The Python GTK+ 3 Tutorial

Release 3.4

Sebastian Pölsterl

Sep 07, 2017

Contents

Release 3.4

Date Sep 07, 2017

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. Also there's a Python-specific reference available.

Contents:

CHAPTER 1

Installation

The first 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.

Dependencies

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

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.

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:

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

```
$ 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
```

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

```
$ jhbuild build gtk+-3
```

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

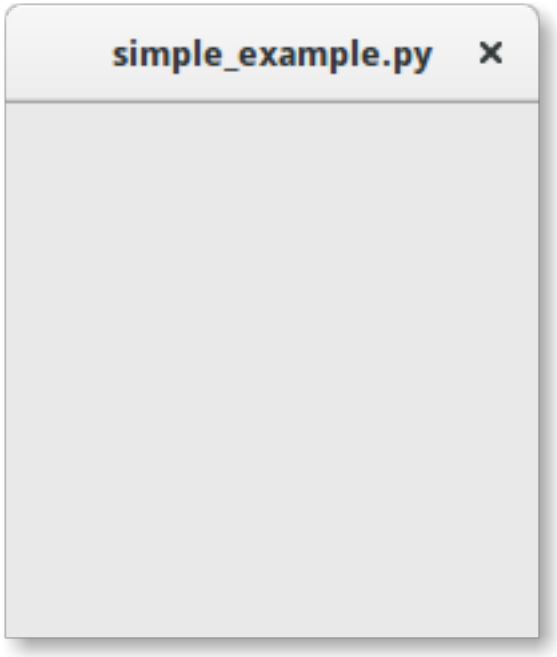
```
$ jhbuild shell
```

CHAPTER 2

Getting Started

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 import gi
2 gi.require_version('Gtk', '3.0')
3 from gi.repository import Gtk
4
5 win = Gtk.Window()
```

```
6 win.connect("delete-event", Gtk.main_quit)
7 win.show_all()
8 Gtk.main()
```

We will now explain each line of the example.

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk
```

In the beginning, we have to import the Gtk module to be able to access GTK+'s classes and functions. Since a user's system can have multiple versions of GTK+ installed at the same, we want to make sure that when we import Gtk that it refers to GTK+ 3 and not any other version of the library, which is the purpose of the statement `gi.require_version('Gtk', '3.0')`.

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).

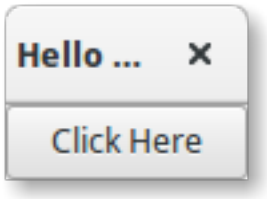
```
Gtk.main()
```

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

```
python simple_example.py
```

Extended Example

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




```
1 import gi
2 gi.require_version('Gtk', '3.0')
3 from gi.repository import Gtk
4
5 class MyWindow(Gtk.Window):
6
7     def __init__(self):
```

```
8           Gtk.Window.__init__(self, title="Hello World")
9
10          self.button = Gtk.Button(label="Click Here")
11          self.button.connect("clicked", self.on_button_clicked)
12          self.add(self.button)
13
14      def on_button_clicked(self, widget):
15          print("Hello World")
16
17  win = MyWindow()
18  win.connect("delete-event", Gtk.main_quit)
19  win.show_all()
20  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 3

Basics

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

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)
```

If you have lost the "handler_id" for some reason (for example the handlers were installed using `Gtk.Builder.connect_signals()`), you can still disconnect a specific callback using the function `disconnect_by_func()`:

```
widget.disconnect_by_func(callback)
```

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.

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 gobject properties through the "props" property such as `widget.props.prop_name = value`. This is equivalent to the more verbose `widget.get_property("prop-name")` and `widget.set_property("prop-name", value)`.

To see which properties are available for a widget in the running version of GTK you can "dir" the "props" property:

```
widget = Gtk.Box()
print(dir(widget.props))
```

This will print in the console the list of properties a Gtk.Box has.

CHAPTER 4

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.

Definitions

Conceptional, 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 form 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.

Python 2

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)
```

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 - interpreting them as being unequal
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
```

Python 3

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. Conceptional, `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'
```

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
```

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 5

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.

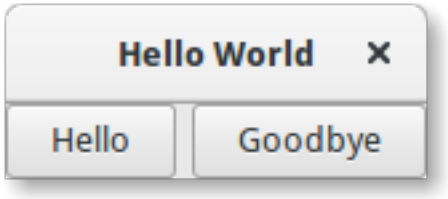
GTK+ arranges widgets hierarchically, using *containers*. They are invisible to the end user and are inserted into a window, or placed within each other to layout components. There are two flavours of containers: single-child containers, which are all descendants of `Gtk.Bin`, and multiple-child containers,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

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.

Example

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




```
import gi
gi.require_version('Gtk', '3.0')
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)

    def on_button1_clicked(self, widget):
        print("Hello")

    def on_button2_clicked(self, widget):
        print("Goodbye")

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

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. The `Gtk.Grid.attach()` method takes five parameters:

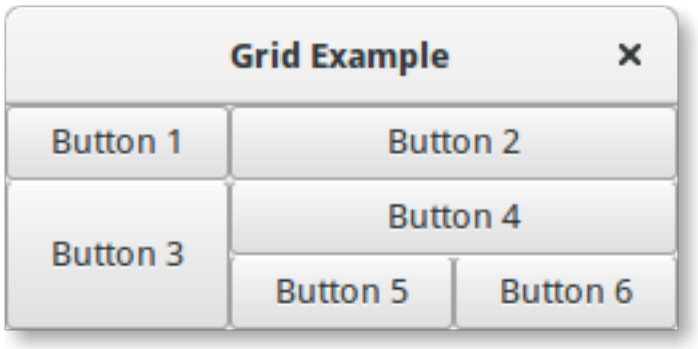
1. The `child` parameter is the `Gtk.Widget` to add.
2. `left` is the column number to attach the left side of `child` to.
3. `top` indicates the row number to attach the top side of `child` to.
4. `width` and `height` indicate the number of columns that the `child` will span, and the number of rows that the `child` will span, respectively.

It is also possible to add a child next to an existing child, using `Gtk.Grid.attach_next_to()`, which also takes five parameters:

1. `child` is the `Gtk.Widget` to add, as above.
2. `sibling` is an existing child widget of `self` (a `Gtk.Grid` instance) or `None`. The `child` widget will be placed next to `sibling`, or if `sibling` is `None`, at the beginning or end of the grid.
3. `side` is a `Gtk.PositionType` indicating the side of `sibling` that `child` is positioned next to.
4. `width` and `height` indicate the number of columns and rows the `child` widget will span, respectively.

Finally, `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`).

Example




