action which can be toggled between two states
- `Gtk.RadioAction`: An action of which only one in a group can be active
- `Gtk.RecentAction`: An action of which represents a list of recently used files

Actions represent operations that the user can be perform, along with some information how it should be presented in the interface, including its name (not for display), its label (for display), an accelerator, whether a label indicates a *stock item*, a tooltip, as well as the callback that is called when the action gets activated.

You can create actions by either calling one of the constructors directly and adding them to a `Gtk.ActionGroup` by calling `Gtk.ActionGroup.add_action()` or `Gtk.ActionGroup.add_action_with_accel()`, or by calling one of the convenience functions:

- `Gtk.ActionGroup.add_actions()`,
- `Gtk.ActionGroup.add_toggle_actions()`
- `Gtk.ActionGroup.add_radio_actions()`.

Note that you must specify actions for sub menus as well as menu items.

16.1.1 Action Objects

class `Gtk.`**`Action`**(*name*, *label*, *tooltip*, *stock_id*)
name must be a unique name of the action.

If *label* is not `None`, it is displayed in menu items and on buttons.

If *tooltip* is not `None`, it is used as tooltip for the action.

If *stock_id* is not `None`, it is used to lookup the *stock item* to display in widgets representing the action.

class `Gtk.`**`ToggleAction`**(*name*, *label*, *tooltip*, *stock_id*)
The arguments are the same as for the `Gtk.Action` constructor.

class `Gtk.`**`RadioAction`**(*name*, *label*, *tooltip*, *stock_id*, *value*)
The first four arguments are the same as for the `Gtk.Action` constructor.

value indicates the value which `get_current_value()` should return if this action is selected.

`get_current_value`()
Obtains the "value" property of the currently active member of the group to which this action belongs.

`join_group`(*group_source*)
Joins this radio action object to the group of the *group_source* radio action object.

group_source must be a radio action object whose group we are joining, or `None` to remove the radio action from its group.

class `Gtk.`**`ActionGroup`**(*name*)
Creates a new `Gtk.ActionGroup` instance. The name of the action group is used when associating keybindings with the actions.

`add_action`(*action*)
Adds an `Gtk.Action` object to the action group.

Note that this method does not set up the accelerator path of the action, use `add_action_with_accel()` instead.

`add_action_with_accel`(*action*, *accelerator*)
Adds an `Gtk.Action` object to the action group and sets up the accelerator.

accelerator must be in the format understood by `Gtk.accelerator_parse()`, or `""` for no accelerator, or `None` to use the stock accelerator.

`add_actions`(*entries*[, *user_data*])
This is a convenience function to create a number of `Gtk.Action` objects and add them to this action group.

entries is a list of tuples which can vary in size from one to six items with the following information:

- The name of the action (mandatory)
- The *stock item* of the action (default: `None`)
- The label for the action (default: `None`)
- The accelerator for the action, in the format understood by the `Gtk.accelerator_parse()` function (default: `None`)
- The tooltip of the action (default: `None`)
- The callback function invoked when the action is activated (default: `None`)

The "activate" signals of the actions are connected to the callbacks.

If *user_data* is not `None`, it is passed to the callback function (if specified).

add_toggle_actions(*entries*[, *user_data*])
This is a convenience function to create a number of Gtk.ToggleAction objects and add them to this action group.

entries is a list of tuples which can vary in size from one to seven items with the following information:

- The name of the action (mandatory)
- The *stock item* of the action (default: None)
- The label for the action (default: None)
- The accelerator for the action, in the format understood by the Gtk.accelerator_parse() function (default: None)
- The tooltip of the action (default: None)
- The callback function invoked when the action is activated (default: None)
- A Boolean indicating whether the toggle action is active (default: False)

The "activate" signals of the actions are connected to the callbacks.

If *user_data* is not None, it is passed to the callback function (if specified).

add_radio_actions(*entries*[, *value*[, *on_change*[, *user_data*]]])
This is a convenience routine to create a group of Gtk.RadioAction objects and add them to this action group.

entries is a list of tuples which can vary in size from one to six items with the following information:

- The name of the action (mandatory)
- The *stock item* of the action (default: None)
- The label for the action (default: None)
- The accelerator for the action, in the format understood by the Gtk.accelerator_parse() function (default: None)
- The tooltip of the action (default: None)
- The value to set on the radio action (default: 0)

value specifies the radio action that should be set active.

The "changed" signal of the first radio action is connected to the *on_change* callback (if specified).

If *user_data* is not None, it is passed to the callback function (if specified).

Gtk.**accelerator_parse**(*accelerator*)
Parses a string representing an accelerator. The format looks like "<Control>a" or "<Shift><Alt>F1" or "<Release>z" (the last one is for key release). The parser is fairly liberal and allows lower or upper case, and also abbreviations such as "<Ctl>" and "<Ctrl>". For character keys the name is not the symbol, but the lowercase name, e.g. one would use "<Ctrl>minus" instead of "<Ctrl>-".

Returns a tuple (accelerator_key, accelerator_mods), where the latter represents the accelerator modifier mask and the first the accelerator keyval. Both values will be set to 0 (zero) if parsing failed.

16.2 UI Manager

Gtk.UIManager provides an easy way of creating menus and toolbars using an XML-like description.

First of all, you should add the `Gtk.ActionGroup` to the UI Manager with `Gtk.UIManager.insert_action_group()`. At this point is also a good idea to tell the parent window to respond to the specified keyboard shortcuts, by using `Gtk.UIManager.get_accel_group()` and `Gtk.Window.add_accel_group()`.

Then, you can define the actual visible layout of the menus and toolbars, and add the UI layout. This "ui string" uses an XML format, in which you should mention the names of the actions that you have already created. Remember that these names are just the identifiers that we used when creating the actions. They are not the text that the user will see in the menus and toolbars. We provided those human-readable names when we created the actions.

Finally, you retrieve the root widget with `Gtk.UIManager.get_widget()` and add the widget to a container such as `Gtk.Box`.

16.2.1 UIManager Objects

class `Gtk.`**`UIManager`**

`insert_action_group`(*action_group*[, *pos*])
Inserts an action group into the list of action groups associated with this manager. Actions in earlier groups hide actions with the same name in later groups.

pos is the position at which the group will be inserted. If omitted, it will be appended.

`get_accel_group`()
Returns the group of global keyboard accelerators associated with this manager.

`get_widget`(*path*)
Looks up a widget by following a path. The path consists of the names specified in the XML description of the UI. separated by '/'. Elements which don't have a name or action attribute in the XML (e.g. <popup>) can be addressed by their XML element name (e.g. "popup"). The root element ("/ui") can be omitted in the path.

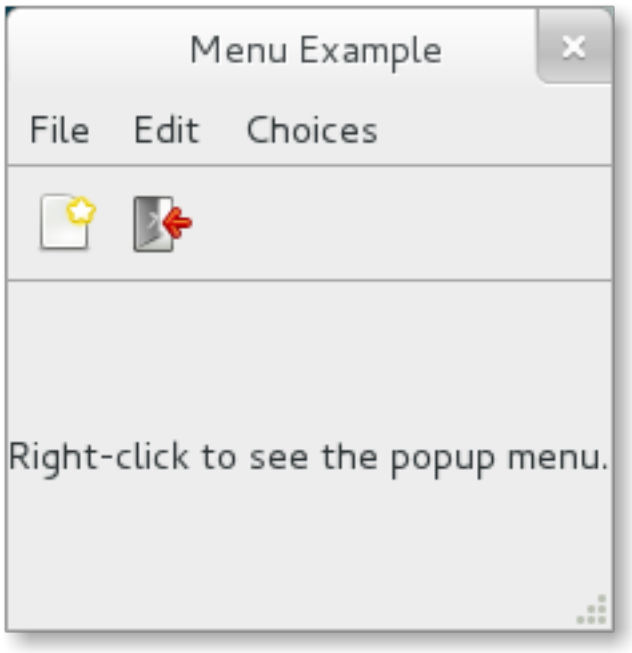
Returns the widget found by following the *path*, or `None` if no widget was found.

`add_ui_from_string`(*text*)
Parses *text* containing a UI definition and merges it with the current contents of manager. An enclosing <ui> element is added if it is missing.

Returns the merge id for the merged UI.

Throws an exception if an error occurred.