```
import gi
gi.require_version('Gtk', '3.0')
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()
```

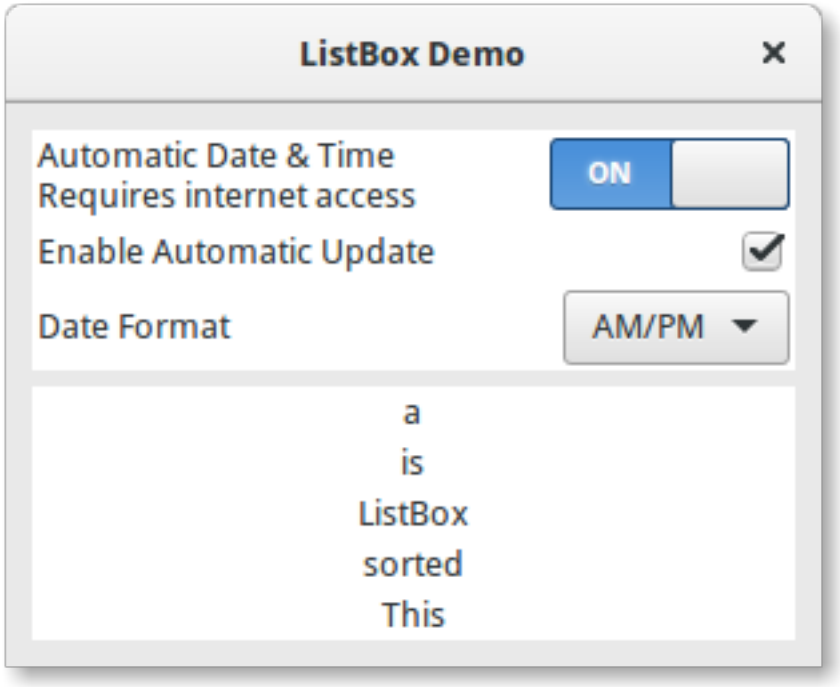
ListBox

A `Gtk.ListBox` is a vertical container that contains `Gtk.ListBoxRow` children. These rows can by dynamically sorted and filtered, and headers can be added dynamically depending on the row content. It also allows keyboard and mouse navigation and selection like a typical list.

Using `Gtk.ListBox` is often an alternative to `Gtk.TreeView`, especially when the list content has a more complicated layout than what is allowed by a `Gtk.CellRenderer`, or when the content is interactive (i.e. has a button in it).

Although a `Gtk.ListBox` must have only `Gtk.ListBoxRow` children, you can add any kind of widget to it via `Gtk.Container.add()` and a `Gtk.ListBoxRow` widget will automatically be inserted between the list and the widget.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class ListBoxRowWithData(Gtk.ListBoxRow):
    def __init__(self, data):
        super(Gtk.ListBoxRow, self).__init__()
        self.data = data
        self.add(Gtk.Label(data))

class ListBoxWindow(Gtk.Window):

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

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

        listbox = Gtk.ListBox()
        listbox.set_selection_mode(Gtk.SelectionMode.NONE)
        box_outer.pack_start(listbox, True, True, 0)

        row = Gtk.ListBoxRow()
        hbox = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=50)
        row.add(hbox)
        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        hbox.pack_start(vbox, True, True, 0)

        label1 = Gtk.Label("Automatic Date & Time", xalign=0)
        label2 = Gtk.Label("Requires internet access", xalign=0)
```

```
        vbox.pack_start(label1, True, True, 0)
        vbox.pack_start(label2, True, True, 0)

        switch = Gtk.Switch()
        switch.props.valign = Gtk.Align.CENTER
        hbox.pack_start(switch, False, True, 0)

        listbox.add(row)

        row = Gtk.ListBoxRow()
        hbox = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=50)
        row.add(hbox)
        label = Gtk.Label("Enable Automatic Update", xalign=0)
        check = Gtk.CheckButton()
        hbox.pack_start(label, True, True, 0)
        hbox.pack_start(check, False, True, 0)

        listbox.add(row)

        row = Gtk.ListBoxRow()
        hbox = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=50)
        row.add(hbox)
        label = Gtk.Label("Date Format", xalign=0)
        combo = Gtk.ComboBoxText()
        combo.insert(0, "0", "24-hour")
        combo.insert(1, "1", "AM/PM")
        hbox.pack_start(label, True, True, 0)
        hbox.pack_start(combo, False, True, 0)

        listbox.add(row)

        listbox_2 = Gtk.ListBox()
        items = 'This is a sorted ListBox Fail'.split()

        for item in items:
            listbox_2.add(ListBoxRowWithData(item))

        def sort_func(row_1, row_2, data, notify_destroy):
            return row_1.data.lower() > row_2.data.lower()

        def filter_func(row, data, notify_destroy):
            return False if row.data == 'Fail' else True

        listbox_2.set_sort_func(sort_func, None, False)
        listbox_2.set_filter_func(filter_func, None, False)

        listbox_2.connect('row-activated', lambda widget, row: print(row.data))

        box_outer.pack_start(listbox_2, True, True, 0)
        listbox_2.show_all()

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

Stack and StackSwitcher

A `Gtk.Stack` is a container which only shows one of its children at a time. In contrast to `Gtk.Notebook`, `Gtk.Stack` does not provide a means for users to change the visible child. Instead, the `Gtk.StackSwitcher` widget can be used with `Gtk.Stack` to provide this functionality.

Transitions between pages can be animated as slides or fades. This can be controlled with `Gtk.Stack.set_transition_type()`. These animations respect the "gtk-enable-animations" setting.

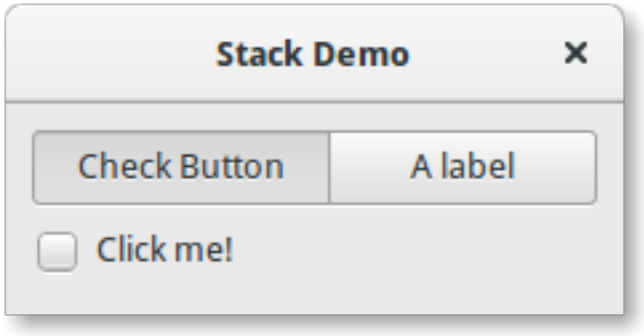
Transition speed can be adjusted with `Gtk.Stack.set_transition_duration()`

The `Gtk.StackSwitcher` widget acts as a controller for a `Gtk.Stack`; it shows a row of buttons to switch between the various pages of the associated stack widget.

All the content for the buttons comes from the child properties of the `Gtk.Stack`.

It is possible to associate multiple `Gtk.StackSwitcher` widgets with the same `Gtk.Stack` widget.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class StackWindow(Gtk.Window):

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

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

        stack = Gtk.Stack()
        stack.set_transition_type(Gtk.StackTransitionType.SLIDE_LEFT_RIGHT)
        stack.set_transition_duration(1000)

        checkbutton = Gtk.CheckButton("Click me!")
        stack.add_titled(checkbutton, "check", "Check Button")

        label = Gtk.Label()
        label.set_markup("<big>A fancy label</big>")
        stack.add_titled(label, "label", "A label")

        stack_switcher = Gtk.StackSwitcher()
```

```
        stack_switcher.set_stack(stack)
        vbox.pack_start(stack_switcher, True, True, 0)
        vbox.pack_start(stack, True, True, 0)

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

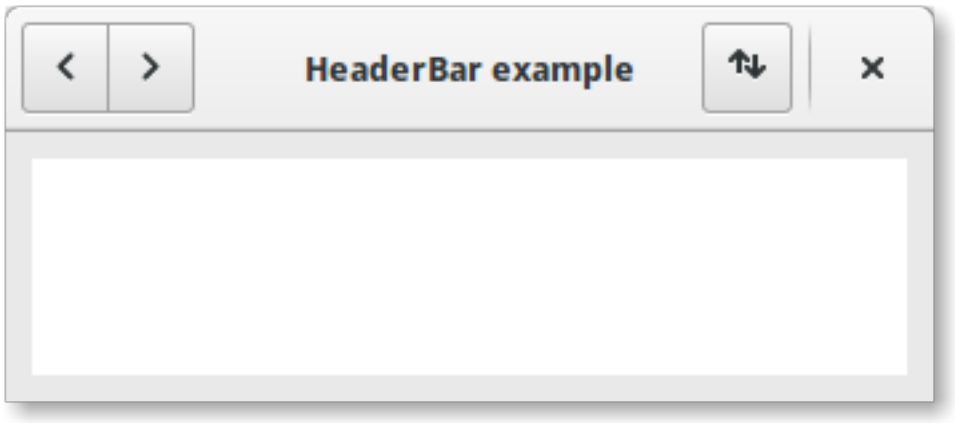
HeaderBar

A `Gtk.HeaderBar` is similar to a horizontal `Gtk.Box`, it allows to place children at the start or the end. In addition, it allows a title to be displayed. The title will be centered with respect to the width of the box, even if the children at either side take up different amounts of space.

Since GTK+ now supports Client Side Decoration, a `Gtk.HeaderBar` can be used in place of the title bar (which is rendered by the Window Manager).

A `Gtk.HeaderBar` is usually located across the top of a window and should contain commonly used controls which affect the content below. They also provide access to window controls, including the close window button and window menu.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk, Gio

class HeaderBarWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="HeaderBar Demo")
        self.set_border_width(10)
        self.set_default_size(400, 200)

        hb = Gtk.HeaderBar()
        hb.set_show_close_button(True)
        hb.props.title = "HeaderBar example"
```

```
        self.set_titlebar(hb)

        button = Gtk.Button()
        icon = Gio.ThemedIcon(name="mail-send-receive-symbolic")
        image = Gtk.Image.new_from_gicon(icon, Gtk.IconSize.BUTTON)
        button.add(image)
        hb.pack_end(button)

        box = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL)
        Gtk.StyleContext.add_class(box.get_style_context(), "linked")

        button = Gtk.Button()
        button.add(Gtk.Arrow(Gtk.ArrowType.LEFT, Gtk.ShadowType.NONE))
        box.add(button)

        button = Gtk.Button()
        button.add(Gtk.Arrow(Gtk.ArrowType.RIGHT, Gtk.ShadowType.NONE))
        box.add(button)

        hb.pack_start(box)

        self.add(Gtk.TextView())

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

FlowBox

A `Gtk.FlowBox` is a container that positions child widgets in sequence according to its orientation.

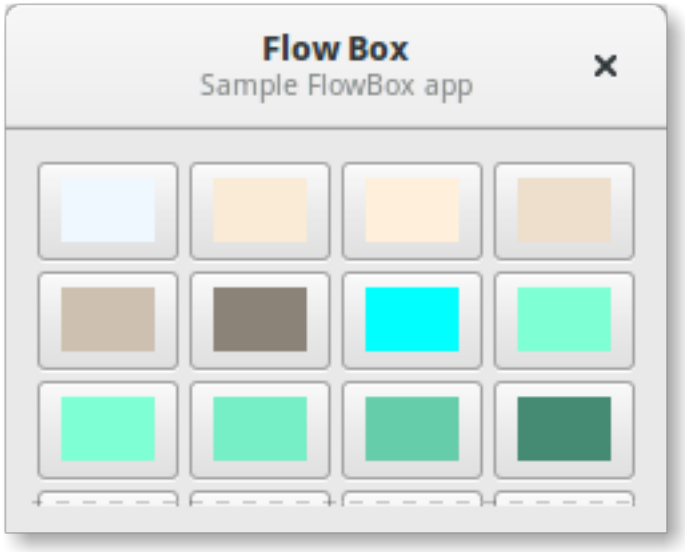
For instance, with the horizontal orientation, the widgets will be arranged from left to right, starting a new row under the previous row when necessary. Reducing the width in this case will require more rows, so a larger height will be requested.

Likewise, with the vertical orientation, the widgets will be arranged from top to bottom, starting a new column to the right when necessary. Reducing the height will require more columns, so a larger width will be requested.

The children of a `Gtk.FlowBox` can be dynamically sorted and filtered.

Although a `Gtk.FlowBox` must have only `Gtk.FlowBoxChild` children, you can add any kind of widget to it via `Gtk.Container.add()`, and a `Gtk.FlowBoxChild` widget will automatically be inserted between the box and the widget.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk, Gdk

class FlowBoxWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="FlowBox Demo")
        self.set_border_width(10)
        self.set_default_size(300, 250)

        header = Gtk.HeaderBar(title="Flow Box")
        header.set_subtitle("Sample FlowBox app")
        header.props.show_close_button = True

        self.set_titlebar(header)

        scrolled = Gtk.ScrolledWindow()
        scrolled.set_policy(Gtk.PolicyType.NEVER, Gtk.PolicyType.AUTOMATIC)

        flowbox = Gtk.FlowBox()
        flowbox.set_valign(Gtk.Align.START)
        flowbox.set_max_children_per_line(30)
        flowbox.set_selection_mode(Gtk.SelectionMode.NONE)

        self.create_flowbox(flowbox)

        scrolled.add(flowbox)

        self.add(scrolled)
        self.show_all()

    def color_swatch_new(self, str_color):
        color = Gdk.color_parse(str_color)

        rgba = Gdk.RGBA.from_color(color)
```

```
        button = Gtk.Button()

        area = Gtk.DrawingArea()
        area.set_size_request(24, 24)
        area.override_background_color(0, rgba)

        button.add(area)

        return button

    def create_flowbox(self, flowbox):
        colors = [
        'AliceBlue',
        'AntiqueWhite',
        'AntiqueWhite1',
        'AntiqueWhite2',
        'AntiqueWhite3',
        'AntiqueWhite4',
        'aqua',
        'aquamarine',
        'aquamarine1',
        'aquamarine2',
        'aquamarine3',
        'aquamarine4',
        'azure',
        'azure1',
        'azure2',
        'azure3',
        'azure4',
        'beige',
        'bisque',
        'bisque1',
        'bisque2',
        'bisque3',
        'bisque4',
        'black',
        'BlanchedAlmond',
        'blue',
        'blue1',
        'blue2',
        'blue3',
        'blue4',
        'BlueViolet',
        'brown',
        'brown1',
        'brown2',
        'brown3',
        'brown4',
        'burlywood',
        'burlywood1',
        'burlywood2',
        'burlywood3',
        'burlywood4',
        'CadetBlue',
        'CadetBlue1',
        'CadetBlue2',
        'CadetBlue3',
        'CadetBlue4',
```

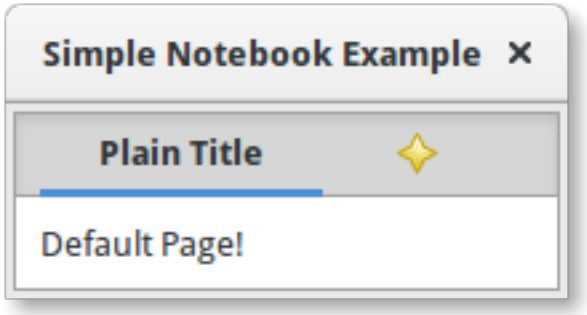
```
        'chartreuse',
        'chartreuse1',
        'chartreuse2',
        'chartreuse3',
        'chartreuse4',
        'chocolate',
        'chocolate1',
        'chocolate2',
        'chocolate3',
        'chocolate4',
        'coral',
        'coral1',
        'coral2',
        'coral3',
        'coral4'
        ]

        for color in colors:
            button = self.color_swatch_new(color)
            flowbox.add(button)


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

Notebook

The `Gtk.Notebook` widget is a `Gtk.Container` whose children are pages that can be switched between using tab labels along one edge.

There are many configuration options for GtkNotebook. Among other things, you can choose on which edge the tabs appear (see `Gtk.Notebook.set_tab_pos()`), whether, if there are too many tabs to fit the notebook should be made bigger or scrolling arrows added (see `Gtk.Notebook.set_scrollable()`, and whether there will be a popup menu allowing the users to switch pages (see `Gtk.Notebook.popup_enable()`, `Gtk.Notebook.popup_disable()`).

Example



```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class MyWindow(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="Simple Notebook Example")
        self.set_border_width(3)

        self.notebook = Gtk.Notebook()
        self.add(self.notebook)

        self.page1 = Gtk.Box()
        self.page1.set_border_width(10)
        self.page1.add(Gtk.Label('Default Page!'))
        self.notebook.append_page(self.page1, Gtk.Label('Plain Title'))

        self.page2 = Gtk.Box()
        self.page2.set_border_width(10)
        self.page2.add(Gtk.Label('A page with an image for a Title.'))
        self.notebook.append_page(
            self.page2,
            Gtk.Image.new_from_icon_name(
                "help-about",
                Gtk.IconSize.MENU
            )
        )

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

CHAPTER 6

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()`.

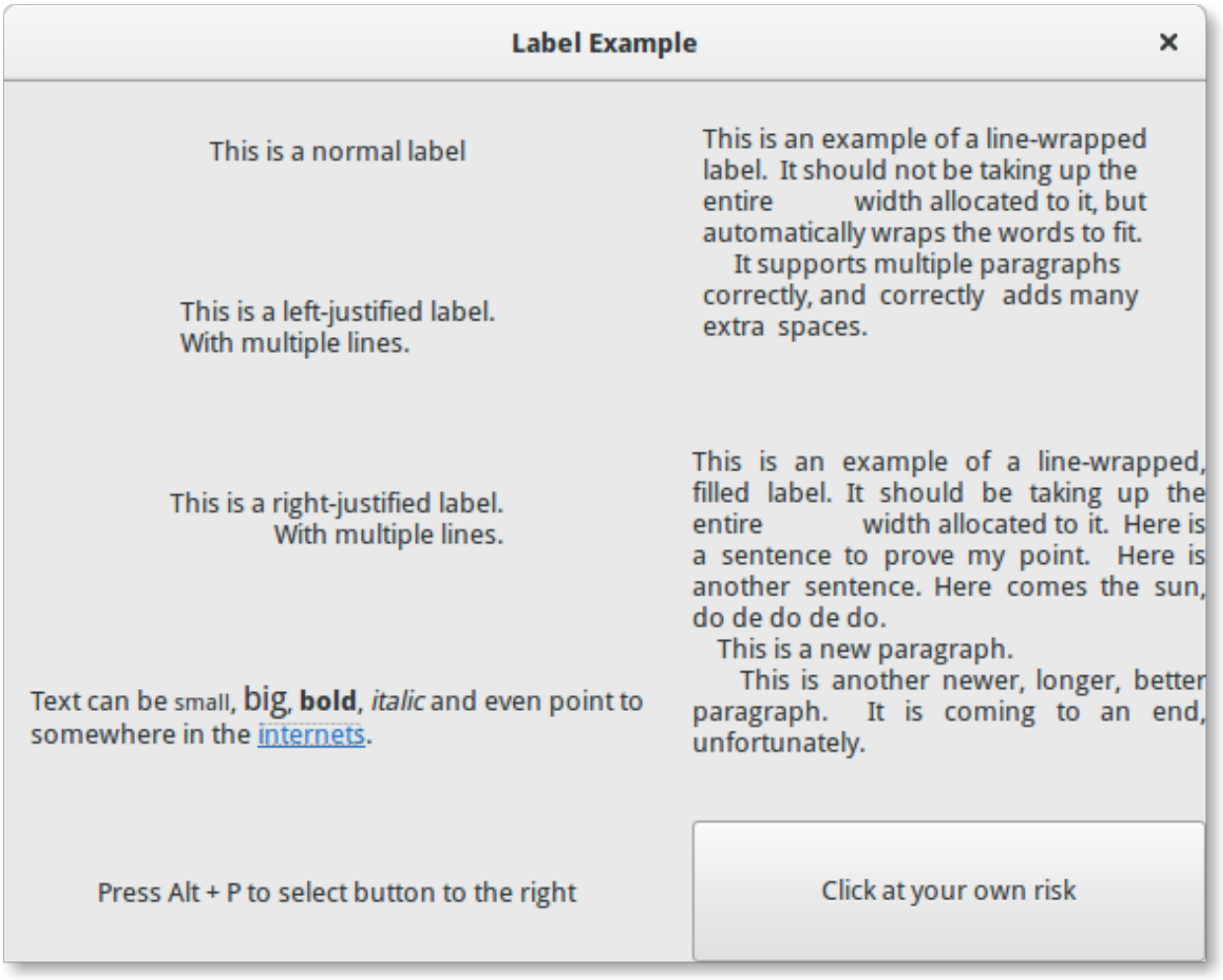
`Gtk.Label` support some simple formatting, for instance allowing you to make some text bold, colored, or larger. You can do this by providing a string to `Gtk.Label.set_markup()`, using the Pango Markup syntax[1]. For instance, `<b>bold text</b> and <s>strikethrough text</s>`. In addition, `Gtk.Label` 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 more")
```

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()`.

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

Example




```
import gi
gi.require_version('Gtk', '3.0')
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")
        label.set_mnemonic_widget(button)
        vbox_right.pack_start(button, True, True, 0)

        self.add(hbox)

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

```
79 Gtk.main()
```

CHAPTER 7

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()`.

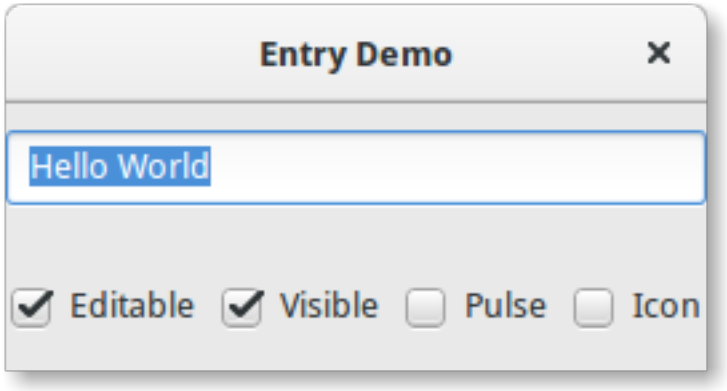
Occasionally you might want to make an Entry widget read-only. This can be done by passing `False` to the `Gtk.Entry.set_editable()` method.

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_icon_name()` 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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk, GObject

class EntryWindow(Gtk.Window):

    def __init__(self):
        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)
```

```
42
43      def on_editable_toggled(self, button):
44          value = button.get_active()
45          self.entry.set_editable(value)
46
47      def on_visible_toggled(self, button):
48          value = button.get_active()
49          self.entry.set_visibility(value)
50
51      def on_pulse_toggled(self, button):
52          if button.get_active():
53              self.entry.set_progress_pulse_step(0.2)
54              # Call self.do_pulse every 100 ms
55              self.timeout_id = GObject.timeout_add(100, self.do_pulse, None)
56          else:
57              # Don't call self.do_pulse anymore
58              GObject.source_remove(self.timeout_id)
59              self.timeout_id = None
60              self.entry.set_progress_pulse_step(0)
61
62      def do_pulse(self, user_data):
63          self.entry.progress_pulse()
64          return True
65
66      def on_icon_toggled(self, button):
67          if button.get_active():
68              icon_name = "system-search-symbolic"
69          else:
70              icon_name = None
71          self.entry.set_icon_from_icon_name(Gtk.EntryIconPosition.PRIMARY,
72              icon_name)
73
74  win = EntryWindow()
75  win.connect("delete-event", Gtk.main_quit)
76  win.show_all()
77  Gtk.main()
```

CHAPTER 8

Button Widgets

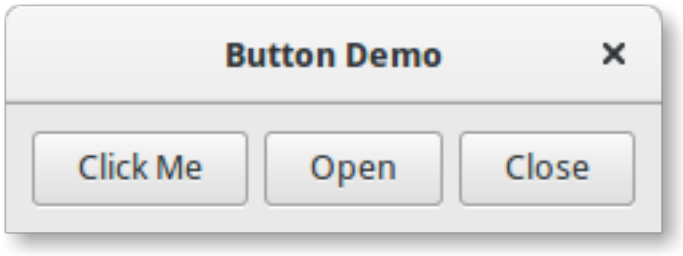
Button

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class ButtonWindow(Gtk.Window):

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

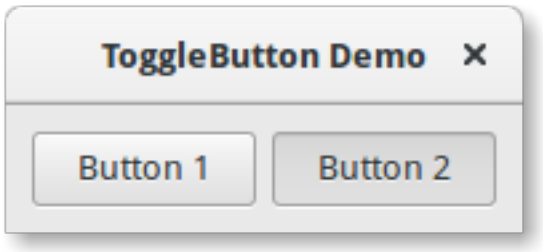
```
10
11          hbox = Gtk.Box(spacing=6)
12          self.add(hbox)
13
14          button = Gtk.Button.new_with_label("Click Me")
15          button.connect("clicked", self.on_click_me_clicked)
16          hbox.pack_start(button, True, True, 0)
17
18          button = Gtk.Button.new_with_mnemonic("_Open")
19          button.connect("clicked", self.on_open_clicked)
20          hbox.pack_start(button, True, True, 0)
21
22          button = Gtk.Button.new_with_mnemonic("_Close")
23          button.connect("clicked", self.on_close_clicked)
24          hbox.pack_start(button, True, True, 0)
25
26      def on_click_me_clicked(self, button):
27          print("\"Click me\" button was clicked")
28
29      def on_open_clicked(self, button):
30          print("\"Open\" button was clicked")
31
32      def on_close_clicked(self, button):
33          print("Closing application")
34          Gtk.main_quit()
35
36  win = ButtonWindow()
37  win.connect("delete-event", Gtk.main_quit)
38  win.show_all()
39  Gtk.main()
```

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.

Example



```
import gi
gi.require_version('Gtk', '3.0')
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()
```

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.

RadioButton

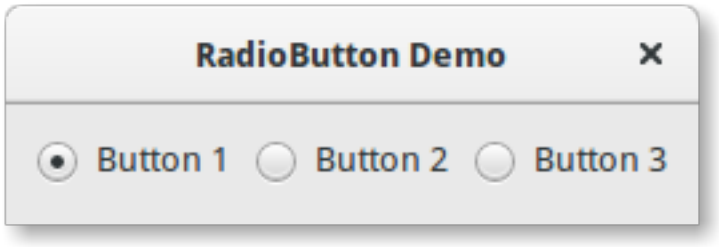
Like checkboxes, radio buttons also inherit from `Gtk.ToggleButton`, but these work in groups, and only one `Gtk.RadioButton` in a group can be selected at any one time. Therefore, a `Gtk.RadioButton` is one way of giving the user a choice from many options.

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()`.

Example



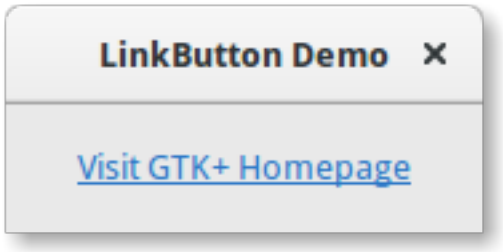

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

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()
```

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()`.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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()
```

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`.

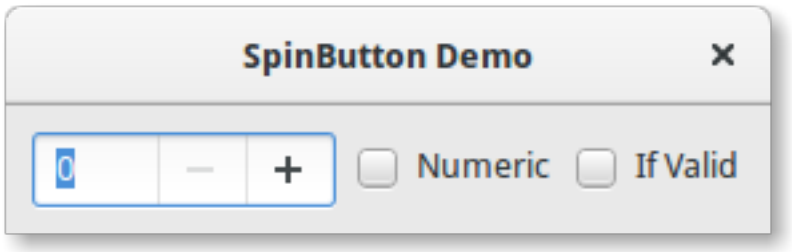
To change the value that `Gtk.SpinButton` is showing, use `Gtk.SpinButton.set_value()`. The value entered can either be an integer or float, depending on your requirements, use `Gtk.SpinButton.get_value()` or `Gtk.SpinButton.get_value_as_int()`, respectively.

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()`.

Example



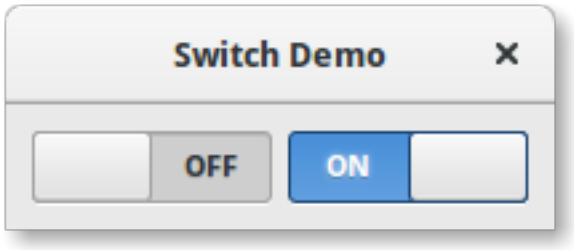

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

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()
```

Switch

A `Gtk.Switch` is a widget that has two states: on or off. The user can control which state should be active by clicking the empty area, or by dragging the handle.

You shouldn't use the "activate" signal on the Gtk.Switch which is an action signal and emitting it causes the switch to animate. Applications should never connect to this signal, but use the "notify::active" signal, see the example here below.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class SwitcherWindow(Gtk.Window):

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

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

        switch = Gtk.Switch()
        switch.connect("notify::active", self.on_switch_activated)
        switch.set_active(False)
        hbox.pack_start(switch, True, True, 0)

        switch = Gtk.Switch()
        switch.connect("notify::active", self.on_switch_activated)
        switch.set_active(True)
        hbox.pack_start(switch, True, True, 0)

    def on_switch_activated(self, switch, gparam):
        if switch.get_active():
            state = "on"
        else:
            state = "off"
        print("Switch was turned", state)

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

```

CHAPTER 9

ProgressBar

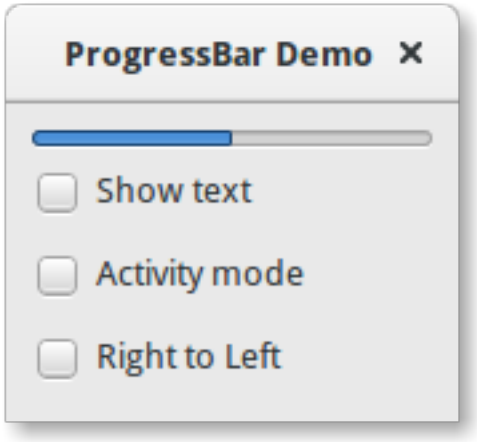
The `Gtk.ProgressBar` is typically used to display the progress of a long running operation. It provides a visual clue that processing is underway. The `Gtk.ProgressBar` can be used in two different modes: *percentage mode* and *activity mode*.

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()`.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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)

        self.timeout_id = GObject.timeout_add(50, self.on_timeout, None)
        self.activity_mode = False

    def on_show_text_toggled(self, button):
        show_text = button.get_active()
        if show_text:
            text = "some text"
        else:
            text = None
        self.progressbar.set_text(text)
        self.progressbar.set_show_text(show_text)
```

```
40
41      def on_activity_mode_toggled(self, button):
42          self.activity_mode = button.get_active()
43          if self.activity_mode:
44              self.progressbar.pulse()
45          else:
46              self.progressbar.set_fraction(0.0)
47
48      def on_right_to_left_toggled(self, button):
49          value = button.get_active()
50          self.progressbar.set_inverted(value)
51
52      def on_timeout(self, user_data):
53          """
54          Update value on the progress bar
55          """
56          if self.activity_mode:
57              self.progressbar.pulse()
58          else:
59              new_value = self.progressbar.get_fraction() + 0.01
60
61              if new_value > 1:
62                  new_value = 0
63
64              self.progressbar.set_fraction(new_value)
65
66          # As this is a timeout function, return True so that it
67          # continues to get called
68          return True
69
70  win = ProgressBarWindow()
71  win.connect("delete-event", Gtk.main_quit)
72  win.show_all()
73  Gtk.main()
```

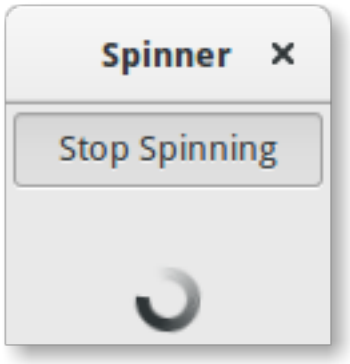
CHAPTER 10

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()`.

Example




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class SpinnerAnimation(Gtk.Window):

    def __init__(self):

        Gtk.Window.__init__(self, title="Spinner")
        self.set_border_width(3)
        self.connect("delete-event", Gtk.main_quit)

```

```
        self.button = Gtk.ToggleButton("Start Spinning")
        self.button.connect("toggled", self.on_button_toggled)
        self.button.set_active(False)

        self.spinner = Gtk.Spinner()

        self.table = Gtk.Table(3, 2, True)
        self.table.attach(self.button, 0, 2, 0, 1)
        self.table.attach(self.spinner, 0, 2, 2, 3)

        self.add(self.table)
        self.show_all()

    def on_button_toggled(self, button):

        if button.get_active():
            self.spinner.start()
            self.button.set_label("Stop Spinning")

        else:
            self.spinner.stop()
            self.button.set_label("Start Spinning")


myspinner = SpinnerAnimation()

Gtk.main()

```

CHAPTER 11

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 it correctly due to the number of methods which are required.

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 last two columns
store[treeiter][1:] = ["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.

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)
```

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.

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.