16.3 Example




```
from gi.repository import Gtk, Gdk

UI_INFO = """
<ui>
  <menubar name='MenuBar'>
    <menu action='FileMenu'>
      <menu action='FileNew'>
        <menuitem action='FileNewStandard' />
        <menuitem action='FileNewFoo' />
        <menuitem action='FileNewGoo' />
      </menu>
      <separator />
      <menuitem action='FileQuit' />
    </menu>
    <menu action='EditMenu'>
      <menuitem action='EditCopy' />
      <menuitem action='EditPaste' />
      <menuitem action='EditSomething' />
    </menu>
    <menu action='ChoicesMenu'>
      <menuitem action='ChoiceOne'/>
      <menuitem action='ChoiceTwo'/>
      <separator />
      <menuitem action='ChoiceThree'/>
    </menu>
  </menubar>
  <toolbar name='ToolBar'>
    <toolitem action='FileNewStandard' />
    <toolitem action='FileQuit' />
  </toolbar>
  <popup name='PopupMenu'>
    <menuitem action='EditCopy' />
```

```
    <menuitem action='EditPaste' />
    <menuitem action='EditSomething' />
  </popup>
</ui>
"""

class MenuExampleWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Menu Example")

        self.set_default_size(200, 200)

        action_group = Gtk.ActionGroup("my_actions")

        self.add_file_menu_actions(action_group)
        self.add_edit_menu_actions(action_group)
        self.add_choices_menu_actions(action_group)

        uimanager = self.create_ui_manager()
        uimanager.insert_action_group(action_group)

        menubar = uimanager.get_widget("/MenuBar")

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        box.pack_start(menubar, False, False, 0)

        toolbar = uimanager.get_widget("/ToolBar")
        box.pack_start(toolbar, False, False, 0)

        eventbox = Gtk.EventBox()
        eventbox.connect("button-press-event", self.on_button_press_event)
        box.pack_start(eventbox, True, True, 0)

        label = Gtk.Label("Right-click to see the popup menu.")
        eventbox.add(label)

        self.popup = uimanager.get_widget("/PopupMenu")

        self.add(box)

    def add_file_menu_actions(self, action_group):
        action_filemenu = Gtk.Action("FileMenu", "File", None, None)
        action_group.add_action(action_filemenu)

        action_filenewmenu = Gtk.Action("FileNew", None, None, Gtk.STOCK_NEW)
        action_group.add_action(action_filenewmenu)

        action_new = Gtk.Action("FileNewStandard", "_New",
            "Create a new file", Gtk.STOCK_NEW)
        action_new.connect("activate", self.on_menu_file_new_generic)
        action_group.add_action_with_accel(action_new, None)

        action_group.add_actions([
            ("FileNewFoo", None, "New Foo", None, "Create new foo",
             self.on_menu_file_new_generic),
            ("FileNewGoo", None, "_New Goo", None, "Create new goo",
             self.on_menu_file_new_generic),
```

```
91          ])
92
93          action_filequit = Gtk.Action("FileQuit", None, None, Gtk.STOCK_QUIT)
94          action_filequit.connect("activate", self.on_menu_file_quit)
95          action_group.add_action(action_filequit)
96
97      def add_edit_menu_actions(self, action_group):
98          action_group.add_actions([
99              ("EditMenu", None, "Edit"),
100             ("EditCopy", Gtk.STOCK_COPY, None, None, None,
101              self.on_menu_others),
102             ("EditPaste", Gtk.STOCK_PASTE, None, None, None,
103              self.on_menu_others),
104             ("EditSomething", None, "Something", "<control><alt>S", None,
105              self.on_menu_others)
106         ])
107
108     def add_choices_menu_actions(self, action_group):
109         action_group.add_action(Gtk.Action("ChoicesMenu", "Choices", None, None))
110
111         action_group.add_radio_actions([
112             ("ChoiceOne", None, "One", None, None, 1),
113             ("ChoiceTwo", None, "Two", None, None, 2)
114         ], 1, self.on_menu_choices_changed)
115
116         three = Gtk.ToggleAction("ChoiceThree", "Three", None, None)
117         three.connect("toggled", self.on_menu_choices_toggled)
118         action_group.add_action(three)
119
120     def create_ui_manager(self):
121         uimanager = Gtk.UIManager()
122
123         # Throws exception if something went wrong
124         uimanager.add_ui_from_string(UI_INFO)
125
126         # Add the accelerator group to the toplevel window
127         accelgroup = uimanager.get_accel_group()
128         self.add_accel_group(accelgroup)
129         return uimanager
130
131     def on_menu_file_new_generic(self, widget):
132         print "A File|New menu item was selected."
133
134     def on_menu_file_quit(self, widget):
135         Gtk.main_quit()
136
137     def on_menu_others(self, widget):
138         print "Menu item " + widget.get_name() + " was selected"
139
140     def on_menu_choices_changed(self, widget, current):
141         print current.get_name() + " was selected."
142
143     def on_menu_choices_toggled(self, widget):
144         if widget.get_active():
145             print widget.get_name() + " activated"
146         else:
147             print widget.get_name() + " deactivated"
148
```

```
149     def on_button_press_event(self, widget, event):
150         # Check if right mouse button was preseed
151         if event.type == Gdk.EventType.BUTTON_PRESS and event.button == 3:
152             self.popup.popup(None, None, None, None, event.button, event.time)
153             return True # event has been handled
154
155 window = MenuExampleWindow()
156 window.connect("delete-event", Gtk.main_quit)
157 window.show_all()
158 Gtk.main()
```

CHAPTER

SEVENTEEN

DIALOGS

Dialog windows are very similar to standard windows, and are used to provide or retrieve information from the user. They are often used to provide a preferences window, for example. The major difference a dialog has is some prepacked widgets which layout the dialog automatically. From there, we can simply add labels, buttons, check buttons, etc. Another big difference is the handling of responses to control how the application should behave after the dialog has been interacted with.

There are several derived Dialog classes which you might find useful. `Gtk.MessageDialog` is used for most simple notifications. But at other times you might need to derive your own dialog class to provide more complex functionality.

17.1 Custom Dialogs

To pack widgets into a custom dialog, you should pack them into the `Gtk.Box`, available via `Gtk.Dialog.get_content_area()`. To just add a `Gtk.Button` to the bottom of the dialog, you could use the `Gtk.Dialog.add_button()` method.

A 'modal' dialog (that is, one which freezes the rest of the application from user input), can be created by calling `Gtk.Dialog.set_modal` on the dialog or set the `flags` argument of the `Gtk.Dialog` constructor to include the `Gtk.DialogFlags.MODAL` flag.

Clicking a button will emit a signal called "response". If you want to block waiting for a dialog to return before returning control flow to your code, you can call `Gtk.Dialog.run()`. This method returns an int which may be a value from the `Gtk.ResponseType` or it could be the custom response value that you specified in the `Gtk.Dialog` constructor or `Gtk.Dialog.add_button()`.

Finally, there are two ways to remove a dialog. The `Gtk.Widget.hide()` method removes the dialog from view, however keeps it stored in memory. This is useful to prevent having to construct the dialog again if it needs to be accessed at a later time. Alternatively, the `Gtk.Widget.destroy()` method can be used to delete the dialog from memory once it is no longer needed. It should be noted that if the dialog needs to be accessed after it has been destroyed, it will need to be constructed again otherwise the dialog window will be empty.

17.1.1 Dialog Objects

class `Gtk.`**`Dialog`**(*[title[, parent[, flags[, buttons]]]]*)

Creates a new `Gtk.Dialog` with title *title* and transient parent *parent*. The *flags* argument can be used to make the dialog model (`Gtk.DialogFlags.MODAL`) and/or to have it destroyed along with its transient parent (`Gtk.DialogFlags.DESTROY_WITH_PARENT`).

buttons is a tuple of buttons which can be set to provide a range of different buttons and responses. See the `add_button()` method for details.

All arguments are optional and can be referred to as key-word arguments as well.

get_content_area()
Return the content area of of this dialog.

add_button(*button_text*, *response_id*)
Adds a button with the given text (or a stock button, if *button_text* is a *stock item*) and sets things up so that clicking the button will emit the "response" signal with the given *response_id*. The button is appended to the end of the dialog's action area.

response_id can be any positive integer or one of the predefined `Gtk.ResponseType` values:

- `Gtk.ResponseType.NONE`
- `Gtk.ResponseType.REJECT`
- `Gtk.ResponseType.ACCEPT`
- `Gtk.ResponseType.DELETE_EVENT`
- `Gtk.ResponseType.OK`
- `Gtk.ResponseType.CANCEL`
- `Gtk.ResponseType.CLOSE`
- `Gtk.ResponseType.YES`
- `Gtk.ResponseType.NO`
- `Gtk.ResponseType.APPLY`
- `Gtk.ResponseType.HELP`

The button widget is returned, but usually you don't need it.