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)
```

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)
```

Filtering

Unlike sorting, filtering is not handled by the two models we previously saw, but by the `Gtk.TreeModelFilter` class. This class, like `Gtk.TreeStore` and `Gtk.ListStore`, is a `Gtk.TreeModel`. It acts as a layer between the "real" model (a `Gtk.TreeStore` or a `Gtk.ListStore`), hiding some elements to the view. In practice, it supplies the `Gtk.TreeView` with a subset of the underlying model. Instances of `Gtk.TreeModelFilter` can be stacked one onto another, to use multiple filters on the same model (in the same way you'd use "AND" clauses in a SQL request). They can also be chained with `Gtk.TreeModelSort` instances.

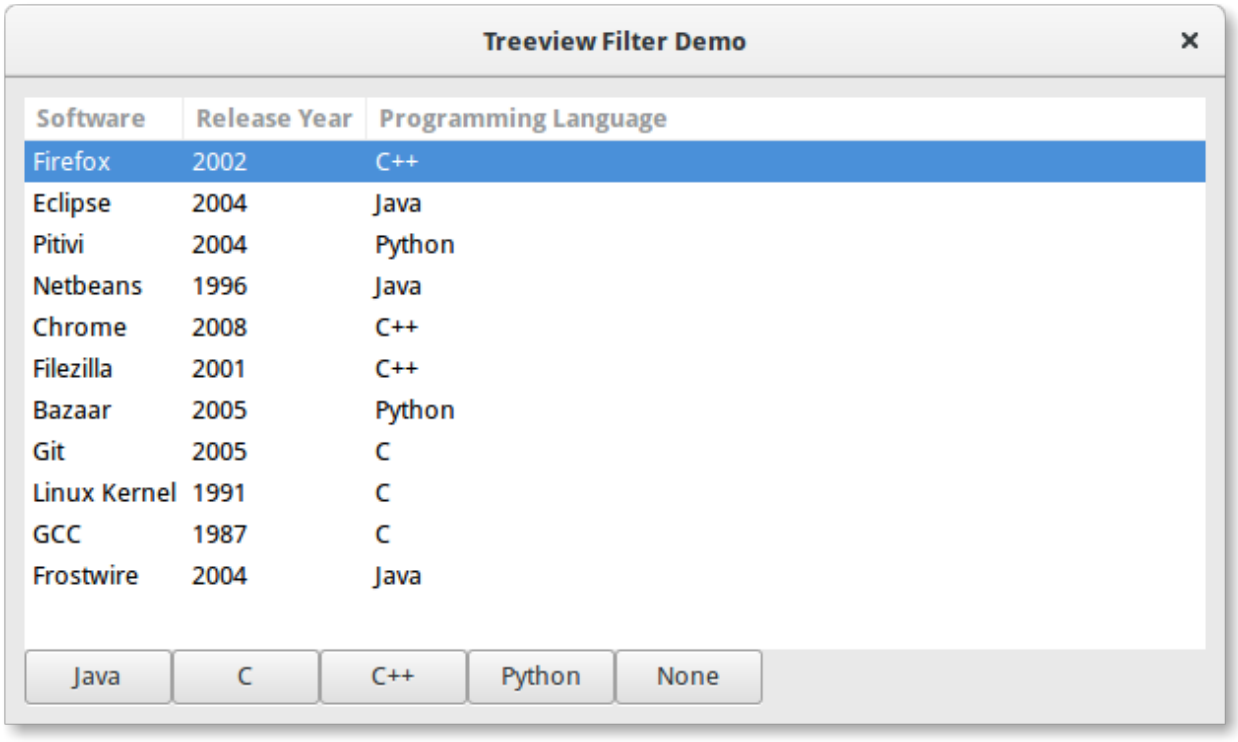
You can create a new instance of a `Gtk.TreeModelFilter` and give it a model to filter, but the easiest way is to spawn it directly from the filtered model, using the `Gtk.TreeModel.filter_new()` method.

```
filter = model.filter_new()
```

In the same way the sorting function works, the `Gtk.TreeModelFilter` needs a "visibility" function, which, given a row from the underlying model, will return a boolean indicating if this row should be filtered out or not. It's set by `Gtk.TreeSortable.set_visible_func()`:

```
filter.set_visible_func(filter_func, data=None)
```

Let's look at a full example which uses the whole `Gtk.ListStore` - `Gtk.TreeModelFilter` - `Gtk.TreeModelFilter` - `Gtk.TreeView` stack.




```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

#list of tuples for each software, containing the software name, initial release, and main programming languages used
software_list = [("Firefox", 2002,  "C++"),
                 ("Eclipse", 2004, "Java" ),
                 ("Pitivi", 2004, "Python"),
                 ("Netbeans", 1996, "Java"),
                 ("Chrome", 2008, "C++"),
                 ("Filezilla", 2001, "C++"),
                 ("Bazaar", 2005, "Python"),
                 ("Git", 2005, "C"),
                 ("Linux Kernel", 1991, "C"),
                 ("GCC", 1987, "C"),
                 ("Frostwire", 2004, "Java")]

class TreeViewFilterWindow(Gtk.Window):

```

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

        #Setting up the self.grid in which the elements are to be positionned
        self.grid = Gtk.Grid()
        self.grid.set_column_homogeneous(True)
        self.grid.set_row_homogeneous(True)
        self.add(self.grid)

        #Creating the ListStore model
        self.software_liststore = Gtk.ListStore(str, int, str)
        for software_ref in software_list:
            self.software_liststore.append(list(software_ref))
        self.current_filter_language = None

        #Creating the filter, feeding it with the liststore model
        self.language_filter = self.software_liststore.filter_new()
        #setting the filter function, note that we're not using the
        self.language_filter.set_visible_func(self.language_filter_func)

        #creating the treeview, making it use the filter as a model, and adding the␣
↪columns
        self.treeview = Gtk.TreeView.new_with_model(self.language_filter)
        for i, column_title in enumerate(["Software", "Release Year", "Programming␣
↪Language"]):
            renderer = Gtk.CellRendererText()
            column = Gtk.TreeViewColumn(column_title, renderer, text=i)
            self.treeview.append_column(column)

        #creating buttons to filter by programming language, and setting up their␣
↪events
        self.buttons = list()
        for prog_language in ["Java", "C", "C++", "Python", "None"]:
            button = Gtk.Button(prog_language)
            self.buttons.append(button)
            button.connect("clicked", self.on_selection_button_clicked)

        #setting up the layout, putting the treeview in a scrollwindow, and the␣
↪buttons in a row
        self.scrollable_treelist = Gtk.ScrolledWindow()
        self.scrollable_treelist.set_vexpand(True)
        self.grid.attach(self.scrollable_treelist, 0, 0, 8, 10)
        self.grid.attach_next_to(self.buttons[0], self.scrollable_treelist, Gtk.
↪PositionType.BOTTOM, 1, 1)
        for i, button in enumerate(self.buttons[1:]):
            self.grid.attach_next_to(button, self.buttons[i], Gtk.PositionType.RIGHT,␣
↪1, 1)
        self.scrollable_treelist.add(self.treeview)

        self.show_all()

    def language_filter_func(self, model, iter, data):
        """Tests if the language in the row is the one in the filter"""
        if self.current_filter_language is None or self.current_filter_language ==
↪"None":
            return True
        else:
```

```
            return model[iter][2] == self.current_filter_language

    def on_selection_button_clicked(self, widget):
        """Called on any of the button clicks"""
        #we set the current language filter to the button's label
        self.current_filter_language = widget.get_label()
        print("%s language selected!" % self.current_filter_language)
        #we update the filter, which updates in turn the view
        self.language_filter.refilter()


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

CHAPTER 12

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

CellRendererText

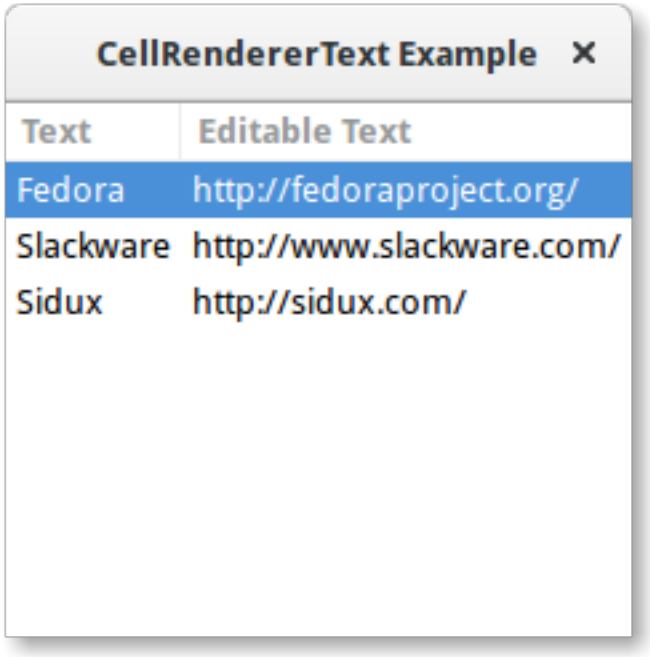
A Gtk.CellRendererText renders a given text in its cell, using the font, color and style information provided by its properties. The text will be ellipsized if it is too long and the "ellipsize" property allows it.

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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)
```

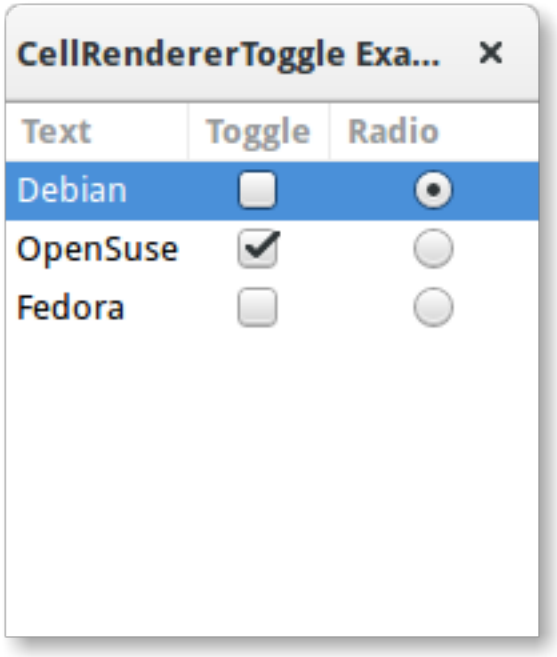
```
    def text_edited(self, widget, path, text):
        self.liststore[path][1] = text

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

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.

Example



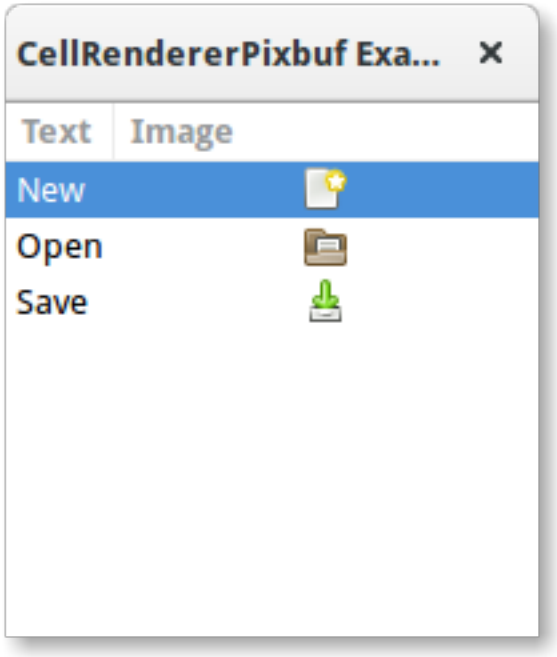

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class CellRendererToggleWindow(Gtk.Window):

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

        self.set_default_size(200, 200)
```

```
11
12          self.liststore = Gtk.ListStore(str, bool, bool)
13          self.liststore.append(["Debian", False, True])
14          self.liststore.append(["OpenSuse", True, False])
15          self.liststore.append(["Fedora", False, False])
16
17          treeview = Gtk.TreeView(model=self.liststore)
18
19          renderer_text = Gtk.CellRendererText()
20          column_text = Gtk.TreeViewColumn("Text", renderer_text, text=0)
21          treeview.append_column(column_text)
22
23          renderer_toggle = Gtk.CellRendererToggle()
24          renderer_toggle.connect("toggled", self.on_cell_toggled)
25
26          column_toggle = Gtk.TreeViewColumn("Toggle", renderer_toggle, active=1)
27          treeview.append_column(column_toggle)
28
29          renderer_radio = Gtk.CellRendererToggle()
30          renderer_radio.set_radio(True)
31          renderer_radio.connect("toggled", self.on_cell_radio_toggled)
32
33          column_radio = Gtk.TreeViewColumn("Radio", renderer_radio, active=2)
34          treeview.append_column(column_radio)
35
36          self.add(treeview)
37
38      def on_cell_toggled(self, widget, path):
39          self.liststore[path][1] = not self.liststore[path][1]
40
41      def on_cell_radio_toggled(self, widget, path):
42          selected_path = Gtk.TreePath(path)
43          for row in self.liststore:
44              row[2] = (row.path == selected_path)
45
46  win = CellRendererToggleWindow()
47  win.connect("delete-event", Gtk.main_quit)
48  win.show_all()
49  Gtk.main()
```

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 named icon (set via the “icon-name” property).

Example




```
import gi
gi.require_version('Gtk', '3.0')
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", "document-new"])
        self.liststore.append(["Open", "document-open"])
        self.liststore.append(["Save", "document-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, icon_name=1)
        treeview.append_column(column_pixbuf)

        self.add(treeview)

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

```
33 Gtk.main()
```

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.