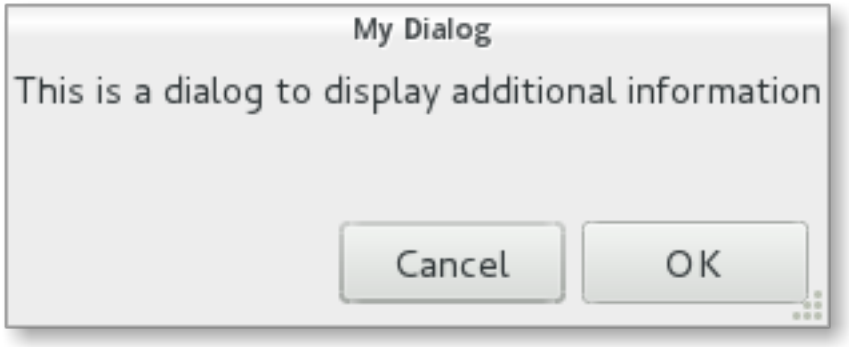
add_buttons(*button_text*, *response_id*[, ...])
Adds several buttons to this dialog using the button data passed as arguments to the method. This method is the same as calling `add_button()` repeatedly. The button data pairs - button text (or *stock item*) and a response ID integer are passed individually. For example:

```
dialog.add_buttons(Gtk.STOCK_OPEN, 42, "Close", Gtk.ResponseType.CLOSE)
```

set_modal(*is_modal*)
Sets a dialog modal or non-modal. Modal dialogs prevent interaction with other windows in the same application.

run()
Blocks in a recursive main loop until the dialog either emits the "response" signal, or is destroyed. If the dialog is destroyed during the call to `run()`, `run()` returns `Gtk.ResponseType.NONE`. Otherwise, it returns the response ID from the "response" signal emission.

17.1.2 Example




```
from gi.repository import Gtk

class DialogExample(Gtk.Dialog):

    def __init__(self, parent):
        Gtk.Dialog.__init__(self, "My Dialog", parent, 0,
            (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
             Gtk.STOCK_OK, Gtk.ResponseType.OK))

        self.set_default_size(150, 100)

        label = Gtk.Label("This is a dialog to display additional information")

        box = self.get_content_area()
        box.add(label)
        self.show_all()

class DialogWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Dialog Example")

        self.set_border_width(6)

        button = Gtk.Button("Open dialog")
        button.connect("clicked", self.on_button_clicked)

        self.add(button)

    def on_button_clicked(self, widget):
        dialog = DialogExample(self)
        response = dialog.run()

        if response == Gtk.ResponseType.OK:
            print "The OK button was clicked"
        elif response == Gtk.ResponseType.CANCEL:
            print "The Cancel button was clicked"

        dialog.destroy()

win = DialogWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
```

```
44 Gtk.main()
```

17.2 MessageDialog

Gtk.MessageDialog is a convenience class, used to create simple, standard message dialogs, with a message, an icon, and buttons for user response You can specify the type of message and the text in the Gtk.MessageDialog constructor, as well as specifying standard buttons.

In some dialogs which require some further explanation of what has happened, a secondary text can be added. In this case, the primary message entered when creating the message dialog is made bigger and set to bold text. The secondary message can be set by calling Gtk.MessageDialog.format_secondary_text().

17.2.1 MessageDialog Objects

class Gtk.**MessageDialog**(*[parent[, flags[, message_type[, buttons, [message_format]]]]]*)

Creates a new Gtk.MessageDialog with transient parent *parent*. The *flags* argument can be used to make the dialog model (Gtk.DialogFlags.MODAL) and/or to have it destroyed along with its transient parent (Gtk.DialogFlags.DESTROY_WITH_PARENT).

message_type can be set to one of the following values:

- Gtk.MessageType.INFO: Informational message
- Gtk.MessageType.WARNING: Non-fatal warning message
- Gtk.MessageType.QUESTION: Question requiring a choice
- Gtk.MessageType.ERROR: Fatal error message
- Gtk.MessageType.OTHER: None of the above, doesn't get an icon

It is also possible to set a variety of buttons on the message dialog, to retrieve different responses from the user. One of the following values can be used:

- Gtk.ButtonsType.NONE: no buttons at all
- Gtk.ButtonsType.OK: an OK button
- Gtk.ButtonsType.CLOSE: a Close button
- Gtk.ButtonsType.CANCEL: a Cancel button
- Gtk.ButtonsType.YES_NO: Yes and No buttons
- Gtk.ButtonsType.OK_CANCEL: OK and Cancel buttons

Finally, *message_format* is some text that the user may want to see.

All arguments are optional and can be referred to as key-word arguments as well.

format_secondary_text(*message_format*)

Sets the secondary text of the message dialog to be *message_format*.

Note that setting a secondary text makes the primary text (*message_format* argument of Gtk.MessageDialog constructor) become bold, unless you have provided explicit markup.