Example


```
1  import gi
2  gi.require_version('Gtk', '3.0')
3  from gi.repository import Gtk
4
5  class CellRendererComboWindow(Gtk.Window):
6
7      def __init__(self):
8          Gtk.Window.__init__(self, title="CellRendererCombo Example")
9
10         self.set_default_size(200, 200)
11
12         liststore_manufacturers = Gtk.ListStore(str)
13         manufacturers = ["Sony", "LG",
```

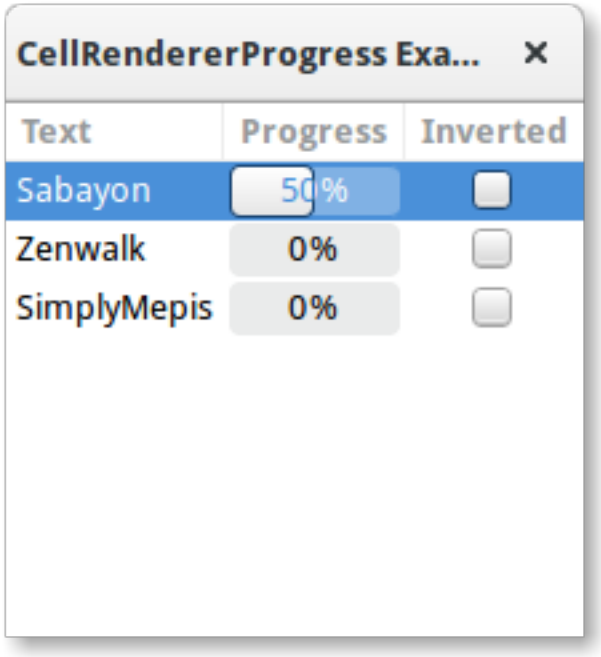
```
            "Panasonic", "Toshiba", "Nokia", "Samsung"]
        for item in manufacturers:
            liststore_manufacturers.append([item])

        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)
        treeview.append_column(column_combo)

        self.add(treeview)

    def on_combo_changed(self, widget, path, text):
        self.liststore_hardware[path][1] = text

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

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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)
```

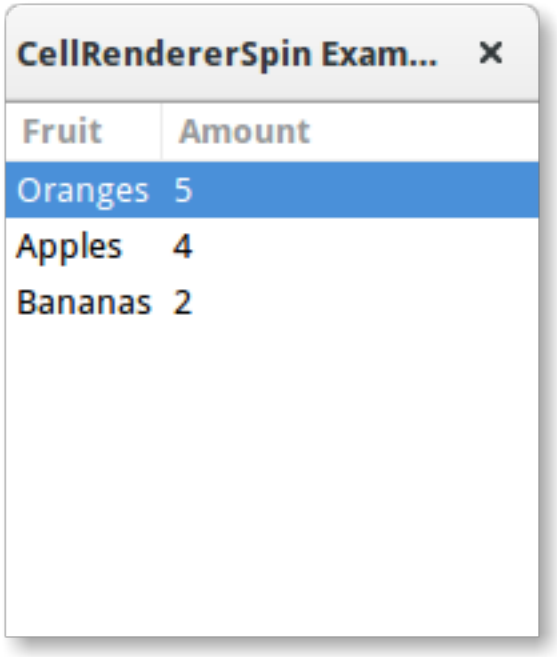
```
33
34          self.add(treeview)
35
36          self.timeout_id = GObject.timeout_add(100, self.on_timeout, None)
37
38      def on_inverted_toggled(self, widget, path):
39          self.liststore[path][2] = not self.liststore[path][2]
40
41      def on_timeout(self, user_data):
42          new_value = self.liststore[self.current_iter][1] + 1
43          if new_value > 100:
44              self.current_iter = self.liststore.iter_next(self.current_iter)
45              if self.current_iter == None:
46                  self.reset_model()
47              new_value = self.liststore[self.current_iter][1] + 1
48
49          self.liststore[self.current_iter][1] = new_value
50          return True
51
52      def reset_model(self):
53          for row in self.liststore:
54              row[1] = 0
55          self.current_iter = self.liststore.get_iter_first()
56
57  win = CellRendererProgressWindow()
58  win.connect("delete-event", Gtk.main_quit)
59  win.show_all()
60  Gtk.main()
```

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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])

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

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

        renderer_spin = Gtk.CellRendererSpin()
        renderer_spin.connect("edited", self.on_amount_edited)
        renderer_spin.set_property("editable", True)

        adjustment = Gtk.Adjustment(0, 0, 100, 1, 10, 0)
        renderer_spin.set_property("adjustment", adjustment)

        column_spin = Gtk.TreeViewColumn("Amount", renderer_spin, text=1)
        treeview.append_column(column_spin)

```

```
33          self.add(treeview)
34
35      def on_amount_edited(self, widget, path, value):
36          self.liststore[path][1] = int(value)
37
38  win = CellRendererSpinWindow()
39  win.connect("delete-event", Gtk.main_quit)
40  win.show_all()
41  Gtk.main()
```

CHAPTER 13

ComboBox

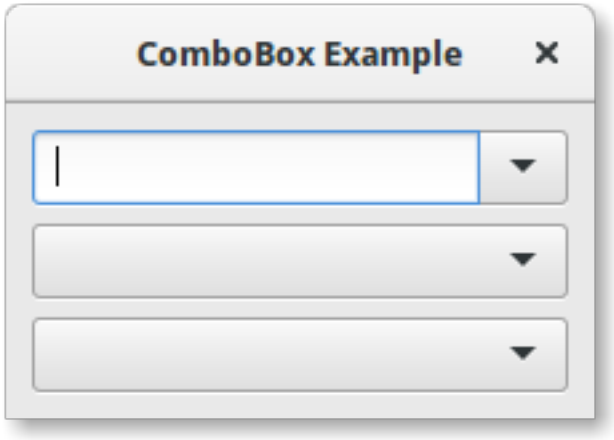
A `Gtk.ComboBox` allows for the selection of an item from a dropdown menu. They are preferable to having many radio buttons on screen as they take up less room. If appropriate, it can show extra information about each item, such as text, a picture, a checkbox, or a progress bar.

`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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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"])
        name_store.append([11, "Billy Bob Junior"])
        name_store.append([12, "Sue Bob"])
        name_store.append([2, "Joey Jojo"])
        name_store.append([3, "Rob McRoberts"])
        name_store.append([31, "Xavier McRoberts"])

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

        name_combo = Gtk.ComboBox.new_with_model_and_entry(name_store)
        name_combo.connect("changed", self.on_name_combo_changed)
        name_combo.set_entry_text_column(1)
        vbox.pack_start(name_combo, False, False, 0)

        country_store = Gtk.ListStore(str)
        countries = ["Austria", "Brazil", "Belgium", "France", "Germany",
            "Switzerland", "United Kingdom", "United States of America",
            "Uruguay"]
        for country in countries:
            country_store.append([country])

        country_combo = Gtk.ComboBox.new_with_model(country_store)
        country_combo.connect("changed", self.on_country_combo_changed)
        renderer_text = Gtk.CellRendererText()
        country_combo.pack_start(renderer_text, True)
        country_combo.add_attribute(renderer_text, "text", 0)
        vbox.pack_start(country_combo, False, False, True)
```

```
40
41          currencies = ["Euro", "US Dollars", "British Pound", "Japanese Yen",
42              "Russian Ruble", "Mexican peso", "Swiss franc"]
43          currency_combo = Gtk.ComboBoxText()
44          currency_combo.set_entry_text_column(0)
45          currency_combo.connect("changed", self.on_currency_combo_changed)
46          for currency in currencies:
47              currency_combo.append_text(currency)
48
49          vbox.pack_start(currency_combo, False, False, 0)
50
51          self.add(vbox)
52
53      def on_name_combo_changed(self, combo):
54          tree_iter = combo.get_active_iter()
55          if tree_iter != None:
56              model = combo.get_model()
57              row_id, name = model[tree_iter][:2]
58              print("Selected: ID=%d, name=%s" % (row_id, name))
59          else:
60              entry = combo.get_child()
61              print("Entered: %s" % entry.get_text())
62
63      def on_country_combo_changed(self, combo):
64          tree_iter = combo.get_active_iter()
65          if tree_iter != None:
66              model = combo.get_model()
67              country = model[tree_iter][0]
68              print("Selected: country=%s" % country)
69
70      def on_currency_combo_changed(self, combo):
71          text = combo.get_active_text()
72          if text != None:
73              print("Selected: currency=%s" % text)
74
75  win = ComboBoxWindow()
76  win.connect("delete-event", Gtk.main_quit)
77  win.show_all()
78  Gtk.main()
```

CHAPTER 14

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.

Example


```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk
from gi.repository.GdkPixbuf import Pixbuf

icons = ["edit-cut", "edit-paste", "edit-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()
```

```
27 win.connect("delete-event", Gtk.main_quit)
28 win.show_all()
29 Gtk.main()
```

CHAPTER 15

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.

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()`.

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.

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()`:

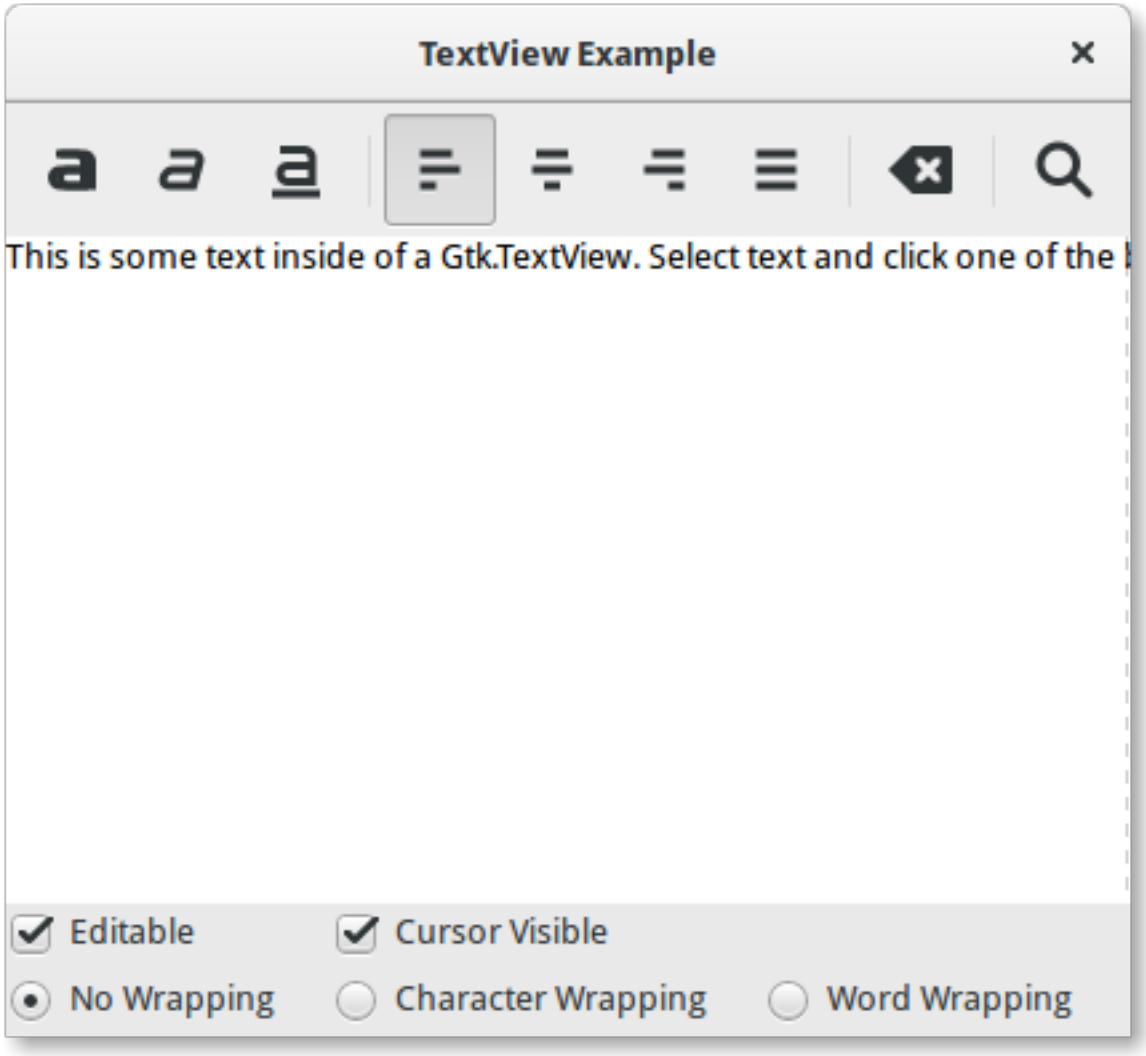
```
tag = textbuffer.create_tag("orange_bg", background="orange")
textbuffer.apply_tag(tag, start_iter, end_iter)
```