17.2.2 Example


```
from gi.repository import Gtk

class MessageDialogWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="MessageDialog Example")

        box = Gtk.Box(spacing=6)
        self.add(box)

        button1 = Gtk.Button("Information")
        button1.connect("clicked", self.on_info_clicked)
        box.add(button1)

        button2 = Gtk.Button("Error")
        button2.connect("clicked", self.on_error_clicked)
        box.add(button2)

        button3 = Gtk.Button("Warning")
        button3.connect("clicked", self.on_warn_clicked)
        box.add(button3)

        button4 = Gtk.Button("Question")
        button4.connect("clicked", self.on_question_clicked)
        box.add(button4)

    def on_info_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.INFO,
            Gtk.ButtonsType.OK, "This is an INFO MessageDialog")
        dialog.format_secondary_text(
            "And this is the secondary text that explains things.")
        dialog.run()
        print "INFO dialog closed"

        dialog.destroy()

    def on_error_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.ERROR,
            Gtk.ButtonsType.CANCEL, "This is an ERROR MessageDialog")
        dialog.format_secondary_text(
```

```
            "And this is the secondary text that explains things.")
        dialog.run()
        print "ERROR dialog closed"

        dialog.destroy()

    def on_warn_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.WARNING,
            Gtk.ButtonsType.OK_CANCEL, "This is an WARNING MessageDialog")
        dialog.format_secondary_text(
            "And this is the secondary text that explains things.")
        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            print "WARN dialog closed by clicking OK button"
        elif response == Gtk.ResponseType.CANCEL:
            print "WARN dialog closed by clicking CANCEL button"

        dialog.destroy()

    def on_question_clicked(self, widget):
        dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.QUESTION,
            Gtk.ButtonsType.YES_NO, "This is an QUESTION MessageDialog")
        dialog.format_secondary_text(
            "And this is the secondary text that explains things.")
        response = dialog.run()
        if response == Gtk.ResponseType.YES:
            print "QUESTION dialog closed by clicking YES button"
        elif response == Gtk.ResponseType.NO:
            print "QUESTION dialog closed by clicking NO button"

        dialog.destroy()

win = MessageDialogWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

17.3 FileChooserDialog

The `Gtk.FileChooserDialog` is suitable for use with "File/Open" or "File/Save" menu items. You can use all of the `Gtk.FileChooser` methods on the file chooser dialog as well as those for `Gtk.Dialog`.

When creating a `Gtk.FileChooserDialog` you have to define the dialog's purpose:

- To select a file for opening, as for a File/Open command, use `Gtk.FileChooserAction.OPEN`
- To save a file for the first time, as for a File/Save command, use `Gtk.FileChooserAction.SAVE`, and suggest a name such as "Untitled" with `Gtk.FileChooser.set_current_name()`.
- To save a file under a different name, as for a File/Save As command, use `Gtk.FileChooserAction.SAVE`, and set the existing filename with `Gtk.FileChooser.set_filename()`.
- To choose a folder instead of a file, use `Gtk.FileChooserAction.SELECT_FOLDER`.

`Gtk.FileChooserDialog` inherits from `Gtk.Dialog`, so buttons have response IDs such as `Gtk.ResponseType.ACCEPT` and `Gtk.ResponseType.CANCEL` which can be specified in the

Gtk.FileChooserDialog constructor. In contrast to Gtk.Dialog, you can not use custom response codes with Gtk.FileChooserDialog. It expects that at least one button will have of the following response IDs:

- Gtk.ResponseType.ACCEPT
- Gtk.ResponseType.OK
- Gtk.ResponseType.YES
- Gtk.ResponseType.APPLY

When the user is finished selecting files, your program can get the selected names either as filenames (Gtk.FileChooser.get_filename()) or as URIs (Gtk.FileChooser.get_uri()).

By default, Gtk.FileChooser only allows a single file to be selected at a time. To enable multiple files to be selected, use Gtk.FileChooser.set_select_multiple(). Retrieving a list of selected files is possible with either Gtk.FileChooser.get_filenames() or Gtk.FileChooser.get_uris().

Gtk.FileChooser also supports a variety of options which make the files and folders more configurable and accessible.

- Gtk.FileChooser.set_local_only(): Only local files can be selected.
- Gtk.FileChooser.show_hidden(): Hidden files and folders are displayed.
- Gtk.FileChooser.set_do_overwrite_confirmation(): If the file chooser was configured in Gtk.FileChooserAction.SAVE mode, it will present a confirmation dialog if the user types a file name that already exists.

Furthermore, you can specify which kind of files are displayed by creating Gtk.FileFilter objects and calling Gtk.FileChooser.add_filter(). The user can then select one of the added filters from a combo box at the bottom of the file chooser.

17.3.1 FileChooser Objects

class Gtk.**FileChooserDialog**(*[title[, parent[, action[, buttons]]]]*)

Creates a new Gtk.FileChooserDialog with title *title and transient parent *parent*.

action can be one of the following:

- Gtk.FileChooserAction.OPEN: The file chooser will only let the user pick an existing file.
- Gtk.FileChooserAction.SAVE: The file chooser will let the user pick an existing file, or type in a new filename.
- Gtk.FileChooserAction.SELECT_FOLDER: The file chooser will let the user pick an existing folder.
- Gtk.FileChooserAction.CREATE_FOLDER: The file chooser will let the user name an existing or new folder.

The *buttons* argument has the same format as for the Gtk.Dialog constructor.

class Gtk.**FileChooser**

set_current_name(*name*)

Sets the current name in the file selector, as if entered by the user.

set_filename(*filename*)

Sets *filename* as the current filename for the file chooser, by changing to the file's parent folder and actually selecting the file in list; all other files will be unselected. If the chooser is in

`Gtk.FileChooserAction.SAVE` mode, the file's base name will also appear in the dialog's file name entry.

Note that the file must exist, or nothing will be done except for the directory change.

set_select_multiple(*select_multiple*)
Sets whether multiple files can be selected. This is only relevant if the mode is `Gtk.FileChooserAction.OPEN` or `Gtk.FileChooserAction.SELECT_FOLDER`.

set_local_only(*local_only*)
Sets whether only local files can be selected.

set_show_hidden(*show_hidden*)
Sets whether to display hidden files and folders.

set_do_overwrite_confirmation(*do_overwrite_confirmation*)
Sets whether to confirm overwriting in save mode.

get_filename()
Returns the filename for the currently selected file in the file selector. If multiple files are selected, use `get_filenames()` instead.

get_filenames()
Returns a list of all the selected files and subfolders in the current folder. The returned names are full absolute paths. If files in the current folder cannot be represented as local filenames they will be ignored. Use `get_uris()` instead.

get_uri()
Returns the URI for the currently selected file in the file selector. If multiple files are selected, use `get_uris()` instead.

get_uris()
Returns a list of all the selected files and subfolders in the current folder. The returned names are full absolute URIs.

add_filter(*filter*)
Adds the `Gtk.FileFilter` instance *filter* to the list of filters that the user can choose from. When a filter is selected, only files that are passed by that filter are displayed.

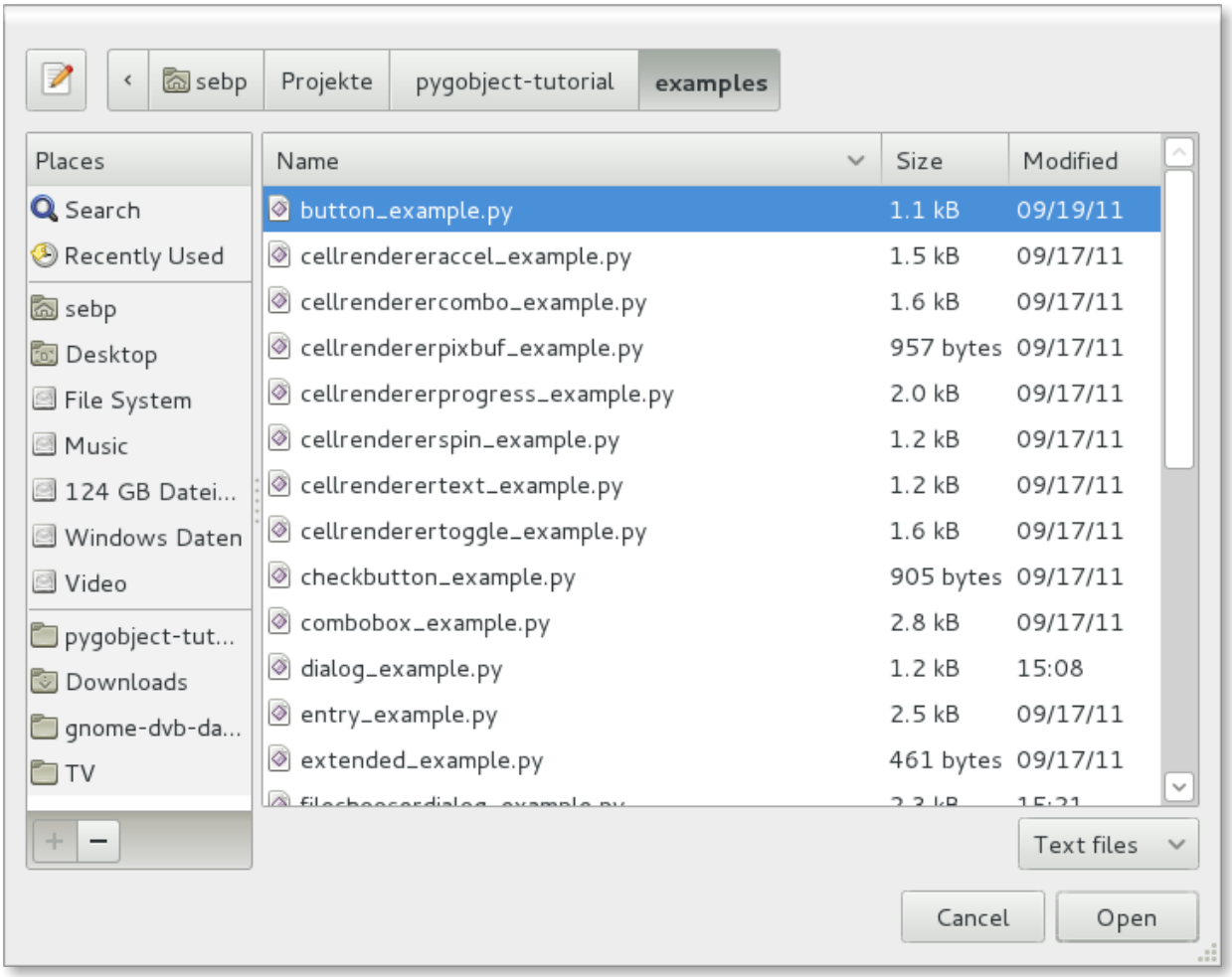
class `Gtk.`**FileFilter**

set_name(*name*)
Sets the human-readable name of the filter; this is the string that will be displayed in the file selector user interface if there is a selectable list of filters.

add_mime_type(*mime_type*)
Adds a rule allowing a given mime type to filter.

add_pattern(*pattern*)
Adds a rule allowing a shell style glob to a filter.

17.3.2 Example




```
from gi.repository import Gtk

class FileChooserWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="FileChooser Example")

        box = Gtk.Box(spacing=6)
        self.add(box)

        button1 = Gtk.Button("Choose File")
        button1.connect("clicked", self.on_file_clicked)
        box.add(button1)

        button2 = Gtk.Button("Choose Folder")
        button2.connect("clicked", self.on_folder_clicked)
        box.add(button2)

    def on_file_clicked(self, widget):
        dialog = Gtk.FileChooserDialog("Please choose a file", self,
            Gtk.FileChooserAction.OPEN,
            (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
```