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()`.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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()
```

```
14
15          label = Gtk.Label("Insert text you want to search for:")
16          box.add(label)
17
18          self.entry = Gtk.Entry()
19          box.add(self.entry)
20
21          self.show_all()
22
23  class TextViewWindow(Gtk.Window):
24
25      def __init__(self):
26          Gtk.Window.__init__(self, title="TextView Example")
27
28          self.set_default_size(-1, 350)
29
30          self.grid = Gtk.Grid()
31          self.add(self.grid)
32
33          self.create_textview()
34          self.create_toolbar()
35          self.create_buttons()
36
37      def create_toolbar(self):
38          toolbar = Gtk.Toolbar()
39          self.grid.attach(toolbar, 0, 0, 3, 1)
40
41          button_bold = Gtk.ToolButton()
42          button_bold.set_icon_name("format-text-bold-symbolic")
43          toolbar.insert(button_bold, 0)
44
45          button_italic = Gtk.ToolButton()
46          button_italic.set_icon_name("format-text-italic-symbolic")
47          toolbar.insert(button_italic, 1)
48
49          button_underline = Gtk.ToolButton()
50          button_underline.set_icon_name("format-text-underline-symbolic")
51          toolbar.insert(button_underline, 2)
52
53          button_bold.connect("clicked", self.on_button_clicked, self.tag_bold)
54          button_italic.connect("clicked", self.on_button_clicked,
55              self.tag_italic)
56          button_underline.connect("clicked", self.on_button_clicked,
57              self.tag_underline)
58
59          toolbar.insert(Gtk.SeparatorToolItem(), 3)
60
61          radio_justifyleft = Gtk.RadioToolButton()
62          radio_justifyleft.set_icon_name("format-justify-left-symbolic")
63          toolbar.insert(radio_justifyleft, 4)
64
65          radio_justifycenter = Gtk.RadioToolButton.new_from_widget(radio_justifyleft)
66          radio_justifycenter.set_icon_name("format-justify-center-symbolic")
67          toolbar.insert(radio_justifycenter, 5)
68
69          radio_justifyright = Gtk.RadioToolButton.new_from_widget(radio_justifyleft)
70          radio_justifyright.set_icon_name("format-justify-right-symbolic")
71          toolbar.insert(radio_justifyright, 6)
```

```
72
73          radio_justifyfill = Gtk.RadioToolButton.new_from_widget(radio_justifyleft)
74          radio_justifyfill.set_icon_name("format-justify-fill-symbolic")
75          toolbar.insert(radio_justifyfill, 7)
76
77          radio_justifyleft.connect("toggled", self.on_justify_toggled,
78              Gtk.Justification.LEFT)
79          radio_justifycenter.connect("toggled", self.on_justify_toggled,
80              Gtk.Justification.CENTER)
81          radio_justifyright.connect("toggled", self.on_justify_toggled,
82              Gtk.Justification.RIGHT)
83          radio_justifyfill.connect("toggled", self.on_justify_toggled,
84              Gtk.Justification.FILL)
85
86          toolbar.insert(Gtk.SeparatorToolItem(), 8)
87
88          button_clear = Gtk.ToolButton()
89          button_clear.set_icon_name("edit-clear-symbolic")
90          button_clear.connect("clicked", self.on_clear_clicked)
91          toolbar.insert(button_clear, 9)
92
93          toolbar.insert(Gtk.SeparatorToolItem(), 10)
94
95          button_search = Gtk.ToolButton()
96          button_search.set_icon_name("system-search-symbolic")
97          button_search.connect("clicked", self.on_search_clicked)
98          toolbar.insert(button_search, 11)
99
100     def create_textview(self):
101         scrolledwindow = Gtk.ScrolledWindow()
102         scrolledwindow.set_hexpand(True)
103         scrolledwindow.set_vexpand(True)
104         self.grid.attach(scrolledwindow, 0, 1, 3, 1)
105
106         self.textview = Gtk.TextView()
107         self.textbuffer = self.textview.get_buffer()
108         self.textbuffer.set_text("This is some text inside of a Gtk.TextView. "
109             + "Select text and click one of the buttons 'bold', 'italic', "
110             + "or 'underline' to modify the text accordingly.")
111         scrolledwindow.add(self.textview)
112
113         self.tag_bold = self.textbuffer.create_tag("bold",
114             weight=Pango.Weight.BOLD)
115         self.tag_italic = self.textbuffer.create_tag("italic",
116             style=Pango.Style.ITALIC)
117         self.tag_underline = self.textbuffer.create_tag("underline",
118             underline=Pango.Underline.SINGLE)
119         self.tag_found = self.textbuffer.create_tag("found",
120             background="yellow")
121
122     def create_buttons(self):
123         check_editable = Gtk.CheckButton("Editable")
124         check_editable.set_active(True)
125         check_editable.connect("toggled", self.on_editable_toggled)
126         self.grid.attach(check_editable, 0, 2, 1, 1)
127
128         check_cursor = Gtk.CheckButton("Cursor Visible")
129         check_cursor.set_active(True)
```

```
        check_editable.connect("toggled", self.on_cursor_toggled)
        self.grid.attach_next_to(check_cursor, check_editable,
            Gtk.PositionType.RIGHT, 1, 1)

        radio_wrapnone = Gtk.RadioButton.new_with_label_from_widget(None,
            "No Wrapping")
        self.grid.attach(radio_wrapnone, 0, 3, 1, 1)

        radio_wrapchar = Gtk.RadioButton.new_with_label_from_widget(
            radio_wrapnone, "Character Wrapping")
        self.grid.attach_next_to(radio_wrapchar, radio_wrapnone,
            Gtk.PositionType.RIGHT, 1, 1)

        radio_wrapword = Gtk.RadioButton.new_with_label_from_widget(
            radio_wrapnone, "Word Wrapping")
        self.grid.attach_next_to(radio_wrapword, radio_wrapchar,
            Gtk.PositionType.RIGHT, 1, 1)

        radio_wrapnone.connect("toggled", self.on_wrap_toggled,
            Gtk.WrapMode.NONE)
        radio_wrapchar.connect("toggled", self.on_wrap_toggled,
            Gtk.WrapMode.CHAR)
        radio_wrapword.connect("toggled", self.on_wrap_toggled,
            Gtk.WrapMode.WORD)

    def on_button_clicked(self, widget, tag):
        bounds = self.textbuffer.get_selection_bounds()
        if len(bounds) != 0:
            start, end = bounds
            self.textbuffer.apply_tag(tag, start, end)

    def on_clear_clicked(self, widget):
        start = self.textbuffer.get_start_iter()
        end = self.textbuffer.get_end_iter()
        self.textbuffer.remove_all_tags(start, end)

    def on_editable_toggled(self, widget):
        self.textview.set_editable(widget.get_active())

    def on_cursor_toggled(self, widget):
        self.textview.set_cursor_visible(widget.get_active())

    def on_wrap_toggled(self, widget, mode):
        self.textview.set_wrap_mode(mode)

    def on_justify_toggled(self, widget, justification):
        self.textview.set_justification(justification)

    def on_search_clicked(self, widget):
        dialog = SearchDialog(self)
        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            cursor_mark = self.textbuffer.get_insert()
            start = self.textbuffer.get_iter_at_mark(cursor_mark)
            if start.get_offset() == self.textbuffer.get_char_count():
                start = self.textbuffer.get_start_iter()

            self.search_and_mark(dialog.entry.get_text(), start)
```

```
188
189         dialog.destroy()
190
191     def search_and_mark(self, text, start):
192         end = self.textbuffer.get_end_iter()
193         match = start.forward_search(text, 0, end)
194
195         if match != None:
196             match_start, match_end = match
197             self.textbuffer.apply_tag(self.tag_found, match_start, match_end)
198             self.search_and_mark(text, match_end)
199
200 win = TextViewWindow()
201 win.connect("delete-event", Gtk.main_quit)
202 win.show_all()
203 Gtk.main()
```

CHAPTER 16

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.

Custom Dialogs

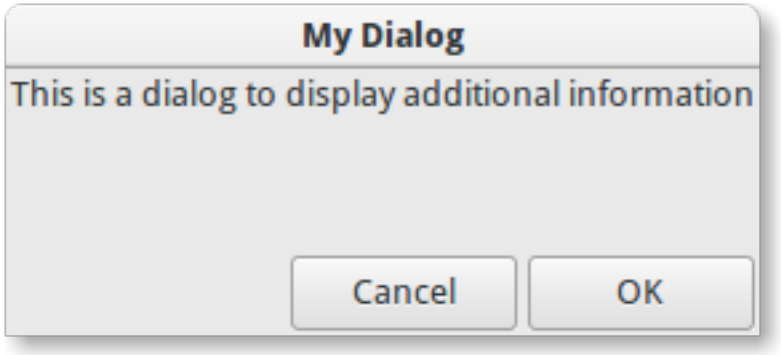
To pack widgets into a custom dialog, you should pack them into the `Gtk.Box`, available via `Gtk.Dialog.get_content_area()`. To just add a `Gtk.Button` to the bottom of the dialog, you could use the `Gtk.Dialog.add_button()` method.

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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()

```

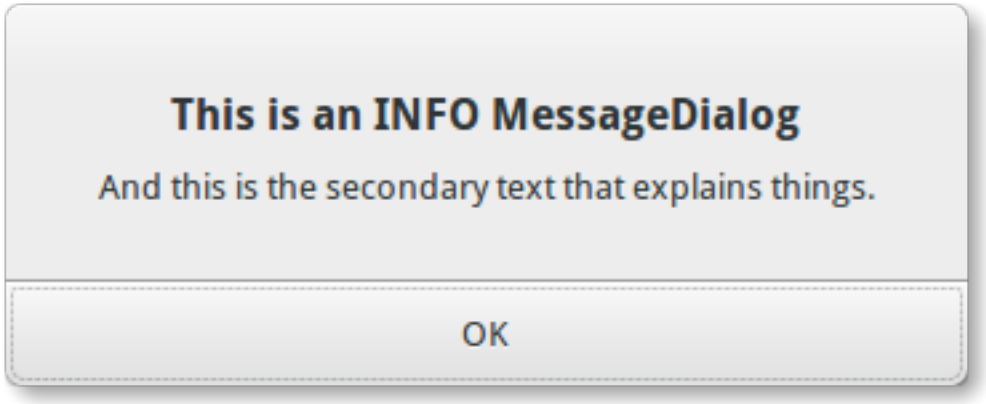
```
win = DialogWindow()
win.connect("delete-event", Gtk.main_quit)
win.show_all()
Gtk.main()
```

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()`.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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()
```

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()).

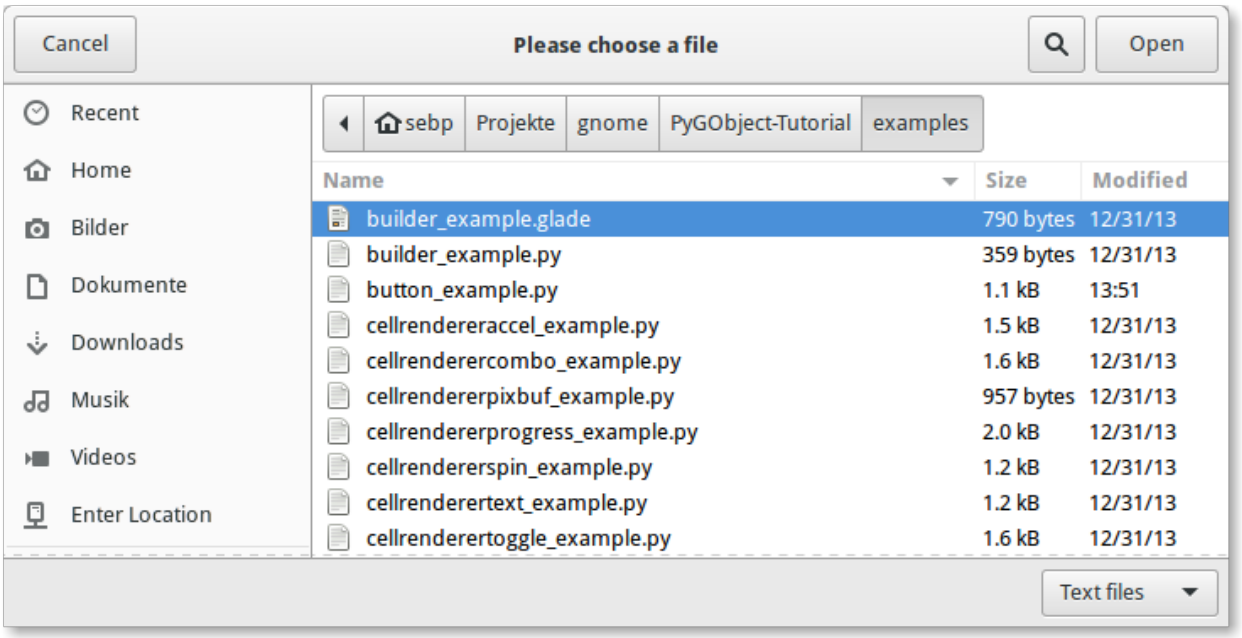
By default, Gtk.FileChooser only allows a single file to be selected at a time. To enable multiple files to be selected, use Gtk.FileChooser.set_select_multiple(). Retrieving a list of selected files is possible with either Gtk.FileChooser.get_filenames() or Gtk.FileChooser.get_uris().

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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,
             Gtk.STOCK_OPEN, Gtk.ResponseType.OK))

        self.add_filters(dialog)

        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            print("Open clicked")
            print("File selected: " + dialog.get_filename())
        elif response == Gtk.ResponseType.CANCEL:
```

```
            print("Cancel clicked")

        dialog.destroy()

    def add_filters(self, dialog):
        filter_text = Gtk.FileFilter()
        filter_text.set_name("Text files")
        filter_text.add_mime_type("text/plain")
        dialog.add_filter(filter_text)

        filter_py = Gtk.FileFilter()
        filter_py.set_name("Python files")
        filter_py.add_mime_type("text/x-python")
        dialog.add_filter(filter_py)

        filter_any = Gtk.FileFilter()
        filter_any.set_name("Any files")
        filter_any.add_pattern("*")
        dialog.add_filter(filter_any)

    def on_folder_clicked(self, widget):
        dialog = Gtk.FileChooserDialog("Please choose a folder", self,
            Gtk.FileChooserAction.SELECT_FOLDER,
            (Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,
             "Select", Gtk.ResponseType.OK))
        dialog.set_default_size(800, 400)

        response = dialog.run()
        if response == Gtk.ResponseType.OK:
            print("Select clicked")
            print("Folder selected: " + dialog.get_filename())
        elif response == Gtk.ResponseType.CANCEL:
            print("Cancel clicked")

        dialog.destroy()

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

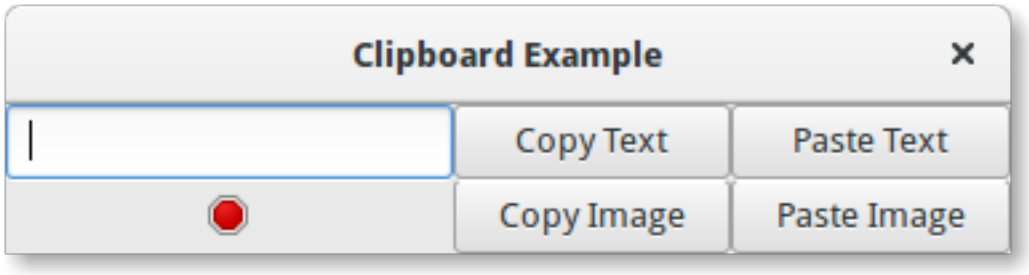
CHAPTER 17

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.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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)
```

```
11
12          self.clipboard = Gtk.Clipboard.get(Gdk.SELECTION_CLIPBOARD)
13          self.entry = Gtk.Entry()
14          self.image = Gtk.Image.new_from_icon_name("process-stop", Gtk.IconSize.MENU)
15
16          button_copy_text = Gtk.Button("Copy Text")
17          button_paste_text = Gtk.Button("Paste Text")
18          button_copy_image = Gtk.Button("Copy Image")
19          button_paste_image = Gtk.Button("Paste Image")
20
21          table.attach(self.entry, 0, 1, 0, 1)
22          table.attach(self.image, 0, 1, 1, 2)
23          table.attach(button_copy_text, 1, 2, 0, 1)
24          table.attach(button_paste_text, 2, 3, 0, 1)
25          table.attach(button_copy_image, 1, 2, 1, 2)
26          table.attach(button_paste_image, 2, 3, 1, 2)
27
28          button_copy_text.connect("clicked", self.copy_text)
29          button_paste_text.connect("clicked", self.paste_text)
30          button_copy_image.connect("clicked", self.copy_image)
31          button_paste_image.connect("clicked", self.paste_image)
32
33          self.add(table)
34
35      def copy_text(self, widget):
36          self.clipboard.set_text(self.entry.get_text(), -1)
37
38      def paste_text(self, widget):
39          text = self.clipboard.wait_for_text()
40          if text != None:
41              self.entry.set_text(text)
42          else:
43              print("No text on the clipboard.")
44
45      def copy_image(self, widget):
46          if self.image.get_storage_type() == Gtk.ImageType.PIXBUF:
47              self.clipboard.set_image(self.image.get_pixbuf())
48          else:
49              print("No image has been pasted yet.")
50
51      def paste_image(self, widget):
52          image = self.clipboard.wait_for_image()
53          if image != None:
54              self.image.set_from_pixbuf(image)
55
56
57  win = ClipboardWindow()
58  win.connect("delete-event", Gtk.main_quit)
59  win.show_all()
60  Gtk.main()
```

CHAPTER 18

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.

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.

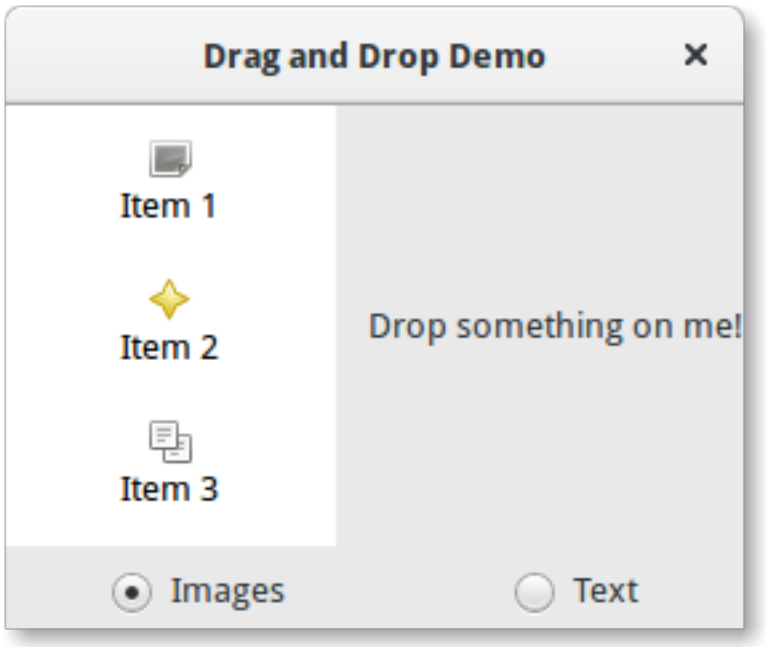
Drag Source Signals

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-end	When the drag is complete	Undo anything done in drag-begin

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

Example




```
import gi
gi.require_version('Gtk', '3.0')
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
```

```
9
10  class DragDropWindow(Gtk.Window):
11
12      def __init__(self):
13          Gtk.Window.__init__(self, title="Drag and Drop Demo")
14
15          vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
16          self.add(vbox)
17
18          hbox = Gtk.Box(spacing=12)
19          vbox.pack_start(hbox, True, True, 0)
20
21          self.iconview = DragSourceIconView()
22          self.drop_area = DropArea()
23
24          hbox.pack_start(self.iconview, True, True, 0)
25          hbox.pack_start(self.drop_area, True, True, 0)
26
27          button_box = Gtk.Box(spacing=6)
28          vbox.pack_start(button_box, True, False, 0)
29
30          image_button = Gtk.RadioButton.new_with_label_from_widget(None,
31              "Images")
32          image_button.connect("toggled", self.add_image_targets)
33          button_box.pack_start(image_button, True, False, 0)
34
35          text_button = Gtk.RadioButton.new_with_label_from_widget(image_button,
36              "Text")
37          text_button.connect("toggled", self.add_text_targets)
38          button_box.pack_start(text_button, True, False, 0)
39
40          self.add_image_targets()
41
42      def add_image_targets(self, button=None):
43          targets = Gtk.TargetList.new([])
44          targets.add_image_targets(TARGET_ENTRY_PIXBUF, True)
45
46          self.drop_area.drag_dest_set_target_list(targets)
47          self.iconview.drag_source_set_target_list(targets)
48
49      def add_text_targets(self, button=None):
50          self.drop_area.drag_dest_set_target_list(None)
51          self.iconview.drag_source_set_target_list(None)
52
53          self.drop_area.drag_dest_add_text_targets()
54          self.iconview.drag_source_add_text_targets()
55
56  class DragSourceIconView(Gtk.IconView):
57
58      def __init__(self):
59          Gtk.IconView.__init__(self)
60          self.set_text_column(COLUMN_TEXT)
61          self.set_pixbuf_column(COLUMN_PIXBUF)
62
63          model = Gtk.ListStore(str, GdkPixbuf.Pixbuf)
64          self.set_model(model)
65          self.add_item("Item 1", "image-missing")
66          self.add_item("Item 2", "help-about")
```

```
        self.add_item("Item 3", "edit-copy")

        self.enable_model_drag_source(Gdk.ModifierType.BUTTON1_MASK, [],
            DRAG_ACTION)
        self.connect("drag-data-get", self.on_drag_data_get)

    def on_drag_data_get(self, widget, drag_context, data, info, time):
        selected_path = self.get_selected_items()[0]
        selected_iter = self.get_model().get_iter(selected_path)

        if info == TARGET_ENTRY_TEXT:
            text = self.get_model().get_value(selected_iter, COLUMN_TEXT)
            data.set_text(text, -1)
        elif info == TARGET_ENTRY_PIXBUF:
            pixbuf = self.get_model().get_value(selected_iter, COLUMN_PIXBUF)
            data.set_pixbuf(pixbuf)

    def add_item(self, text, icon_name):
        pixbuf = Gtk.IconTheme.get_default().load_icon(icon_name, 16, 0)
        self.get_model().append([text, pixbuf])


class DropArea(Gtk.Label):

    def __init__(self):
        Gtk.Label.__init__(self, "Drop something on me!")
        self.drag_dest_set(Gtk.DestDefaults.ALL, [], DRAG_ACTION)

        self.connect("drag-data-received", self.on_drag_data_received)

    def on_drag_data_received(self, widget, drag_context, x,y, data,info, time):
        if info == TARGET_ENTRY_TEXT:
            text = data.get_text()
            print("Received text: %s" % text)

        elif info == TARGET_ENTRY_PIXBUF:
            pixbuf = data.get_pixbuf()
            width = pixbuf.get_width()
            height = pixbuf.get_height()

            print("Received pixbuf with width %spx and height %spx" % (width,
                height))

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

CHAPTER 19

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.

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.

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()
```

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:

```
from gi.repository import Gtk

class Handler:
    def onDeleteWindow(self, *args):
        Gtk.main_quit(*args)

    def onButtonPressed(self, button):
        print("Hello World!")

builder.connect_signals(Handler())
```

Example

The final code of the example

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class Handler:
```

```
6       def onDeleteWindow(self, *args):
7           Gtk.main_quit(*args)
8
9       def onButtonPressed(self, button):
10          print("Hello World!")
11
12  builder = Gtk.Builder()
13  builder.add_from_file("builder_example.glade")
14  builder.connect_signals(Handler())
15
16  window = builder.get_object("window1")
17  window.show_all()
18
19  Gtk.main()
```

CHAPTER 20

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.

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)
```

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.

Receive signals

See *Main loop and Signals*

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
```

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 has 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.

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()`.

Create new properties

A property is defined with a name and a type. Even if 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.ParamFlags.READABLE* (only read access for external code), *GObject.ParamFlags.WRITABLE* (only write access), *GObject.ParamFlags.READWRITE* (public):

```
foo = GObject.Property(type=str, flags = GObject.ParamFlags.READABLE) # not writable
bar = GObject.Property(type=str, flags = GObject.ParamFlags.WRITABLE) # not 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"))
```

The API of `GObject.Property` is similar to the builtin `property()`. You can create property setter in a way similar to Python property:

```
class AnotherObject(GObject.Object):
    value = 0

    @GObject.Property
    def prop(self):
        'Read only property.'
        return 1

    @GObject.Property(type=int)
    def propInt(self):
        'Read-write integer property.'
        return self.value

    @propInt.setter
```

```
    def propInt(self, value):
        self.value = value
```

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.ParamFlags.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.

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
```

Note that you have to use the canonical property name when connecting to the notify signals, as explained in `GObject.Object.signals.notify()`. For instance, for a Python property *foo_bar_baz* you would connect to the signal *notify::foo-bar-baz* using

```
my_object = MyObject()

def on_notify_foo_bar_baz(obj, gparamstring):
    print("foo_bar_baz changed")

my_object.connect("notify::foo-bar-baz", on_notify_foo_bar_baz)
```

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.ParamFlags.`**READABLE**
The property is readable.

`GObject.ParamFlags.`**WRITABLE**
The property is writable.

GObject.ParamFlags.**READWRITE**
The property is readable and writable.

CHAPTER 21

Application

`Gtk.Application` encompasses many repetitive tasks that a modern application needs such as handling multiple instances, D-Bus activation, opening files, command line parsing, startup/shutdown, menu management, window management, and more.

Actions

`Gio.Action` is a way to expose any single task your application or widget does by a name. These actions can be disabled/enabled at runtime and they can either be activated or have a state changed (if they contain state).

The reason to use actions is to separate out the logic from the UI. For example this allows using a menubar on OSX and a gear menu on GNOME both simply referencing the name of an action. The main implementation of this you will be using is `Gio.SimpleAction` which will be showed off later.

Many classes such as `Gio.MenuItem` and `Gtk.ModelButton` support properties to set an action name.