```
23              Gtk.STOCK_OPEN, Gtk.ResponseType.OK))
24
25          self.add_filters(dialog)
26
27          response = dialog.run()
28          if response == Gtk.ResponseType.OK:
29              print "Open clicked"
30              print "File selected: " + dialog.get_filename()
31          elif response == Gtk.ResponseType.CANCEL:
32              print "Cancel clicked"
33
34          dialog.destroy()
35
36      def add_filters(self, dialog):
37          filter_text = Gtk.FileFilter()
38          filter_text.set_name("Text files")
39          filter_text.add_mime_type("text/plain")
40          dialog.add_filter(filter_text)
41
42          filter_py = Gtk.FileFilter()
43          filter_py.set_name("Python files")
44          filter_py.add_mime_type("text/x-python")
45          dialog.add_filter(filter_py)
46
47          filter_any = Gtk.FileFilter()
48          filter_any.set_name("Any files")
49          filter_any.add_pattern("*")
50          dialog.add_filter(filter_any)
51
52      def on_folder_clicked(self, widget):
53          dialog = Gtk.FileChooserDialog("Please choose a folder", self,
54              Gtk.FileChooserAction.SELECT_FOLDER,
55              (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
56               "Select", Gtk.ResponseType.OK))
57          dialog.set_default_size(800, 400)
58
59          response = dialog.run()
60          if response == Gtk.ResponseType.OK:
61              print "Select clicked"
62              print "Folder selected: " + dialog.get_filename()
63          elif response == Gtk.ResponseType.CANCEL:
64              print "Cancel clicked"
65
66          dialog.destroy()
67
68  win = FileChooserWindow()
69  win.connect("delete-event", Gtk.main_quit)
70  win.show_all()
71  Gtk.main()
```

CHAPTER

EIGHTEEN

CLIPBOARD

`Gtk.Clipboard` provides a storage area for a variety of data, including text and images. Using a clipboard allows this data to be shared between applications through actions such as copying, cutting, and pasting. These actions are usually done in three ways: using keyboard shortcuts, using a `Gtk.MenuItem`, and connecting the functions to `Gtk.Button` widgets.

There are multiple clipboard selections for different purposes. In most circumstances, the selection named `CLIPBOARD` is used for everyday copying and pasting. `PRIMARY` is another common selection which stores text selected by the user with the cursor.

18.1 Clipboard Objects

class `Gtk.`**`Clipboard`**

`get`(*selection*)
Obtains the `Gtk.Clipboard` for the given selection.

selection is a `Gdk.Atom` describing which clipboard to use. Predefined values include:

- `Gdk.SELECTION_CLIPBOARD`
- `Gdk.SELECTION_PRIMARY`

`set_text`(*text*, *length*)
Sets the contents of the clipboard to the given text.

text is the string to put in the clipboard.

length is the number of characters to store. It may be omitted to store the entire string.

`set_image`(*image*)
Sets the contents of the clipboard to the given image.

image must be a `Gdk.Pixbuf`. To retrieve one from a `Gdk.Image`, use `image.get_pixbuf()`.

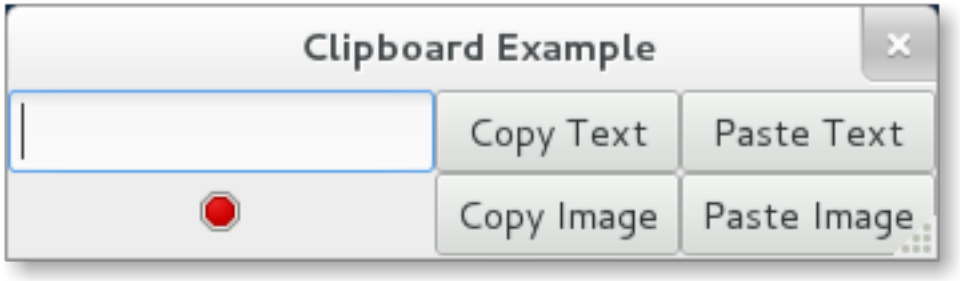
`wait_for_text`()
Returns the clipboard's content as a string, or returns `None` if the clipboard is empty or not currently holding text.

`wait_for_image`()
Returns the clipboard's content as a `Gtk.Pixbuf`, or returns `None` if the clipboard is empty or not currently holding an image.

store()
Stores the clipboard's data outside the application. Otherwise, data copied to the clipboard may be lost when the application exits.

clear()
Clears the contents of the clipboard. Use with caution; this may clear data from another application.

18.2 Example




```
from gi.repository import Gtk, Gdk

class ClipboardWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Clipboard Example")

        table = Gtk.Table(3, 2)

        self.clipboard = Gtk.Clipboard.get(Gdk.SELECTION_CLIPBOARD)
        self.entry = Gtk.Entry()
        self.image = Gtk.Image.new_from_stock(Gtk.STOCK_STOP, Gtk.IconSize.MENU)

        button_copy_text = Gtk.Button("Copy Text")
        button_paste_text = Gtk.Button("Paste Text")
        button_copy_image = Gtk.Button("Copy Image")
        button_paste_image = Gtk.Button("Paste Image")

        table.attach(self.entry, 0, 1, 0, 1)
        table.attach(self.image, 0, 1, 1, 2)
        table.attach(button_copy_text, 1, 2, 0, 1)
        table.attach(button_paste_text, 2, 3, 0, 1)
        table.attach(button_copy_image, 1, 2, 1, 2)
        table.attach(button_paste_image, 2, 3, 1, 2)

        button_copy_text.connect("clicked", self.copy_text)
        button_paste_text.connect("clicked", self.paste_text)
        button_copy_image.connect("clicked", self.copy_image)
        button_paste_image.connect("clicked", self.paste_image)

        self.add(table)

    def copy_text(self, widget):
        self.clipboard.set_text(self.entry.get_text(), -1)

    def paste_text(self, widget):
```

```
37          text = self.clipboard.wait_for_text()
38          if text != None:
39              self.entry.set_text(text)
40          else:
41              print "No text on the clipboard."
42
43      def copy_image(self, widget):
44          if self.image.get_storage_type() == Gtk.ImageType.PIXBUF:
45              self.clipboard.set_image(self.image.get_pixbuf())
46          else:
47              print "No image has been pasted yet."
48
49      def paste_image(self, widget):
50          image = self.clipboard.wait_for_image()
51          if image != None:
52              self.image.set_from_pixbuf(image)
53
54
55  win = ClipboardWindow()
56  win.connect("delete-event", Gtk.main_quit)
57  win.show_all()
58  Gtk.main()
```

CHAPTER

NINETEEN

DRAG AND DROP

Note: Versions of PyGObject < 3.0.3 contain a bug which does not allow drag and drop to function correctly. Therefore a version of PyGObject >= 3.0.3 is required for the following examples to work.

Setting up drag and drop between widgets consists of selecting a drag source (the widget which the user starts the drag from) with the `Gtk.Widget.drag_source_set()` method, selecting a drag destination (the widget which the user drops onto) with the `Gtk.Widget.drag_dest_set()` method and then handling the relevant signals on both widgets.

Instead of using `Gtk.Widget.drag_source_set()` and `Gtk.Widget.drag_dest_set()` some specialised widgets require the use of specific functions (such as `Gtk.TreeView` and `Gtk.IconView`).

A basic drag and drop only requires the source to connect to the "drag-data-get" signal and the destination to connect to the "drag-data-received" signal. More complex things such as specific drop areas and custom drag icons will require you to connect to *additional signals* and interact with the `Gdk.DragContext` object it supplies.

In order to transfer data between the source and destination, you must interact with the `Gtk.SelectionData` variable supplied in the *"drag-data-get"* and *"drag-data-received"* signals using the `Gtk.SelectionData` get and set methods.

19.1 Target Entries

To allow the drag source and destination to know what data they are receiving and sending, a common list of `Gtk.TargetEntry's` are required. A `Gtk.TargetEntry` describes a piece of data that will be sent by the drag source and received by the drag destination.

There are two ways of adding `Gtk.TargetEntry's` to a source and destination. If the drag and drop is simple and each target entry is of a different type, you can use the group of methods `mentioned here`.

If you require more than one type of data or wish to do more complex things with the data, you will need to create the `Gtk.TargetEntry's` using the `Gtk.TargetEntry.new()` method.

19.2 Drag and Drop Methods and Objects

class `Gtk.`**`Widget`**

`drag_source_set`(*start_button_mask*, *targets*, *actions*)

Sets the widget to be a drag source.

start_button_mask are a combination of `Gdk.ModifierType` masks which sets which buttons must be pressed for a drag to occur. *targets* is a list of `Gtk.TargetEntry's` which describe the data to be passed between source and destination. *actions* are a combination `Gdk.DragAction` masks to show possible drag actions.

drag_dest_set (*flags*, *targets*, *actions*)

Sets the widget to be a drag destination.

flags are a combination of `Gtk.DestDefaults` masks which configures the processes which occur on a drag site. *targets* is a list of `Gtk.TargetEntry's` which describe the data to be passed between source and destination. *actions* are a combination `Gdk.DragAction` masks to show possible drag actions.

drag_source_add_text_targets()
drag_dest_add_text_targets()

Add a `Gtk.TargetEntry` to the drag source/destination which contains a piece of text.

drag_source_add_image_targets()
drag_dest_add_image_targets()

Add a `Gtk.TargetEntry` to the drag source/destination which contains a `GdkPixbuf.Pixbuf`.

drag_source_add_uri_targets()
drag_dest_add_uri_targets()

Add a `Gtk.TargetEntry` to the drag source/destination which contains a list of URIs.

class `Gtk.`**TargetEntry**

static new (*target*, *flags*, *info*)

Creates a new target entry.

target is a string describing the type of data the target entry describes.

flags controls under which conditions will the data be transferred in a drag and drop and is a combination of the `Gtk.TargetFlags` values:

- `Gtk.TargetFlags.SAME_APP` - Only transferred in the same application
- `Gtk.TargetFlags.SAME_WIDGET` - Only transferred within the same widget
- `Gtk.TargetFlags.OTHER_APP` - Only transferred in a different application
- `Gtk.TargetFlags.OTHER_WIDGET` - Only transferred within a different widget

info is an ID which the application can use to determine between different pieces of data contained in a drag and drop operation.

class `Gtk.`**SelectionData**

get_text()

Returns the contents of the text contained in selection data

set_text (*text*)

Sets the contents of the text contained in selection data to *text*

get_pixbuf()

Returns the pixbuf contained in selection data

set_pixbuf (*pixbuf*)

Sets the pixbuf contained in selection data to *pixbuf*

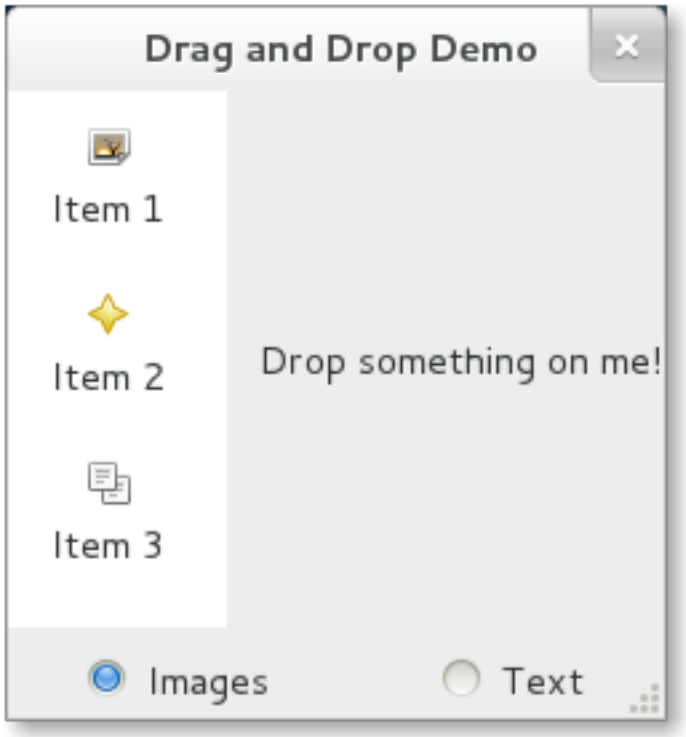
19.3 Drag Source Signals

Name	When it is emitted	Common Purpose
drag-begin	User starts a drag	Set-up drag icon
drag-data-get	When drag data is requested by the destination	Transfer drag data from source to destination
drag-data-delete	When a drag with the action Gdk.DragAction.MOVE is completed	Delete data from the source to complete the 'move'
drag-data-end	When the drag is complete	Undo anything done in drag-begin

19.4 Drag Destination Signals

Name	When it is emitted	Common Purpose
drag-motion	Drag icon moves over a drop area	Allow only certain areas to be dropped onto
drag-drop	Icon is dropped onto a drag area	Allow only certain areas to be dropped onto
drag-data-received	When drag data is received by the destination	Transfer drag data from source to destination

19.5 Example



```
from gi.repository import Gtk, Gdk, GdkPixbuf

(TARGET_ENTRY_TEXT, TARGET_ENTRY_PIXBUF) = range(2)
(COLUMN_TEXT, COLUMN_PIXBUF) = range(2)

DRAG_ACTION = Gdk.DragAction.COPY

class DragDropWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Drag and Drop Demo")

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
        self.add(vbox)

        hbox = Gtk.Box(spacing=12)
        vbox.pack_start(hbox, True, True, 0)

        self.iconview = DragSourceIconView()
        self.drop_area = DropArea()

        hbox.pack_start(self.iconview, True, True, 0)
        hbox.pack_start(self.drop_area, True, True, 0)

        button_box = Gtk.Box(spacing=6)
        vbox.pack_start(button_box, True, False, 0)

        image_button = Gtk.RadioButton.new_with_label_from_widget(None,
            "Images")
        image_button.connect("toggled", self.add_image_targets)
        button_box.pack_start(image_button, True, False, 0)

        text_button = Gtk.RadioButton.new_with_label_from_widget(image_button,
            "Text")
        text_button.connect("toggled", self.add_text_targets)
        button_box.pack_start(text_button, True, False, 0)

        self.add_image_targets()

    def add_image_targets(self, button=None):
        targets = Gtk.TargetList.new([])
        targets.add_image_targets(TARGET_ENTRY_PIXBUF, True)

        self.drop_area.drag_dest_set_target_list(targets)
        self.iconview.drag_source_set_target_list(targets)

    def add_text_targets(self, button=None):
        self.drop_area.drag_dest_set_target_list(None)
        self.iconview.drag_source_set_target_list(None)

        self.drop_area.drag_dest_add_text_targets()
        self.iconview.drag_source_add_text_targets()

class DragSourceIconView(Gtk.IconView):

    def __init__(self):
        Gtk.IconView.__init__(self)
        self.set_text_column(COLUMN_TEXT)
```

```
59          self.set_pixbuf_column(COLUMN_PIXBUF)
60
61          model = Gtk.ListStore(str, GdkPixbuf.Pixbuf)
62          self.set_model(model)
63          self.add_item("Item 1", "image")
64          self.add_item("Item 2", "gtk-about")
65          self.add_item("Item 3", "edit-copy")
66
67          self.enable_model_drag_source(Gdk.ModifierType.BUTTON1_MASK, [], DRAG_ACTION)
68          self.connect("drag-data-get", self.on_drag_data_get)
69
70      def on_drag_data_get(self, widget, drag_context, data, info, time):
71          selected_path = self.get_selected_items()[0]
72          selected_iter = self.get_model().get_iter(selected_path)
73
74          if info == TARGET_ENTRY_TEXT:
75              text = self.get_model().get_value(selected_iter, COLUMN_TEXT)
76              data.set_text(text, -1)
77          elif info == TARGET_ENTRY_PIXBUF:
78              pixbuf = self.get_model().get_value(selected_iter, COLUMN_PIXBUF)
79              data.set_pixbuf(pixbuf)
80
81      def add_item(self, text, icon_name):
82          pixbuf = Gtk.IconTheme.get_default().load_icon(icon_name, 16, 0)
83          self.get_model().append([text, pixbuf])
84
85
86  class DropArea(Gtk.Label):
87
88      def __init__(self):
89          Gtk.Label.__init__(self, "Drop something on me!")
90          self.drag_dest_set(Gtk.DestDefaults.ALL, [], DRAG_ACTION)
91
92          self.connect("drag-data-received", self.on_drag_data_received)
93
94      def on_drag_data_received(self, widget, drag_context, x, y, data, info, time):
95          if info == TARGET_ENTRY_TEXT:
96              text = data.get_text()
97              print "Received text: %s" % text
98
99          elif info == TARGET_ENTRY_PIXBUF:
100             pixbuf = data.get_pixbuf()
101             width = pixbuf.get_width()
102             height = pixbuf.get_height()
103
104             print "Received pixbuf with width %spx and height %spx" % (width, height)
105
106 win = DragDropWindow()
107 win.connect("delete-event", Gtk.main_quit)
108 win.show_all()
109 Gtk.main()
```

CHAPTER

TWENTY

GLADE AND GTK.BUILDER

The `Gtk.Builder` class offers you the opportunity to design user interfaces without writing a single line of code. This is possible through describing the interface by a XML file and then loading the XML description at runtime and create the objects automatically, which the Builder class does for you. For the purpose of not needing to write the XML manually the Glade application lets you create the user interface in a WYSIWYG (what you see is what you get) manner

This method has several advantages:

- Less code needs to be written.
- UI changes can be seen more quickly, so UIs are able to improve.
- Designers without programming skills can create and edit UIs.
- The description of the user interface is independent from the programming language being used.

There is still code required for handling interface changes triggered by the user, but `Gtk.Builder` allows you to focus on implementing that functionality.

20.1 Creating and loading the .glade file

First of all you have to download and install Glade. There are several tutorials about Glade, so this is not explained here in detail. Let's start by creating a window with a button in it and saving it to a file named *example.glade*. The resulting XML file should look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <child>
      <object class="GtkButton" id="button1">
        <property name="label" translatable="yes">button</property>
        <property name="use_action_appearance">False</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="receives_default">True</property>
        <property name="use_action_appearance">False</property>
      </object>
    </child>
  </object>
</interface>
```

To load this file in Python we need a `Gtk.Builder` object.

```
builder = Gtk.Builder()
builder.add_from_file("example.glade")
```

The second line loads all objects defined in *example.glade* into the Builder object.

It is also possible to load only some of the objects. The following line would add only the objects (and their child objects) given in the tuple.

```
# we don't really have two buttons here, this is just an example
builder.add_objects_from_file("example.glade", ("button1", "button2"))
```

These two methods exist also for loading from a string rather than a file. Their corresponding names are `Gtk.Builder.add_from_string()` and `Gtk.Builder.add_objects_from_string()` and they simply take a XML string instead of a file name.

20.2 Accessing widgets

Now that the window and the button are loaded we also want to show them. Therefore the `Gtk.Window.show_all()` method has to be called on the window. But how do we access the associated object?

```
window = builder.get_object("window1")
window.show_all()
```

Every widget can be retrieved from the builder by the `Gtk.Builder.get_object()` method and the widget's *id*. It is really *that* simple.

It is also possible to get a list of all objects with

```
builder.get_objects()
```

20.3 Connecting Signals

Glade also makes it possible to define signals which you can connect to handlers in your code without extracting every object from the builder and connecting to the signals manually. The first thing to do is to declare the signal names in Glade. For this example we will act when the window should be closed and when the button was pressed, so we give the name "onDeleteWindow" to the "delete-event" signal of the window and "onButtonPressed" to the "pressed" signal of the button. Now the XML file should look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <object class="GtkWindow" id="window1">
    <property name="can_focus">False</property>
    <signal name="delete-event" handler="onDeleteWindow" swapped="no"/>
    <child>
      <object class="GtkButton" id="button1">
        <property name="label" translatable="yes">button</property>
        <property name="use_action_appearance">False</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="receives_default">True</property>
        <property name="use_action_appearance">False</property>
```

```
        <signal name="pressed" handler="onButtonPressed" swapped="no"/>
      </object>
    </child>
  </object>
</interface>
```

Now we have to define the handler functions in our code. The *onDeleteWindow* should simply result in a call to `Gtk.main_quit()`. When the button is pressed we would like to print the string "Hello World!", so we define the handler as follows

```
def hello(button):
    print "Hello World!"
```

Next, we have to connect the signals and the handler functions. The easiest way to do this is to define a *dict* with a mapping from the names to the handlers and then pass it to the `Gtk.Builder.connect_signals()` method.