These actions can be grouped together into a `Gio.ActionGroup` and when these groups are added to a widget with `Gtk.Widget.insert_action_group()` they will gain a prefix. Such as "win" when added to a `Gtk.ApplicationWindow`. You will use the full action name when referencing it such as "app.about" but when you create the action it will just be "about" until added to the application.

You can also very easily make keybindings for actions by setting the *accel* property in the `Gio.Menu` file or by using `Gtk.Application.add_accelerator()`.

Menus

Your menus should be defined in XML using `Gio.Menu` and would reference the previously mentioned actions you defined. `Gtk.Application` allows you to set a menu either via `Gtk.Application.set_app_menu()` or `Gtk.Application.set_menubar()`. If you make use of `Gio.Resource` this can automatically use the correct menu based on platform, otherwise you can set them manually. A detailed example is shown below.

Command Line

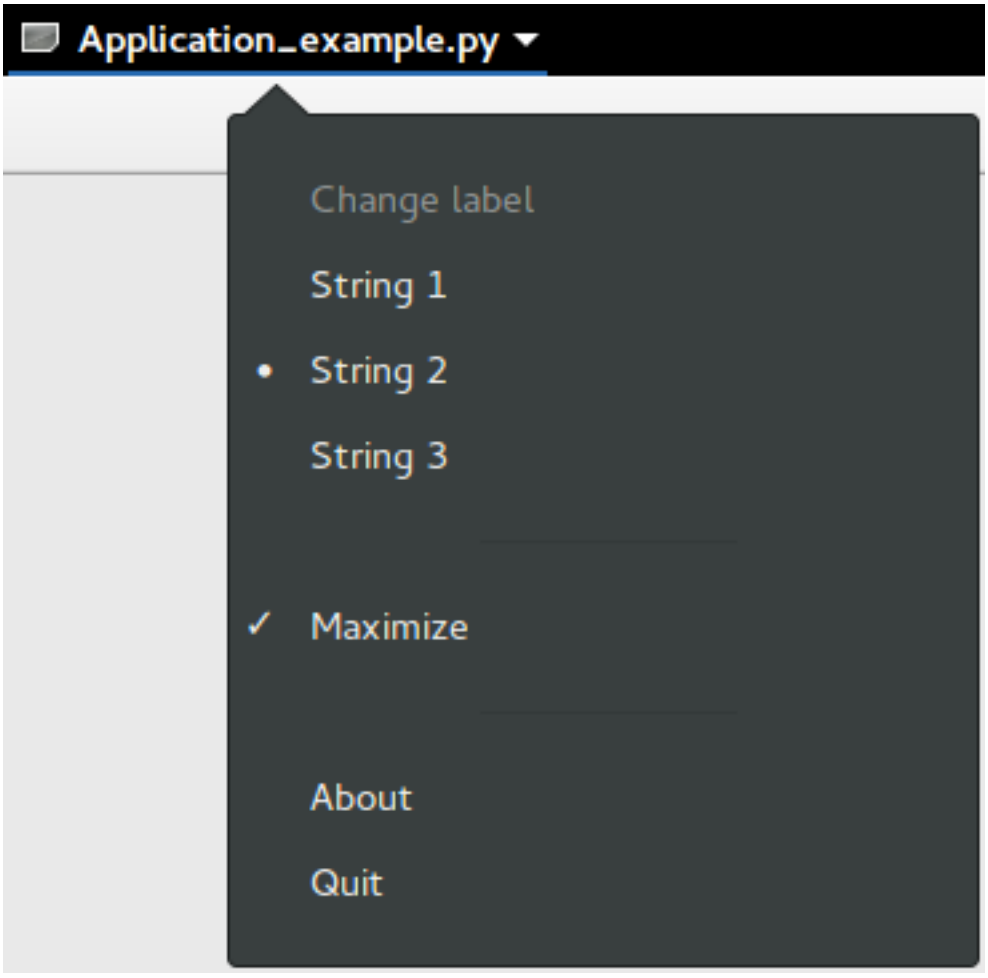
When creating your application it takes a flag property of `Gio.ApplicationFlags`. Using this you can let it handle everything itself or have more custom behavior.

You can use *HANDLES_COMMAND_LINE* to allow custom behavior in `Gio.Application.do_command_line()`. In combination with `Gio.Application.add_main_option()` to add custom options.

Using *HANDLES_OPEN* will do the work of simply taking file arguments for you and let you handle it in `Gio.Application.do_open()`.

If your application is already open these will all be sent to the existing instance unless you use *NON_UNIQUE* to allow multiple instances.

Example




```
1 import sys
2
3 import gi
4 gi.require_version('Gtk', '3.0')
5 from gi.repository import GLib, Gio, Gtk
```

```
6
7  # This would typically be its own file
8  MENU_XML="""
9  <?xml version="1.0" encoding="UTF-8"?>
10 <interface>
11   <menu id="app-menu">
12     <section>
13       <attribute name="label" translatable="yes">Change label</attribute>
14       <item>
15         <attribute name="action">win.change_label</attribute>
16         <attribute name="target">String 1</attribute>
17         <attribute name="label" translatable="yes">String 1</attribute>
18       </item>
19       <item>
20         <attribute name="action">win.change_label</attribute>
21         <attribute name="target">String 2</attribute>
22         <attribute name="label" translatable="yes">String 2</attribute>
23       </item>
24       <item>
25         <attribute name="action">win.change_label</attribute>
26         <attribute name="target">String 3</attribute>
27         <attribute name="label" translatable="yes">String 3</attribute>
28       </item>
29     </section>
30     <section>
31       <item>
32         <attribute name="action">win.maximize</attribute>
33         <attribute name="label" translatable="yes">Maximize</attribute>
34       </item>
35     </section>
36     <section>
37       <item>
38         <attribute name="action">app.about</attribute>
39         <attribute name="label" translatable="yes">_About</attribute>
40       </item>
41       <item>
42         <attribute name="action">app.quit</attribute>
43         <attribute name="label" translatable="yes">_Quit</attribute>
44         <attribute name="accel">&lt;Primary&gt;q</attribute>
45     </item>
46     </section>
47   </menu>
48 </interface>
49 """
50
51 class AppWindow(Gtk.ApplicationWindow):
52
53     def __init__(self, *args, **kwargs):
54         super().__init__(*args, **kwargs)
55
56         # This will be in the windows group and have the "win" prefix
57         max_action = Gio.SimpleAction.new_stateful("maximize", None,
58                                                    GLib.Variant.new_boolean(False))
59         max_action.connect("change-state", self.on_maximize_toggle)
60         self.add_action(max_action)
61
62         # Keep it in sync with the actual state
63         self.connect("notify::is-maximized",
```

```
                            lambda obj, pspec: max_action.set_state(
                                                   GLib.Variant.new_boolean(obj.props.is_
↪maximized)))

        lbl_variant = GLib.Variant.new_string("String 1")
        lbl_action = Gio.SimpleAction.new_stateful("change_label", lbl_variant.get_
↪type(),
                                                   lbl_variant)
        lbl_action.connect("change-state", self.on_change_label_state)
        self.add_action(lbl_action)

        self.label = Gtk.Label(label=lbl_variant.get_string(),
                               margin=30)
        self.add(self.label)
        self.label.show()

    def on_change_label_state(self, action, value):
        action.set_state(value)
        self.label.set_text(value.get_string())

    def on_maximize_toggle(self, action, value):
        action.set_state(value)
        if value.get_boolean():
            self.maximize()
        else:
            self.unmaximize()

class Application(Gtk.Application):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, application_id="org.example.myapp",
                         flags=Gio.ApplicationFlags.HANDLES_COMMAND_LINE,
                         **kwargs)
        self.window = None

        self.add_main_option("test", ord("t"), GLib.OptionFlags.NONE,
                             GLib.OptionArg.NONE, "Command line test", None)

    def do_startup(self):
        Gtk.Application.do_startup(self)

        action = Gio.SimpleAction.new("about", None)
        action.connect("activate", self.on_about)
        self.add_action(action)

        action = Gio.SimpleAction.new("quit", None)
        action.connect("activate", self.on_quit)
        self.add_action(action)

        builder = Gtk.Builder.new_from_string(MENU_XML, -1)
        self.set_app_menu(builder.get_object("app-menu"))

    def do_activate(self):
        # We only allow a single window and raise any existing ones
        if not self.window:
            # Windows are associated with the application
            # when the last one is closed the application shuts down
            self.window = AppWindow(application=self, title="Main Window")
```

```
120
121         self.window.present()
122
123     def do_command_line(self, command_line):
124         options = command_line.get_options_dict()
125
126         if options.contains("test"):
127             # This is printed on the main instance
128             print("Test argument recieved")
129
130         self.activate()
131         return 0
132
133     def on_about(self, action, param):
134         about_dialog = Gtk.AboutDialog(transient_for=self.window, modal=True)
135         about_dialog.present()
136
137     def on_quit(self, action, param):
138         self.quit()
139
140 if __name__ == "__main__":
141     app = Application()
142     app.run(sys.argv)
```

See Also

- https://wiki.gnome.org/HowDoI/GtkApplication
- https://wiki.gnome.org/HowDoI/GAction
- https://wiki.gnome.org/HowDoI/ApplicationMenu
- https://wiki.gnome.org/HowDoI/GMenu

CHAPTER 22

Menus

Note: `Gtk.UIManager`, `Gtk.Action`, and `Gtk.ActionGroup` have been deprecated since GTK+ version 3.10 and should not be used in newly-written code. Use the *Application* framework instead.

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.

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 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.

UI Manager

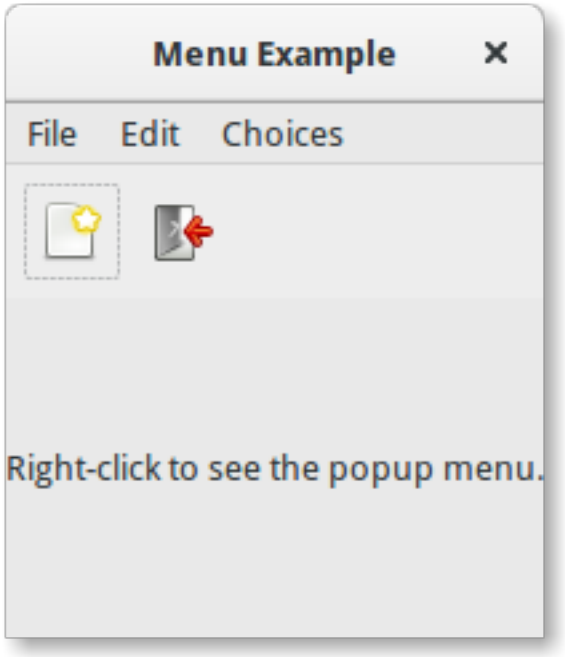
`Gtk.UIManager` provides an easy way of creating menus and toolbars using an XML-like description.

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`.

Example




```
import gi
gi.require_version('Gtk', '3.0')
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),
        ])

        action_filequit = Gtk.Action("FileQuit", None, None, Gtk.STOCK_QUIT)
        action_filequit.connect("activate", self.on_menu_file_quit)
        action_group.add_action(action_filequit)

    def add_edit_menu_actions(self, action_group):
        action_group.add_actions([
            ("EditMenu", None, "Edit"),
            ("EditCopy", Gtk.STOCK_COPY, None, None, None,
             self.on_menu_others),
            ("EditPaste", Gtk.STOCK_PASTE, None, None, None,
             self.on_menu_others),
            ("EditSomething", None, "Something", "<control><alt>S", None,
             self.on_menu_others)
        ])

    def add_choices_menu_actions(self, action_group):
        action_group.add_action(Gtk.Action("ChoicesMenu", "Choices", None,
            None))

        action_group.add_radio_actions([
            ("ChoiceOne", None, "One", None, None, 1),
            ("ChoiceTwo", None, "Two", None, None, 2)
        ], 1, self.on_menu_choices_changed)

        three = Gtk.ToggleAction("ChoiceThree", "Three", None, None)
        three.connect("toggled", self.on_menu_choices_toggled)
        action_group.add_action(three)

    def create_ui_manager(self):
        uimanager = Gtk.UIManager()

        # Throws exception if something went wrong
        uimanager.add_ui_from_string(UI_INFO)

        # Add the accelerator group to the toplevel window
        accelgroup = uimanager.get_accel_group()
        self.add_accel_group(accelgroup)
        return uimanager

    def on_menu_file_new_generic(self, widget):
        print("A File|New menu item was selected.")

    def on_menu_file_quit(self, widget):
        Gtk.main_quit()

    def on_menu_others(self, widget):
        print("Menu item " + widget.get_name() + " was selected")

    def on_menu_choices_changed(self, widget, current):
        print(current.get_name() + " was selected.")

    def on_menu_choices_toggled(self, widget):
        if widget.get_active():
```

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

CHAPTER 23

Table

Note: `Gtk.Table` has been deprecated since GTK+ version 3.4 and should not be used in newly-written code. Use the *Grid* class instead.

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.

Deprecated since version 3.4: It is reccomened that you use the `Gtk.Grid` for new code.

Example


```
import gi
gi.require_version('Gtk', '3.0')
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 24

Indices and tables

- search

Index

Symbols

E

F

G

H

R

S

T

W