```
handlers = {
    "onDeleteWindow": Gtk.main_quit,
    "onButtonPressed": hello
}
builder.connect_signals(handlers)
```

An alternative approach is to create a class which has methods that are called like the signals. In our example the last code snippet could be rewritten as:

```
1  class Handler:
2      def onDeleteWindow(self, *args):
3          Gtk.main_quit(*args)
4
5      def onButtonPressed(self, button):
6          print "Hello World!"
7
8  builder = Gtk.Builder()
9  builder.add_from_file("builder_example.glade")
10 builder.connect_signals(Handler())
```

20.4 Builder Objects

class `Gtk.`**`Builder`**

`add_from_file`(*filename*)
Loads and parses the given file and merges it with the current contents of builder.

`add_from_string`(*string*)
Parses the given string and merges it with the current contents of builder..

`add_objects_from_file`(*filename*, *object_ids*)
Same as `Gtk.Builder.add_from_file()`, but loads only the objects with the ids given in the *object_ids* list.

`add_objects_from_string`(*filename*, *object_ids*)
Same as `Gtk.Builder.add_from_string()`, but loads only the objects with the ids given in the *object_ids* list.

`get_object`(*object_id*)
Retrieves the widget with the id *object_id* from the loaded objects in the builder.

get_objects()
Returns all loaded objects.

connect_signals(*handler_object*)
Connects the signals to the methods given in the *handler_object*. The *handler_object* can be any object which contains keys or attributes that are called like the signal handler names given in the interface description, e.g. a class or a dict.

20.5 Example

The final code of the example

```
from gi.repository import Gtk

class Handler:
    def onDeleteWindow(self, *args):
        Gtk.main_quit(*args)

    def onButtonPressed(self, button):
        print "Hello World!"

builder = Gtk.Builder()
builder.add_from_file("builder_example.glade")
builder.connect_signals(Handler())

window = builder.get_object("window1")
window.show_all()

Gtk.main()
```

CHAPTER

TWENTYONE

OBJECTS

GObject is the fundamental type providing the common attributes and methods for all object types in GTK+, Pango and other libraries based on GObject. The `GObject.GObject` class provides methods for object construction and destruction, property access methods, and signal support.

This section will introduce some important aspects about the GObject implementation in python.

21.1 Inherit from GObject.GObject

A native GObject is accessible via `GObject.GObject`. It is rarely instantiated directly, we generally use inherited class. A `Gtk.Widget` is an inherited class of a `GObject.GObject`. It may be interesting to make an inherited class to create a new widget, like a settings dialog.

To inherit from `GObject.GObject`, you must call `GObject.GObject.__init__()` in your constructor (if the class inherits from `Gtk.Button`, it must call `Gtk.Button.__init__()` for instance), like in the example below:

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    def __init__(self):
        GObject.GObject.__init__(self)
```

21.2 Signals

Signals connect arbitrary application-specific events with any number of listeners. For example, in GTK+, every user event (keystroke or mouse move) is received from the X server and generates a GTK+ event under the form of a signal emission on a given object instance.

Each signal is registered in the type system together with the type on which it can be emitted: users of the type are said to connect to the signal on a given type instance when they register a function to be invoked upon the signal emission. Users can also emit the signal by themselves or stop the emission of the signal from within one of the functions connected to the signal.

21.2.1 Receive signals

See *Main loop and Signals*

21.2.2 Create new signals

New signals can be created by adding them to GObject.GObject.__gsignals__, a dictionary:

When a new signal is created, a method handler can also be defined, it will be called each time the signal is emitted. It is called do_signal_name.

```
class MyObject(GObject.GObject):
    __gsignals__ = {
        'my_signal': (GObject.SIGNAL_RUN_FIRST, None,
                      (int,))
    }

    def do_my_signal(self, arg):
        print "class method for 'my_signal' called with argument", arg
```

GObject.SIGNAL_RUN_FIRST indicates that this signal will invoke the object method handler (do_my_signal() here) in the first emission stage. Alternatives are GObject.SIGNAL_RUN_LAST (the method handler will be invoked in the third emission stage) and GObject.SIGNAL_RUN_CLEANUP (invoke the method handler in the last emission stage).

The second part, None, indicates the return type of the signal, usually None.

(int,) indicates the signal arguments, here, the signal will only take one argument, whose type is int. This argument type list must end with a comma.

Signals can be emitted using GObject.GObject.emit():

```
my_obj.emit("my_signal", 42) # emit the signal "my_signal", with the
                             # argument 42
```

21.3 Properties

One of GObject's nice features is its generic get/set mechanism for object properties. Each class inherited from GObject.GObject can define new properties. Each property as a type which never changes (e.g. str, float, int...). For instance, they are used for Gtk.Button where there is a "label" property which contains the text of the button.

21.3.1 Use existing properties

The class GObject.GObject provides several useful functions to manage existing properties, GObject.GObject.get_property() and GObject.GObject.set_property().

Some properties also have functions dedicated to them, called getter and setter. For the property "label" of a button, there are two functions to get and set them, Gtk.Button.get_label() and Gtk.Button.set_label().

21.3.2 Create new properties

A property is defined with a name and a type. Even is python itself is dynamically typed, you can't change the type of a property once it is defined. A property can be created using GObject.property().

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    foo = GObject.property(type=str, default='bar')
```

```
    property_float = GObject.property(type=float)
    def __init__(self):
        GObject.GObject.__init__(self)
```

Properties can also be read-only, if you want some properties to be readable but not writable. To do so, you can add some flags to the property definition, to control read/write access. Flags are `GObject.PARAM_READABLE` (only read access for external code), `GObject.PARAM_WRITABLE` (only write access), `GObject.PARAM_READWRITE` (public):

```
foo = GObject.property(type=str, flags = GObject.PARAM_READABLE) # won't be writable
bar = GObject.property(type=str, flags = GObject.PARAM_WRITABLE) # won't be readable
```

You can also define new read-only properties with a new method decorated with `GObject.property()`:

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    def __init__(self):
        GObject.GObject.__init__(self)

    @GObject.property
    def readonly(self):
        return 'This is read-only.'
```

You can get this property using:

```
my_object = MyObject()
print my_object.readonly
print my_object.get_property("readonly")
```

There is also a way to define minimum and maximum values for numbers, using a more verbose form:

```
from gi.repository import GObject

class MyObject(GObject.GObject):

    __gproperties__ = {
        "int-prop": (int, # type
                     "integer prop", # nick
                     "A property that contains an integer", # blurb
                     1, # min
                     5, # max
                     2, # default
                     GObject.PARAM_READWRITE # flags
                    ),
    }

    def __init__(self):
        GObject.GObject.__init__(self)
        self.int_prop = 2

    def do_get_property(self, prop):
        if prop.name == 'int-prop':
            return self.int_prop
        else:
            raise AttributeError, 'unknown property %s' % prop.name

    def do_set_property(self, prop, value):
```

```
        if prop.name == 'int-prop':
            self.int_prop = value
        else:
            raise AttributeError, 'unknown property %s' % prop.name
```

Properties must be defined in `GObject.GObject.__gproperties__`, a dictionary, and handled in do_get_property and do_set_property.

21.3.3 Watch properties

When a property is modified, a signal is emitted, whose name is "notify::property_name":

```
my_object = MyObject()

def on_notify_foo(obj, gparamstring):
    print "foo changed"

my_object.connect("notify::foo", on_notify_foo)

my_object.set_property("foo", "bar") # on_notify_foo will be called
```

21.4 API

class `GObject.`**GObject**

get_property(*property_name*)
Retrieves a property value.

set_property(*property_name*, *value*)
Set property *property_name* to *value*.

emit(*signal_name*, ...)
Emit signal *signal_name*. Signal arguments must follow, e.g. if your signal is of type `(int,)`, it must be emitted with:

```
self.emit(signal_name, 42)
```

freeze_notify()
This method freezes all the "notify::" signals (which are emitted when any property is changed) until the `thaw_notify()` method is called.

It recommended to use the *with* statement when calling `freeze_notify()`, that way it is ensured that `thaw_notify()` is called implicitly at the end of the block:

```
with an_object.freeze_notify():
    # Do your work here
    ...
```

thaw_notify()
Thaw all the "notify::" signals which were thawed by `freeze_notify()`.

It is recommended to not call `thaw_notify()` explicitly but use `freeze_notify()` together with the *with* statement.

handler_block(*handler_id*)
Blocks a handler of an instance so it will not be called during any signal emissions unless `handler_unblock()` is called for that *handler_id*. Thus "blocking" a signal handler means to temporarily deactivate it, a signal handler has to be unblocked exactly the same amount of times it has been blocked before to become active again.

It is recommended to use `handler_block()` in conjunction with the *with* statement which will call `handler_unblock()` implicitly at the end of the block:

```
with an_object.handler_block(handler_id):
    # Do your work here
    ...
```

handler_unblock(*handler_id*)
Undoes the effect of `handler_block()`. A blocked handler is skipped during signal emissions and will not be invoked until it has been unblocked exactly the amount of times it has been blocked before.

It is recommended to not call `handler_unblock()` explicitly but use `handler_block()` together with the *with* statement.

__gsignals__
A dictionary where inherited class can define new signals.

Each element in the dictionary is a new signal. The key is the signal name. The value is a tuple, with the form:

```
(GObject.SIGNAL_RUN_FIRST, None, (int,))
```

`GObject.SIGNAL_RUN_FIRST` can be replaced with `GObject.SIGNAL_RUN_LAST` or `GObject.SIGNAL_RUN_CLEANUP`. `None` is the return type of the signal. `(int,)` is the list of the parameters of the signal, it must end with a comma.

__gproperties__
The `__gproperties__` dictionary is a class property where you define the properties of your object. This is not the recommend way to define new properties, the method written above is much less verbose. The benefits of this method is that a property can be defined with more settings, like the minimum or the maximum for numbers.

The key is the name of the property

The value is a tuple which describe the property. The number of elements of this tuple depends on its first element but the tuple will always contain at least the following items:

The first element is the property's type (e.g. `int`, `float`...).

The second element is the property's nick name, which is a string with a short description of the property. This is generally used by programs with strong introspection capabilities, like the graphical user interface builder Glade.

The third one is the property's description or blurb, which is another string with a longer description of the property. Also used by Glade and similar programs.

The last one (which is not necessarily the forth one as we will see later) is the property's flags: `GObject.PARAM_READABLE`, `GObject.PARAM_WRITABLE`, `GObject.PARAM_READWRITE`.

The absolute length of the tuple depends on the property type (the first element of the tuple). Thus we have the following situations:

If the type is `bool` or `str`, the forth element is the default value of the property.

If the type is `int` or `float`, the forth element is the minimum accepted value, the fifth element is the maximum accepted value and the sixth element is the default value.

If the type is not one of these, there is no extra element.

`GObject.`**`SIGNAL_RUN_FIRST`**
Invoke the object method handler in the first emission stage.

`GObject.`**`SIGNAL_RUN_LAST`**
Invoke the object method handler in the third emission stage.

`GObject.`**`SIGNAL_RUN_CLEANUP`**
Invoke the object method handler in the last emission stage.

`GObject.`**`PARAM_READABLE`**
The property is readable.

`GObject.`**`PARAM_WRITABLE`**
The property is writable.

`GObject.`**`PARAM_READWRITE`**
The property is readable and writable.

CHAPTER

TWENTYTWO

STOCK ITEMS

Stock items represent commonly-used menu or toolbar items such as “Open” or “Exit”. Each stock item is identified by a stock ID; stock IDs are just strings, but constants such as `Gtk.STOCK_OPEN` are provided to avoid typing mistakes in the strings.

`Gtk.`**`STOCK_ABOUT`**

`Gtk.`**`STOCK_ADD`**

`Gtk.`**`STOCK_APPLY`**

`Gtk.`**`STOCK_BOLD`**

`Gtk.`**`STOCK_CANCEL`**

`Gtk.`**`STOCK_CAPS_LOCK_WARNING`**

`Gtk.`**`STOCK_CDROM`**

`Gtk.`**`STOCK_CLEAR`**

`Gtk.`**`STOCK_CLOSE`**

`Gtk.`**`STOCK_COLOR_PICKER`**

`Gtk.`**`STOCK_CONNECT`**

Gtk.**STOCK_CONVERT**

Gtk.**STOCK_COPY**

Gtk.**STOCK_CUT**

Gtk.**STOCK_DELETE**

Gtk.**STOCK_DIALOG_AUTHENTICATION**

Gtk.**STOCK_DIALOG_INFO**

Gtk.**STOCK_DIALOG_WARNING**

Gtk.**STOCK_DIALOG_ERROR**

Gtk.**STOCK_DIALOG_QUESTION**

Gtk.**STOCK_DISCARD**

Gtk.**STOCK_DISCONNECT**

Gtk.**STOCK_DND**

Gtk.**STOCK_DND_MULTIPLE**

Gtk.**STOCK_EDIT**

Gtk.**STOCK_EXECUTE**

Gtk.**STOCK_FILE**

Gtk.**STOCK_FIND**

Gtk.**STOCK_FIND_AND_REPLACE**

Gtk.**STOCK_FLOPPY**

Gtk.**STOCK_FULLSCREEN**

Gtk.**STOCK_GOTO_BOTTOM**

Gtk.**STOCK_GOTO_FIRST**

LTR variant:

RTL variant:

Gtk.**STOCK_GOTO_LAST**

LTR variant:

RTL variant:

Gtk.**STOCK_GOTO_TOP**

Gtk.**STOCK_GO_BACK**

LTR variant:

RTL variant:

Gtk.**STOCK_GO_DOWN**

Gtk.**STOCK_GO_FORWARD**

LTR variant:

RTL variant:

Gtk.**STOCK_GO_UP**

Gtk.**STOCK_HARDDISK**

Gtk.**STOCK_HELP**

Gtk.**STOCK_HOME**

Gtk.**STOCK_INDEX**

Gtk.**STOCK_INDENT**

LTR variant:

RTL variant:

Gtk.**STOCK_INFO**

Gtk.**STOCK_ITALIC**

Gtk.**STOCK_JUMP_TO**

LTR variant:

RTL variant:

Gtk.**STOCK_JUSTIFY_CENTER**

Gtk.**STOCK_JUSTIFY_FILL**

Gtk.**STOCK_JUSTIFY_LEFT**

Gtk.**STOCK_JUSTIFY_RIGHT**

Gtk.**STOCK_LEAVE_FULLSCREEN**

Gtk.**STOCK_MISSING_IMAGE**

Gtk.**STOCK_MEDIA_FORWARD**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_NEXT**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_PAUSE**

Gtk.**STOCK_MEDIA_PLAY**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_PREVIOUS**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_RECORD**

Gtk.**STOCK_MEDIA_REWIND**

LTR variant:

RTL variant:

Gtk.**STOCK_MEDIA_STOP**

Gtk.**STOCK_NETWORK**

Gtk.**STOCK_NEW**

Gtk.**STOCK_NO**

Gtk.**STOCK_OK**

Gtk.**STOCK_OPEN**

Gtk.**STOCK_ORIENTATION_PORTRAIT**

Gtk.**STOCK_ORIENTATION_LANDSCAPE**

Gtk.**STOCK_ORIENTATION_REVERSE_LANDSCAPE**

Gtk.**STOCK_ORIENTATION_REVERSE_PORTRAIT**

Gtk.**STOCK_PAGE_SETUP**

Gtk.**STOCK_PASTE**

Gtk.**STOCK_PREFERENCES**

Gtk.**STOCK_PRINT**

Gtk.**STOCK_PRINT_ERROR**

Gtk.**STOCK_PRINT_PAUSED**

Gtk.**STOCK_PRINT_PREVIEW**

Gtk.**STOCK_PRINT_REPORT**

Gtk.**STOCK_PRINT_WARNING**

Gtk.**STOCK_PROPERTIES**

Gtk.**STOCK_QUIT**

Gtk.**STOCK_REDO**

LTR variant:

RTL variant:

Gtk.**STOCK_REFRESH**

Gtk.**STOCK_REMOVE**

Gtk.**STOCK_REVERT_TO_SAVED**

LTR variant:

RTL variant:

Gtk.**STOCK_SAVE**

Gtk.**STOCK_SAVE_AS**

Gtk.**STOCK_SELECT_ALL**

Gtk.**STOCK_SELECT_COLOR**

Gtk.**STOCK_SELECT_FONT**

Gtk.**STOCK_SORT_ASCENDING**

Gtk.**STOCK_SORT_DESCENDING**

Gtk.**STOCK_SPELL_CHECK**

Gtk.**STOCK_STOP**

Gtk.**STOCK_STRIKETHROUGH**

Gtk.**STOCK_UNDELETE**

LTR variant:

RTL variant:

Gtk.**STOCK_UNDERLINE**

Gtk.**STOCK_UNDO**

LTR variant:

RTL variant:

Gtk.**STOCK_UNINDENT**

LTR variant:

RTL variant:

Gtk.**STOCK_YES**

Gtk.**STOCK_ZOOM_100**

Gtk.**STOCK_ZOOM_FIT**

Gtk.**STOCK_ZOOM_IN**

CHAPTER

TWENTYTHREE

INDICES AND TABLES

- *search*